VMCI Sockets Programming Guide

for VMware Workstation 6.5 and VMware Server 2.0

VMCI Sockets Programming Guide
Revision: 20080815
Item: EN-000054-00

You can find the most up-to-date technical documentation on our 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

© 2008 VMware, Inc. All rights reserved. Protected by one or more U.S. Patent Nos. 6,397,242, 6,496,847, 6,704,925, 6,711,672, 6,725,289, 6,735,601, 6,785,886, 6,789,156, 6,795,966, 6,880,022, 6,944,699, 6,961,806, 6,961,941, 7,069,413, 7,082,598, 7,089,377, 7,111,086, 7,111,145, 7,117,481, 7,149,843, 7,155,558, 7,222,221, 7,260,815, 7,260,820, 7,269,683, 7,275,136, 7,277,998, 7,277,999, 7,278,030, 7,281,102, 7,290,253, 7,356,679 7,409,487, 7,412,492, and 7,412,702; patents pending.

VMware, the VMware "boxes" logo and design, Virtual SMP and VMotion are registered trademarks or trademarks 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
- Document Feedback 5

1 About VMCI Sockets 7
- Introducing VMCI Sockets 7
  - How VMCI Sockets Work 8
  - Efficiency Compared to Network Sockets 8
  - Possible Use Cases 8
- Contents of This Release 8
  - Experimental VMCI Library Deprecated 8
- Illustrated Connections with VMCI Sockets 9
  - Web Access with Stream VMCI Sockets 9
  - Home Directories with Datagram VMCI Sockets 10
- Porting Existing Socket Applications 10
  - Including a New Header File 10
  - Changing AF_INET to VMCI Sockets 10
    - Obtaining the Context ID (CID) 10
    - The VMCISock_GetLocalCID() Function 11
  - Connection-oriented Stream Socket 11
  - Connectionless Datagram Socket 11
  - Initializing the Address Structure 11
  - Limitations on Persistence 11
- Communicating Between Guests 12
  - VMCI Sockets and the Network Stack 12
  - Setting up a Networkless Guest 12
- Communicating between Guest and Host 12
  - Using UDP Datagram Sockets 12

2 Using VMCI Sockets 13
- Stream VMCI Sockets 13
  - Preparing the Server for a Connection 14
    - Socket() Function Call 14
    - Set and Get Socket Options 14
    - Bind() Function Call 15
    - Listen() Function Call 15
    - Accept() Function Call 15
    - Select() Function Call 15
    - Recv() Function Call 15
    - Send() Function Call 16
    - Close() Function Call 16
    - Poll() Information 16
    - Read() and Write() 16
    - Getsockname() Function 16
  - Having the Client Request a Connection 16
    - Socket() Function Call 16
    - Connect() Function Call 17
    - Send() Function Call 17

Recv() Function Call 17
Close() Function Call 17
Poll() Information 17
Read() and Write() 17
Datagram VMCI Sockets 18
Preparing the Server for a Connection 18
Socket() Function 18
Socket Options 18
Bind() Function 19
Getsockname() Function 19
Recvfrom() Function 19
Sendto() Function 19
Close() Function 19
Having the Client Request a Connection 19
Socket() Function 20
Sendto() Function 20
Connect() and Send() 20
Recvfrom() Function 20
Close() Function 20

A Learning About TCP Sockets 21
Hidden Information 21
Resources on the Web 21
Wikipedia 21
IBM 21
Sockaddr 21
MSDN 21
Linux Man Pages 22
Hardcopy Books 22

Glossary 23

Index 25

About This Book

The *VMCI Sockets Programming Guide* describes how to create and program virtual machine communications interface (VMCI) sockets, a familiar API to facilitate fast and efficient communication between a host and its virtual machines.

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
20080327	First draft of this manual for possible inclusion in early beta releases.
20080620	Second draft for VMware Workstation 6.5 Beta 2 and VMware Server 2.0 RC1 releases.
20080815	Third draft, with socket options, for VMware Workstation 6.5 RC release.

Intended Audience

This guide is intended for programmers planning to develop applications with VMCI sockets to create C or C++ networking applications that target guest operating systems on VMware hosts, or virtual machine communications. Because VMCI sockets are based on TCP sockets, this guide assumes that you are familiar with either Berkeley sockets or Winsock, the Windows implementation.

