CS415 Module 5 Part B - IPC, Shared Memory, and Named Pipes

Athens State University

Outline

Contents

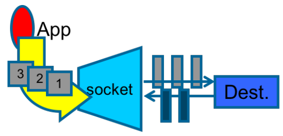
1 Introduction

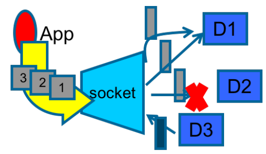
What Is A Socket?

- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured the application can:
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

Types of Sockets

- `SOCK_STREAM`
  - TCP (connection-based)
  - Reliable delivery
  - Bidirectionals
- `SOCK_DGRAM`
  - Use for UDP (broadcast)
  - Unreliable delivery
  - No order guarantees
  - No notion of "connection" - app indicates dest. for each packet
  - Can send or receive





Stream sockets (`SOCK_STREAM`) provide a reliable, bidirectional, byte-stream communication channel. By the terms in this description, we mean the following:

- Reliable means that we are guaranteed that either the transmitted data will arrive intact at the receiving application, exactly as it was transmitted by the sender (assuming that neither the network link nor the receiver crashes), or that we'll receive notification of a probable failure in transmission.
- Bidirectional means that data may be transmitted in either direction between two sockets.
- Byte-stream means that, as with pipes, there is no concept of message boundaries

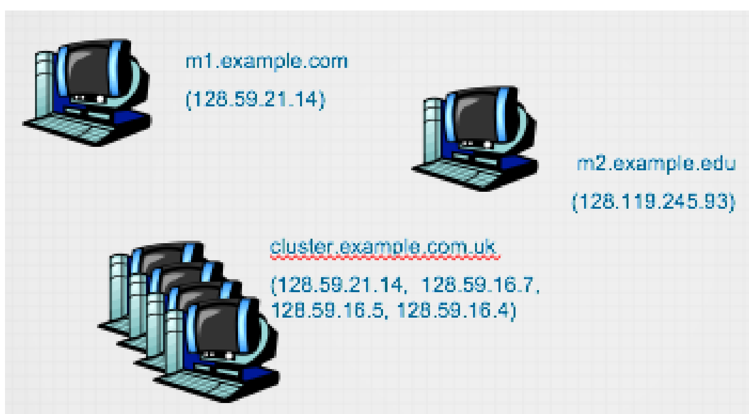
Stream sockets operate in connected pairs. The term *peer socket* refers to the socket at the other end of a connection; peer address denotes the address of that socket; and *peer application* denotes the application utilizing the peer socket. A stream socket can be connected to only one peer.

Using a stream socket is similar to using a named pipe, except we have the ability to communicate over the network.

Datagram sockets (`SOCK_DGRAM`) allow data to be exchanged in the form of mes- sages called datagrams. With datagram sockets, message boundaries are preserved, but data transmission is not reliable. Messages may arrive out of order, be dupli- cated, or not arrive at all.

Datagram sockets are an example of the more generic concept of a connectionless socket. Unlike a stream socket, a datagram socket doesn't need to be connected to another socket in order to be used.

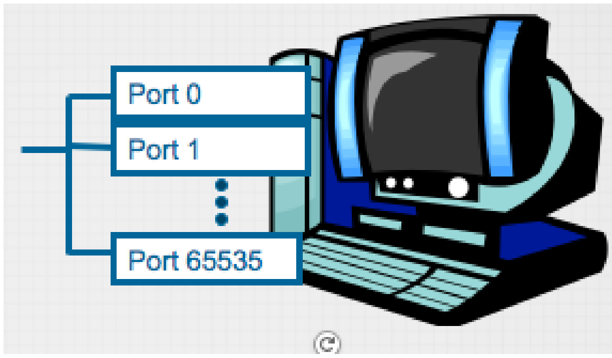
A Socket's View of the Network



- Each machine has an IP address
- Packets are delivered to a port at that IP address

For sockets in the Internet domain, datagram sockets are implemented using UDP and stream sockets with TCP. So, you will hear me often refer to them as *UDP-sockets* and *TCP-sockets*

