

# Interprocess and Network Communications

# Sockets

- Mechanism for exchanging data between processes/programs on the same computer or over a network
- Connection oriented – a connection must be established before data can be exchanged (e.g. TCP sockets)
  - Or
- Message oriented – no connection needs to be establish, each message is given an address for the destination (e.g. UDP sockets)
- Point – to – point: data is exchanged between two processes
  - Or
- Point – to – multipoint: data is broadcast from one process to one or more other processes
- Reliable: data will be received in order with error checking (e.g. TCP sockets).
  - Or
- Unreliable: an attempt will be made to send the data. It may arrive out of order, duplicates may arrive, or it may never arrive. No error checking (e.g. UDP sockets).

# TCP

- Transmission Control Protocol
- Connection-oriented protocol that operates on top of IP (internet protocol)

## Properties:

- Ordered data transfer - the destination host rearranges according to sequence number
- Retransmission of lost packets - any cumulative stream not acknowledged will be retransmitted
- Discarding duplicate packets
- Error-free data transfer
- Flow control - limits the rate a sender transfers data to guarantee reliable delivery. When the receiving host's buffer fills, the next acknowledgement contains a 0 in the window size, to stop transfer and allow the data in the buffer to be processed
- Congestion control - sliding window

# UDP

- User Datagram Protocol
- Connectionless protocol that operates on top of IP (internet protocol)

| TCP  | UDP   |
|--|---|
| <p><b>Reliable</b> – TCP manages message acknowledgment, retransmission and timeout. Many attempts to reliably deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.</p> | <p><b>Unreliable</b> – When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission and timeout.</p>   |
| <p><b>Ordered</b> – if two messages are sent over a connection in sequence, the first message will reach the receiving application first. When data segments arrive in the wrong order, TCP buffers the out-of-order data until all data can be properly re-ordered and delivered to the application.</p>                      | <p><b>Not ordered</b> – If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.</p>   |
| <p><b>Heavyweight</b> – TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.</p>  | <p><b>Lightweight</b> – There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.</p>  |
| <p><b>Streaming</b> – Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.</p>   | <p><b>Datagrams</b> – Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.</p> |

# TCP Socket Programming

| Server   | Client  |
|--|---|
| 1. Establish a listening socket and wait for connections from clients. |   |
|  | 2. Create a client socket and attempt to connect to server. |
| 3. Accept the client's connection attempt.                             |   |
| 4. Send and receive data.  | 4. Send and receive data.                                   |
| 5. Close the connection.   | 5. Close the connection.                                    |

# Exception Handling

- Exception handling is a methodology for handling run-time errors
- With exception handling your code **“try”**s to execute and you **“catch”** any errors that come along. When an error does occur you **“throw”** the error to the nearest **“catch”**er.
- The nearest catch doesn't have to be in the same function. It may be in a higher level function. The exception gets passed up the call chain until it gets to main().
- If the exception isn't caught anywhere, the program terminates.
- A throw statement throws a particular data type (i.e. class). Multiple catch statements can look for different data types.

# Exception Handling

```
int main()
{
    try
    {
        int value;
        cout << "Enter a positive integer:";
        cin >> value;
        if (value <= 0)
            throw value;

        // do something useful with a positive integer here
    }
    catch (int x)
    {
        cout << "I said a positive integer!!!!" << endl;
    }
    catch (...)
    {
        cout << "How did we end up here???" << endl;
    }
}
```

# Socket Example Code

- <http://linuxgazette.net/issue74/tougher.html>

# Server

```
int main ( int argc, int argv[] )
{
    std::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 )
    {
        std::cout << "Exception was caught:" << e.description() << "\nExiting.\n";
    }

    return 0;
}
```

# Client

```
int main ( int argc, int argv[] )
{
    try
    {

        ClientSocket client_socket ( "localhost", 30000 );

        std::string reply;

        try
        {
            client_socket << "Test message.";
            client_socket >> reply;
        }
        catch ( SocketException& ) {}

        std::cout << "We received this response from the server:\n\"" << reply << "\"\n";

    }
    catch ( SocketException& e )
    {
        std::cout << "Exception was caught:" << e.description() << "\n";
    }

    return 0;
}
```

# Server Constructor

```
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." );
    }
}
```

# Create Socket

```
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;
}
```

# Listen

```
bool Socket::listen() const
{
    if ( ! is_valid() )
    {
        return false;
    }

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

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

    return true;
}
```

# Accept

```
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;
}
```

# Bind

```
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;
}
```

# Client

```
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." );
    }
}

const ClientSocket& ClientSocket::operator << ( const std::string& s ) const
{
    if ( ! Socket::send ( s ) )
    {
        throw SocketException ( "Could not write to socket." );
    }

    return *this;
}

const ClientSocket& ClientSocket::operator >> ( std::string& s ) const
{
    if ( ! Socket::recv ( s ) )
    {
        throw SocketException ( "Could not read from socket." );
    }

    return *this;
}
```

# Connect

```
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;
}
```

# Send / Receive

```
bool Socket::send ( const std::string s ) const
{
    int status = ::send ( m_sock, s.c_str(), s.size(), MSG_NOSIGNAL );
    if ( status == -1 )
    {
        return false;
    }
    else
    {
        return true;
    }
}

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;
    }
}
```