API and SDK Documentation

VMware provides many different products targeting different developer communities and platforms. For the most up-to-date information about API and SDK products, this is the place to go:

http://www.vmware.com/support/pubs/sdk_pubs.html

Document Feedback

VMware welcomes your suggestions for improving our documentation. Send your comments to:

docfeedback@vmware.com

Technical Support and Education Resources

The following sections describe the technical support resources available to you. You can access the most current versions of other VMware manuals by going to:

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

Online Support

You can submit questions or post comments to the Developer Community: SDKs and APIs forum, which the VMware technical support and product teams monitor. You can access the forum at:

http://communities.vmware.com/community/developer

Support Offerings

To find out how VMware support offerings can help meet your business needs, go to:

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

VMware Education Services

VMware courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. For more information about VMware Education Services, go to:

http://mylearn1.vmware.com/mgrreg/index.cfm

About VMCI Sockets 1

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 universal by their incorporation into Windows.

VMware now offers a similar implementation of virtual machine communications interface (VMCI) Sockets.

This chapter contains the following sections:

- "Introducing VMCI Sockets" on page 7
- "Contents of This Release" on page 8
- "Illustrated Connections with VMCI Sockets" on page 9
- "Porting Existing Socket Applications" on page 10
- "Communicating Between Guests" on page 12
- "Communicating between Guest and Host" on page 12

This manual assumes that you know about either Berkeley sockets or Winsock, the Windows implementation. If you are new to sockets, see Appendix A, "Learning About TCP Sockets," on page 21.

Socket-based communications usually employ a client-server approach. One application (the server) tries to make itself always available while another (the client) requests services as needed.

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

Introducing VMCI Sockets

The VMware VMCI sockets library offers a familiar API to support fast and efficient communication between a virtual machine and its host, or between virtual machines on the same host.

VMCI sockets are similar to other socket types. Like Unix (local) sockets, VMCI sockets work on a discrete physical machine, while Unix sockets perform interprocess communication on a discrete local system. With Internet sockets, communicating processes usually reside on different systems across a network. Like Internet sockets, VMCI sockets allow different virtual machines to communicate with each other, provided they reside on the same VMware host.

If you have existing socket-based applications, 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.

Re-purposing any networking program to use VMCI sockets requires minimal effort, because VMCI sockets behave like traditional Internet sockets on a given platform. Some socket options do not make sense for communication across the VMCI device; such options are silently ignored to ease program portability.

Modification is straightforward. You include a new header file. Before the `socket()` function call, you call the `VMCISock_GetAFValue()` function to return the VMCI sockets address family to replace `AF_INET`. You also allocate a different socket addressing structure, `sockaddr_vm` instead of `sockaddr_in`.

Otherwise VMCI sockets use the same API as Berkeley sockets or Windows sockets.

How VMCI Sockets Work

VMCI sockets communicate from guest to guest, or guest to host, on one VMware host. They can also be used 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 host across a network.

Communicating guest virtual machines must be running, not powered off.

Efficiency Compared to Network Sockets

Early performance testing indicates VMCI sockets have low latency and high throughput. Socket endpoints communicate with each other at, or close to, the speed of memory.

Because communication is on a single physical system, VMCI sockets can attain performance comparable to Unix sockets even though they enable communication across virtual machines like Internet sockets do.

VMware tailored its VMCI sockets implementation for high performance, especially with large data sets. VMware recommends message sizes larger than 512 bytes to fully realize the performance benefits. On a discrete physical machine, VMCI sockets are higher performance than over-the-wire network sockets, as you would expect.

Interprocess communication implies data transfer among processes on the same system. VMCI sockets support this. VMCI sockets also allow communication among processes on different systems, even ones running different versions and types of operating systems. This combines the best aspects of interprocess communication with the advantages of networking in a hosted virtual environment.

Possible Use Cases

Here are some potential applications for VMCI sockets:

- Improved intra-host performance for socket-modified applications.
- Choice of stream or datagram communication for off-the-network virtual machines.
- Increased privacy of communications for hosted virtual machines.
- Alternate data path for administrative control of guest virtual machines.
- Database-backed applications can be made more efficient when going off-box for data.
- A host-guest file system can be implemented.

Contents of This Release

In the VMware Server 2.0 and VMware Workstation 6.5 releases, 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 VMware 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.

Experimental VMCI Library Deprecated

The VMCI library was released as an experimental C language interface with Workstation 6.0. A `README` file in the VMCI directory and a document on the VMware Web site described how to enable the facility and develop applications with it. The experimental VMCI library offered programmers two choices: a datagram API and a shared memory API. Both these interfaces have been deprecated.