Ports

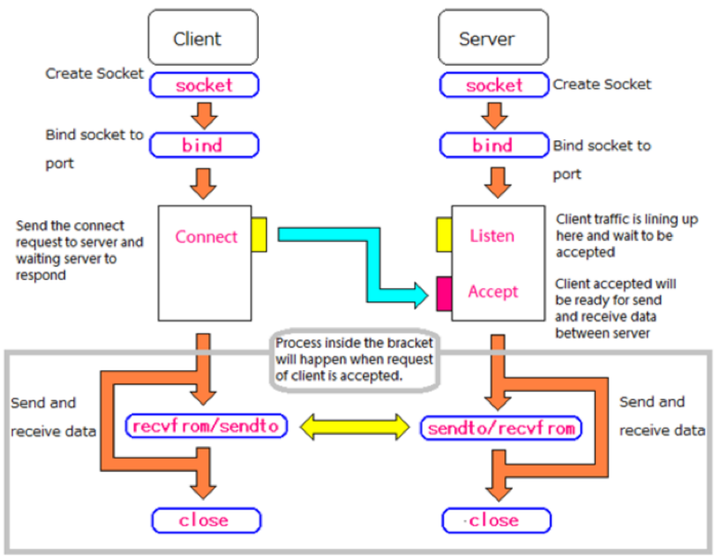


A socket provides an interface to send data to/form the network through a port

- Each host has 65,536 ports
- Some ports are reserved for specific purposes
  - 20,21: FTP
  - 22: SSH
  - 23: Telnet
  - 80: HTTP
  - see RFC1700 (about 2000 ports are reserved)

2 Socket Programming

Socket Programming Workflow



The Five Most Important Socket System Calls

- `socket()`: creates a new socket
- `bind()`: binds a socket to an IP address
- `listen()`: Allows a stream socket to accept incoming connections
- `accept()`: Accepts a connection from a peer application on a listening stream socket
- `connect()`: Establishes a connection with another socket

Stream Socket Application Architecture

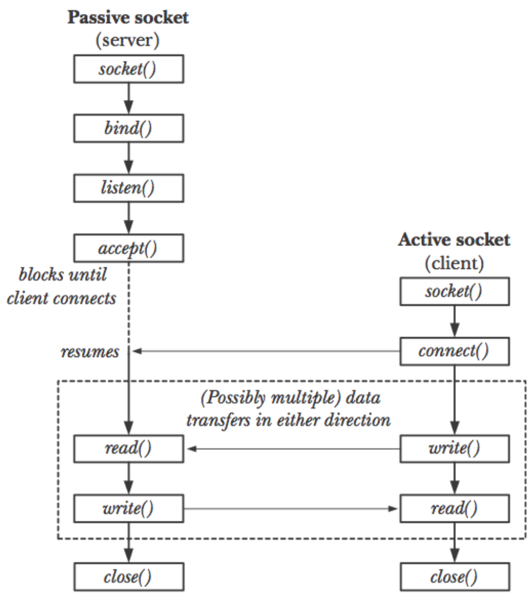


Figure 56-1: Overview of system calls used with stream sockets

The operation of stream sockets can be explained by analogy with the telephone system:

1. The `socket()` system call is the equivalent of installing a telephone. Each side of the communication must create a socket.

2. Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:

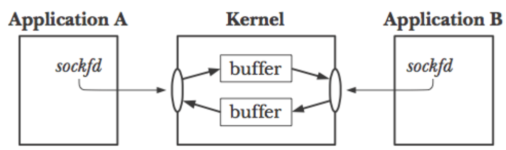
   (a) One application calls `bind()` in order to bind the socket to a well-known address, and then calls `listen()` to notify the kernel of its willingness to accept incoming connections. This step is analogous to having a known telephone number and ensuring that our telephone is turned on so that people can call us.

   (b) The other application establishes the connection by calling `connect()`, specifying the address of the socket to which the connection is to be made. This is analogous to dialing someone's telephone number.

   (c) The application that called `listen()` then accepts the connection using `accept()`. This is analogous to picking up the telephone when it rings. If the `accept()` is performed before the peer application calls `connect()`, then the `accept()` blocks ("waiting by the telephone").

3. Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a two-way telephone conversation) until one of them closes the connection using `close()`. Communication is performed using the conventional `read()` and `write()` system calls or via a number of socket- specific system calls (such as `send()` and `recv()`) that provide additional functionality.

I/O on Stream Sockets



- We use the `read()` and `write()` system calls as if we were working a file
- There are also specialized `send()` and `recv()` system calls for the same use
- Note that sockets are bidirectional... each end may read and write to the socket

Datagram Socket Application Architecture

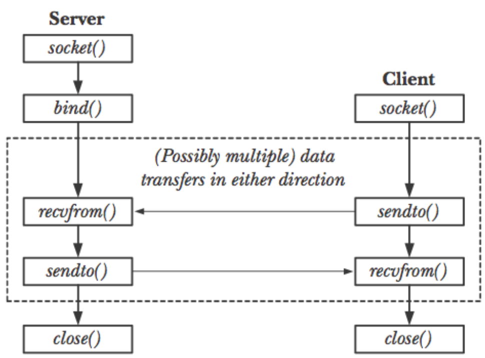


Figure 56-4: Overview of system calls used with datagram sockets

The operation of datagram sockets can be explained by analogy with the postal system:

1. The `socket()` system call is the equivalent of setting up a mailbox. Each application that wants to send or receive datagrams creates a datagram socket using socket().
2. In order to allow another application to send it datagrams (letters), an application uses `bind()` to bind its socket to a well-known address. Typically, a server binds its socket to a well-known address, and a client initiates communication by sending a datagram to that address.
3. To send a datagram, an application calls sendto(), which takes as one of its arguments the address of the socket to which the datagram is to be sent. This is analogous to putting the recipient's address on a letter and posting it.
4. In order to receive a datagram, an application calls `recvfrom()`, which may block if no datagram has yet arrived. Because `recvfrom()` allows us to obtain the address of the sender, we can send a reply if desired. (This is useful if the sender's socket is bound to an address that is not well known, which is typical of a client.)
5. When the socket is no longer needed, the application closes it using `close()`.

Network Byte Order

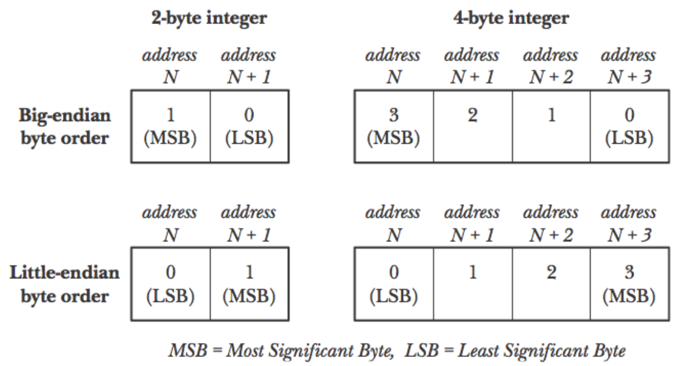


Figure 59-1: Big-endian and little-endian byte order for 2-byte and 4-byte integers

IP addresses and port numbers are integer values. One problem we encounter when passing these values across a network is that different hardware architectures store the bytes of a multibyte integer in different orders. Architectures that store integers with the most significant byte first (i.e., at the lowest memory address) are termed `big-endian`; those that store the least significant byte first are termed `little-endian`. The most notable example of a little-endian architecture is x86.

BTW, The terms "big-endian" and "little-endian" derive from Jonathan Swift's 1726 satirical novel Gulliver's Travels, in which the terms refer to opposing political factions who open their boiled eggs at opposite ends.

3 Case Study: Sockets in C++: Echo Server

Case Study: Sockets in C++: Echo Server

- There are a lot of "fiddlely" bits one must address with socket programming
- This case study looks at a set of C++ classes that does that work for you
- And we demo the use of our classes to build a simple echo server as

Case Study: Sockets in C++: The Server Socket Class

```
class ServerSocket : private Socket
{
 public:
  ServerSocket ( int port );
  ServerSocket (){};
  virtual ~ServerSocket();
  const ServerSocket& operator << ( const std::string& ) const;
  const ServerSocket& operator >> ( std::string& ) const;
  void accept ( ServerSocket& );
};
```

Case Study: Sockets in C++: The Server Socket Class

```cpp
#include "ServerSocket.h"
#include "SocketException.h"
ServerSocket::ServerSocket ( int port ) {
  if ( ! Socket::create() )
      throw SocketException ( "Could not create server socket." );
  if ( ! Socket::bind ( port ) )
      throw SocketException ( "Could not bind to port." );
  if ( ! Socket::listen() )
      throw SocketException ( "Could not listen to socket." );
}
ServerSocket::~ServerSocket() {}
const ServerSocket& ServerSocket::operator << ( const std::string& s ) const {
  if ( ! Socket::send ( s ) )
      throw SocketException ( "Could not write to socket." );
  return *this;
}
const ServerSocket& ServerSocket::operator >> ( std::string& s ) const {
  if ( ! Socket::recv ( s ) )
      throw SocketException ( "Could not read from socket." );
  return *this;
}
void ServerSocket::accept ( ServerSocket& sock ) {
  if ( ! Socket::accept ( sock ) )
      throw SocketException ( "Could not accept socket." );
}
```

Case Study: Sockets in C++: The Socket Base Class

```cpp
class Socket {
 public:
  Socket();
  virtual ~Socket();
  // Server initialization
  bool create();
  bool bind ( const int port );
  bool listen() const;
  bool accept ( Socket& ) const;
  // Client initialization
  bool connect ( const std::string host, const int port );
  // Data Transimission
  bool send ( const std::string ) const;
  int recv ( std::string& ) const;

  void set_non_blocking ( const bool );

  bool is_valid() const { return m_sock != -1; }

 private:
  int m_sock;
  sockaddr_in m_addr;
};
```

Here's the details on the base class:

```cpp
#include "Socket.h"
#include "string.h"
#include <string.h>
#include <iostream>
#include <errno.h>
```

```
#include <fcntl.h>
//
// Socket::Socket()
// Construct a new instance of our class.
// Pre-condition:
// NONE.
//
// Post-condition:
// A cleared space for the address instance variable has been created.
//
Socket::Socket() :
  m_sock ( -1 )
{

  memset ( &m_addr,
      0,
      sizeof ( m_addr ) );

}
//
// Socket::~Socket()
// Destory an existing instance of our class.
// Pre-condition:
// A valide socket is available
//
// Post-condition:
// The socket is closed as needed.
//
Socket::~Socket()
{
  if ( is_valid() )
    ::close ( m_sock );
}
//
// Socket::create()
// Create a new socket and associate it with our instance.
// Pre-condition:
// Instance has been created.
//
// Post-condition:
// A reference to a valid socket can be found in our private variable.
//
bool Socket::create()
{
  m_sock = socket ( AF_INET,
        SOCK_STREAM,
        0 );

  if ( ! is_valid() )
    return false;


  // TIME_WAIT - argh
  int on = 1;
  if ( setsockopt ( m_sock, SOL_SOCKET, SO_REUSEADDR, ( const char* ) &on, sizeof ( on ) )
      == -1 )
    return false;
  return true;

}
//
// Socket::bind()
// Create a new socket and associate it with our instance.
// Pre-condition:
// We have a valid socket
```

```
//
// Post-condition:
// That socket is bound to the IP and port.
//
bool Socket::bind ( const int port )
{

  if ( ! is_valid() )
    {
      return false;
    }
  m_addr.sin_family = AF_INET;
  m_addr.sin_addr.s_addr = INADDR_ANY;
  m_addr.sin_port = htons ( port );
  int bind_return = ::bind ( m_sock,
            ( struct sockaddr * ) &m_addr,
            sizeof ( m_addr ) );
  if ( bind_return == -1 )
    {
      return false;
    }

  return true;
}
//
// Socket::listen()
// Create a new socket and associate it with our instance.
// Pre-condition:
// We have a valid socket
//
// Post-condition:
// We are listening on a valid socket.
//
bool Socket::listen() const
{
  if ( ! is_valid() )
    {
      return false;
    }

  int listen_return = ::listen ( m_sock, MAXCONNECTIONS );


  if ( listen_return == -1 )
    {
      return false;
    }

  return true;
}
//
// Socket::accept()
// Accpet a connection on this socket.
// Pre-condition:
// A valid and correct socket is passed to us for connection purposes
//
// Post-condition:
// We are accepting connections on that socket
//
bool Socket::accept ( Socket& new_socket ) const
{
  int addr_length = sizeof ( m_addr );
  new_socket.m_sock = ::accept ( m_sock, ( sockaddr * ) &m_addr, ( socklen_t * ) &
      addr_length );