In this release, an API with familiar sockets interface and functionality is available. This new library replaces the original shared memory implementation and has more flexible algorithms, wrapped in a stream sockets interface for external presentation. Similarly, an interface with datagram sockets replace VMCI datagrams. The new VMCI sockets interfaces are now supported, not experimental.

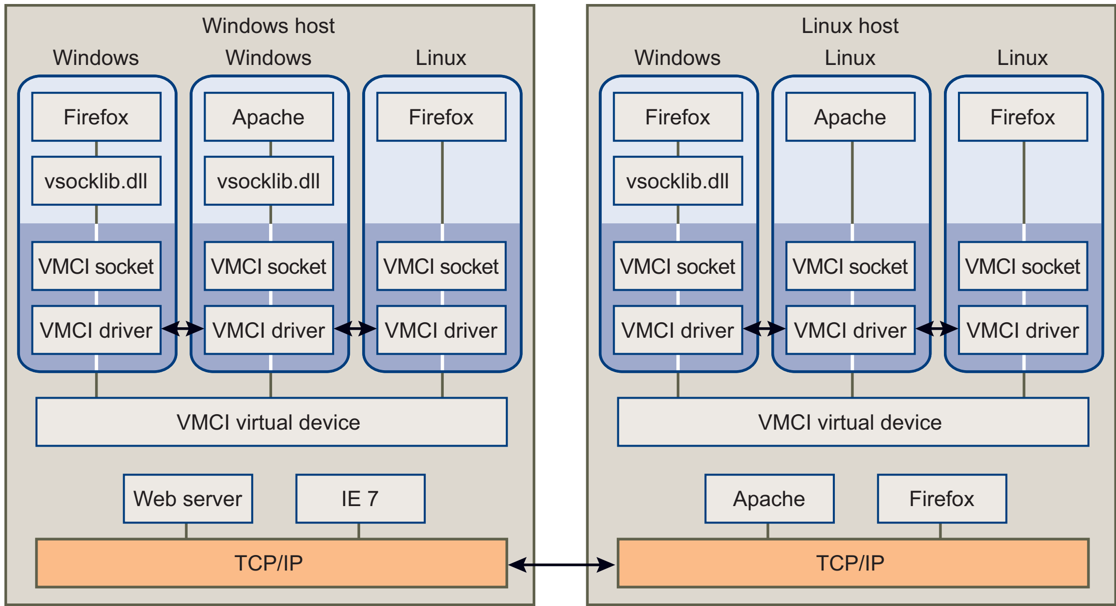
Illustrated Connections with VMCI Sockets

The illustrations in this section are merely examples of what can be done using VMCI sockets. VMware does not provide modified versions of any third-party applications shown here. However, open-source versions of Firefox, Apache, and NFS are available, so VMCI sockets modification is feasible.

Web Access with Stream VMCI Sockets

Figure 1-1 shows two 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 TCP/IP networking.

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



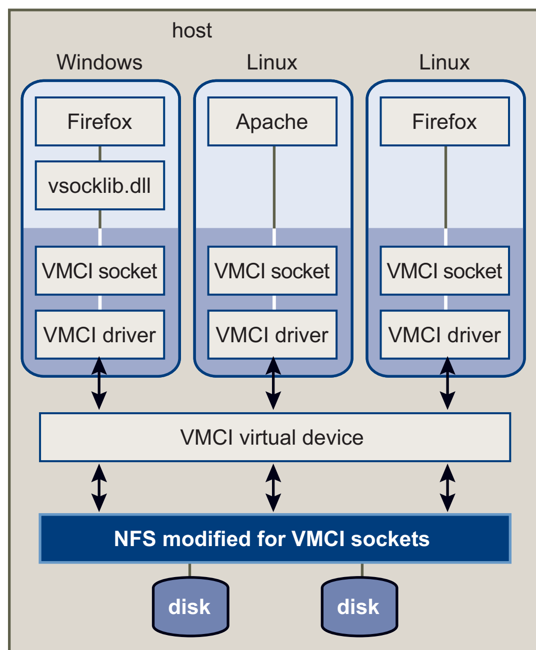
When the Firefox browsers on Linux and Windows request a connection to the Apache Web server, the VMCI sockets layer creates a socket endpoint and establishes a connection through the VMCI driver and virtual device. The VMCI sockets layer on the system with Apache receive the connection and provide an 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 others' Web servers over a standard TCP/IP network connection. If guest operating systems needed to access the Web outside the physical machine, they would have to use different (unmodified) Web browsers, or have a fallback capability, outside of VMCI sockets.

Home Directories with Datagram VMCI Sockets

Figure 1-2 shows 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.

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



Porting Existing Socket Applications

This section discusses the major differences between Berkeley sockets and VMCI sockets, highlighting the items you need to change in your existing socket applications.

Including a New Header File

To obtain the VMware definitions for VMCI sockets, include the new `vmci_sockets.h` header file, which is packaged with VMware Tools.

```
#include “vmci_sockets.h”
```

Changing AF_INET to VMCI Sockets

Call the `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 of a virtual machine.

Obtaining the Context ID (CID)

In hardware version 6 (Workstation 6.0.x releases), the VMCI virtual device is not present by default. After you upgrade the virtual hardware to version 7, the following line appears in the `.vmx` configuration file:

```
vmci0.present = “TRUE”
```

When the virtual machine powers on, a new `vmci0.id` line appears added to the configuration file.

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

On the `vmci0.id` line, context ID is the hexadecimal number in double quotes.

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

The host usually has CID = 2.

The VMCISock_GetLocalCID() Function

For convenience, the `VMCISock_GetLocalCID()` function returns the local system's CID. It works on both host server and guest virtual machines.

Connection-oriented Stream Socket

To establish a stream socket, the source code includes these declarations and calls:

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

`AF_INET` has been replaced by `afVMCI` as set by `VMCISock_GetAFValue()`.

Connectionless Datagram Socket

To establish a datagram socket, the source code includes these declarations and calls:

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

Again, `AF_INET` has been replaced by `afVMCI` as set by `VMCISock_GetAFValue()`.

Initializing the Address Structure

To initialize the address structure passed to `bind()`, the source code includes these statements; `sockaddr_vm` replaces `sockaddr_in` used 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` has been replaced by `afVMCI` as above. `VMADDR_CID_ANY` and `VMADDR_PORT_ANY` are predefined so that at runtime on the server, the appropriate context ID and port values can be filled in during the bind operation.

The initiating side of the connection, usually the client, must provide the proper context ID and port, instead of `VMADDR_CID_ANY` and `VMADDR_PORT_ANY`.

Limitations on Persistence

VMCI sockets connections are dropped after suspend and resume of a virtual machine.

Connections cannot survive live migration with VMotion from source to destination host.

Communicating Between Guests

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

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

VMCI Sockets and the Network Stack

Most virtual machines are installed with networking enabled, but if limited access is sufficient, VMCI sockets could replace TCP networking, saving memory and processor bandwidth by disabling the network stack.

Typically however, networking is enabled. 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 is incapable of connecting 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.

You create a networkless guest with the option “Do not use a network connection” in Workstation wizard. After installing VMware Tools, you can use VMCI sockets to communicate with the networkless guest.

After its creation, you can also 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 the host, you can establish a VMCI sockets connection using the `SOCK_DGRAM` socket type.

Using UDP Datagram Sockets

The VMCI sockets datagram offers an alternative to the `AF_INET` datagram.

Using VMCI Sockets

2

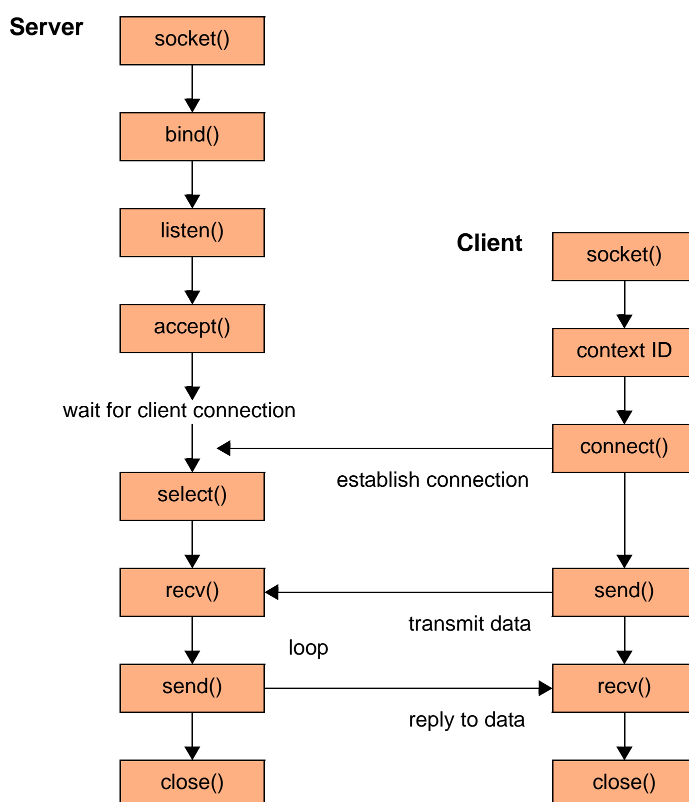
This chapter describes the details of creating VMCI sockets to replace TCP sockets, in two sections:

- "Stream VMCI Sockets" on page 13
- "Datagram VMCI Sockets" on page 18

Stream VMCI Sockets

The flowchart in Figure 2-1 shows how to establish connection-oriented sockets on server and client.

Figure 2-1. 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 though the attached socket. In VMCI sockets, a virtual socket can have only two endpoints, and the server cannot initiate a connection. In TCP sockets, more than two endpoints are possible, though rare, and the server can initiate connections. Otherwise the protocols are identical.

Programmers use stream sockets for their high reliability.

Preparing the Server for a Connection

At the top of the program, 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
```

For Winsock you must call the 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 Call

To alter a TCP sockets program for VMCI sockets, 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;
}
```

The 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, include 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 Call

The `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 context ID (CID) to specify the virtual machine. For the server (listener) this could be any connecting virtual machine. `VMADDR_CID_ANY` and `VMADDR_PORT_ANY` are predefined so that at bind or connection time, the appropriate context ID and port number are filled in from the client. `VMADDR_CID_ANY` is replaced with the context ID of the virtual machine and `VMADDR_PORT_ANY` provides an ephemeral port from the non-reserved range (>= 1024).

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

The `bind()` function itself should be unchanged from a regular TCP sockets application.

Listen() Function Call

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 should be unchanged from the `listen()` in a regular TCP sockets application.

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

Accept() Function Call

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

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

Select() Function Call

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 and microseconds, at which point the function will return zero. The read, write, and exception file descriptors can be specified as `NULL` if the program can safely ignore them.

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

Recv() Function Call

The `recv()` call reads data from the client application. 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 Call

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 Call

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. With Winsock, call 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 should be required. See "Select() Function Call" 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 the program, 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
```

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

Socket() Function Call

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

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

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

Connect() Function Call

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

```
struct sockaddr_vm their_addr = {0};
their_addr.svm_family = afVMCI;
their_addr.svm_cid = VMCISock_GetLocalCID();
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 context ID (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 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 Call

The send() call writes data to the server application. 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 Call

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. On the client, 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 Call

The close() call shuts down a connection, given the original socket descriptor obtained from the socket() call. With Winsock, call 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 should be required.

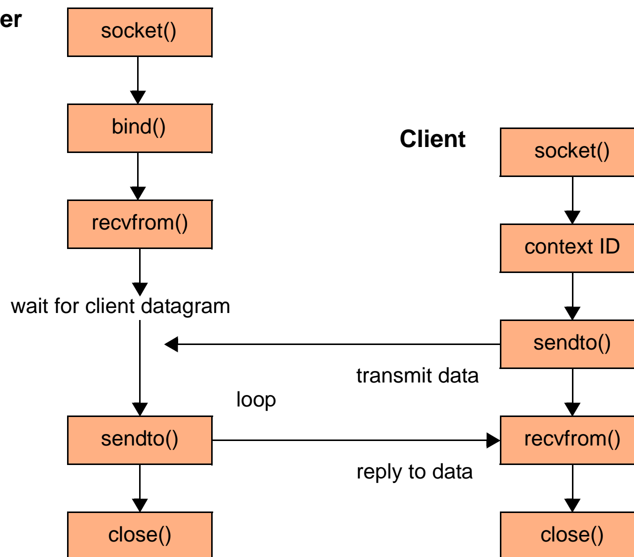
Read() and Write()

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

Datagram VMCI Sockets

The flowchart in Figure 2-2 shows how to establish connectionless sockets on server and client.

Figure 2-2. 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.

Programmers use datagram sockets for their speed and low overhead.

Preparing the Server for a Connection

At the top of the program, 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 2048
```

For Winsock you must call the `WSAStartup()` function.