```

```
  if ( new_socket.m_sock <= 0 )
    return false;
  else
    return true;
}
//
// Socket::send(const std::string)
// Send data out over our socket.
// Pre-condition:
// A valid and correct socket is passed to us for connection purposes
// Data is passed to us in the method parameter.
//
// Post-condition:
// The data gets sent over the socket.
//
bool Socket::send ( const std::string s ) const
{
  int status = ::send( m_sock, s.c_str(), s.size(), 0);
  if ( status == -1 )
    {
      return false;
    }
  else
    {
      return true;
    }
}
//
// Socket::recv(std::string&)
// Get data from our socket.
// Pre-condition:
// A valid and correct socket is passed to us for connection purposes
// We have a valid string to put data into
//
// Post-condition:
// The data gets read from the socket.
//
int Socket::recv ( std::string& s ) const
{
  char buf [ MAXRECV + 1 ];

  s = "";

  memset ( buf, 0, MAXRECV + 1 );

  int status = ::recv ( m_sock, buf, MAXRECV, 0 );

  if ( status == -1 )
    {
      std::cout << "status == -1   errno == " << errno << "  in Socket::recv\n";
      return 0;
    }
  else if ( status == 0 )
    {
      return 0;
    }
  else
    {
      s = buf;
      return status;
    }
}
//
// Socket::connect(const std::string, const int)
// Connect a socket to a host and port
```

```
// Pre-condition:
// A valid and correct socket is passed to us for connection purposes
// We have a valid host and port
//
// Post-condition:
// Our socket is connected to the host and port.
//
bool Socket::connect ( const std::string host, const int port )
{
  if ( ! is_valid() ) return false;

  m_addr.sin_family = AF_INET;
  m_addr.sin_port = htons ( port );

  int status = inet_pton ( AF_INET, host.c_str(), &m_addr.sin_addr );

  if ( errno == EAFNOSUPPORT ) return false;

  status = ::connect ( m_sock, ( sockaddr * ) &m_addr, sizeof ( m_addr ) );

  if ( status == 0 )
    return true;
  else
    return false;
}
//
// Socket::set_non_blocking(bool)
// The socket state is set to non-blocking.
// Pre-condition:
// A valid and correct socket is passed to us for connection purposes
//
// Post-condition:
// The state of the socket has been changed.
//
void Socket::set_non_blocking ( const bool b )
{
  int opts;
  opts = fcntl ( m_sock,
     F_GETFL );
  if ( opts < 0 )
    {
      return;
    }
  if ( b )
    opts = ( opts | O_NONBLOCK );
  else
    opts = ( opts & ~O_NONBLOCK );
  fcntl ( m_sock,
    F_SETFL,opts );
}
```

Case study: Sockets in C++: The Client Socket Class

```
class ClientSocket : private Socket {
 public:
  ClientSocket ( std::string host, int port );
  virtual ~ClientSocket(){};
  const ClientSocket& operator << ( const std::string& ) const;
  const ClientSocket& operator >> ( std::string& ) const;
};
```

And the implementation of our client socket class:

```
#include "ClientSocket.h"
#include "SocketException.h"
//
// ClientSocket::ClientSocket(std::string, int)
// Construct a new instance of the client side of our C++ socket interface.
// Pre-condition:
// Host and port passed from calling application.
//
// Post-condition:
// A new instance of our class is created on the heap.
//
ClientSocket::ClientSocket ( std::string host, int port )
{
  if ( ! Socket::create() )
    {
      throw SocketException ( "Could not create client socket." );
    }

  if ( ! Socket::connect ( host, port ) )
    {
      throw SocketException ( "Could not bind to port." );
    }
}
//
// ClientSocket::operator <<(const std::string &)
// Put a piece of data to the socket
// Pre-condition:
// String to be output is passed into method.
//
// Post-condition:
// The information is sent to the socket or an exception is thrown if the
// send fails.
//
const ClientSocket& ClientSocket::operator << ( const std::string& s ) const
{
  if ( ! Socket::send ( s ) )
    {
      throw SocketException ( "Could not write to socket." );
    }

  return *this;

}
//
// ClientSocket::operator >>(const std::string &)
// Get a piece of data from the socket.
// Pre-condition:
// String into which the data is to be dumped is passed into method.
//
// Post-condition:
// The information is read from the socket or an exception is thrown if the
// read failed.
//
const ClientSocket& ClientSocket::operator >> ( std::string& s ) const
{
  if ( ! Socket::recv ( s ) )
    {
      throw SocketException ( "Could not read from socket." );
    }

  return *this;
}
```

Case Study: Sockets in C++: A Simple Echo Server

```
#include "ClientSocket.h"
#include "SocketException.h"
#include <iostream>
#include <string>
using namespace std;
int main ( int argc, char *argv[] ) {
  try {
      ClientSocket client_socket ( "localhost", 30000 );
      string reply;
      try {
    client_socket << "Test message.";
    client_socket >> reply;
  }
      catch ( SocketException& ) {}
      cout << "We received this response from the server:\n\"" << reply << "\"\n";;
    }
  catch ( SocketException& e ) {}
      cout << "Exception was caught:" << e.description() << "\n";
    }
  return 0;
}
```

Case Study: Sockets in C++: A Simple Echo Server

```
#include "ServerSocket.h"
#include "SocketException.h"
#include <string>
#include <iostream>
using namespace std;
int main ( int argc, char * argv[] ) {
  cout << "running....\n";
  try {
      // Create the socket
      ServerSocket server ( 30000 );
      while ( true ) {
    ServerSocket new_sock;
    server.accept ( new_sock );
    try {
        while ( true ) {
      std::string data;
      new_sock >> data;
      new_sock << data;
    }
          }
    catch ( SocketException& ) {}
  }
  }
  catch ( SocketException& e ) {
      cout << "Exception was caught:" << e.description() << "\nExiting.\n";
    }
  return 0;
}
```