```
    WSA wsaData;
    WORD versionRequested = MAKEWORD(2, 0);
    return WSAStartup(versionRequested, &wsaData);
```

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;
    }
```

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

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

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 context ID (CID) to specify the virtual machine. For the server (listener) this could be any connecting virtual machine. `VMADDR_CID_ANY` and `VMADDR_PORT_ANY` are predefined so that at bind or connection time, the appropriate context ID and port number are filled in from the client. `VMADDR_CID_ANY` is replaced with the context ID of the virtual machine and `VMADDR_PORT_ANY` provides an ephemeral port from the non-reserved range (>= 1024).

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

The `bind()` function itself should be unchanged from a regular UDP datagram application.

Getsockname() Function

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

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

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, &svm_size)) == -1) {
    perror("recvfrom");
    goto close;
}
```

Sendto() Function

The `sendto()` call optionally writes data back to the client application. See "Sendto() Function" on page 20.

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. With Winsock, call `closesocket()` instead of `close()`.

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

Having the Client Request a Connection

At the top of the program, 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 2048
```

For Winsock you must call the WSAStartup() function. See "Preparing the Server for a Connection" on page 18 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 context ID (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 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 19.

Close() Function

The close() call shuts down a connection, given the original socket descriptor obtained from the socket() call. With Winsock, call closesocket() instead of close(), as shown in "Close() Function" on page 19.

Learning About TCP Sockets

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 two endpoints simultaneously initiating connections with one another.

Hidden Information

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` requires `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.

Resources on the Web

Here is a succinct and comprehensible API explanation:

http://www.cas.mcmaster.ca/~qiao/courses/cs3mh3/tutorials/socket.html

Wikipedia

Here is an overview of the history and design of sockets:

http://en.wikipedia.org/wiki/Berkeley_sockets

IBM

Here is a good technical description of programming with sockets:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzab6/rzab6designrec.htm

Sockaddr

Here is an overview of Winsock design and usage:

http://www.sockaddr.com/TheSocketsParadigm.html

MSDN

Here is reference information about Windows sockets, Winsock:

http://msdn2.microsoft.com/en-us/library/ms740673(VS.85).aspx

Linux Man Pages

socket(7) – type `man socket` on a Linux system.

http://www.yo-linux.com/cgi-bin/man2html?cgi_command=socket(2)

http://www.yo-linux.com/cgi-bin/man2html?cgi_command=socket(7)

Hardcopy 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.

Glossary

D

datagram

A self-contained unit of data containing enough information to be routed from its source to its destination without reliance on earlier exchanges between source and destination hosts and the transporting network.

S

socket

A communications connection endpoint with a name and address in a network. The two endpoints of a socket can reside on the same system or on different systems across the network. Sockets are most often used for network applications, but they are also useful for interprocess communications on a single server.

stream socket

A network connection that provides a two-way, sequenced, reliable, unduplicated flow of data without record boundaries, with well-defined mechanisms for establishing connections and detecting errors.

V

VMCI

Virtual machine communication interface, a shared memory API and datagram interface now replaced by VMCI sockets, a stream and datagram sockets interface.

Index

A
about VMCI sockets **7**
accept() **15**
address structure for sockets **11**
AF_INET and PF_INET **21**
AF_INET and VMCI sockets **10**

B
bind() **11**, **15**, **19**
books about sockets **22**

C
close() **16**, **17**, **19**, **20**
connect() **17**
connectionless socket **11**
connection-oriented socket **11**
context ID (CID) summary **10**

D
datagram VMCI sockets **18**

G
getsockname() **16**, **19**
guest to guest **8**, **12**
guest to host **8**, **12**

H
hidden information about sockets **21**
host to guest **8**, **12**

I
illustration of datagram VMCI sockets **10**
illustration of stream VMCI sockets **9**

L
listen() **15**

P
PF_INET and AF_INET **21**
poll() **16**, **17**
porting sockets applications **10**

R
read() **16**, **17**
recv() **15**, **17**
recvfrom() **19**, **20**
release contents **8**

S
select() **15**
send() **16**, **17**
sendto() **19**, **20**
SOCK_DGRAM **12**
SOCK_STREAM **12**
socket() **7**, **11**, **14**, **16**, **18**, **20**
stream VMCI sockets **13**

T
technical support resources **5**

U
use cases for VMCI sockets **8**

V
VMCI library deprecated **8**
VMCISock_GetAFValue() **7**, **14**, **16**, **18**
VMCISock_GetLocalCID() **11**, **15**, **17**, **19**, **20**

W
Web resources about sockets **21**
write() **16**, **17**
WSAStartup() **14**, **18**