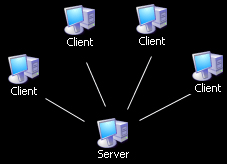
**Tutorial 1: UDP Communication**

http://web.archive.org/web/20140722013916/http:/rastertek.com/pic1001.gif

This network programming tutorial will cover how to implement UDP socket communication between a Linux server and multiple Windows clients.

Network communication is achieved by using sockets to read and send data packets between the server and client. The model we will be using is single server/multiple clients such as follows:



First the choice to use Linux instead of Windows as a server is one of speed and efficiency. Linux was designed from the ground up to be an efficient multithreading operating system which lends itself well to multithreading server based software. Windows was designed to run powerful single user applications as well as be far more user friendly which incurs incredible overhead and doesn't deliver the efficiency required by most server software. Memory handling by the two operating systems is also noticeably different the longer the server is online, and once again Linux has the more efficient model for memory management.

The second important point is that this tutorial will be using the UDP protocol for sending and reading data packets between the server and the client. UDP is the fastest common protocol used for sending and receiving packets. However to gain this speed it had to give up reliability that other protocols such as TCP have. This however is not a limiting factor provided we design the client/server application with this in mind. Assume that the server or client can miss a packet at anytime and design the software around this limitation and you will gain the best of both worlds.

As this is the first networking tutorial we will create a very simple application. The server will start and loop forever in a thread checking if any packets have come in. If it receives a packet it will read that packet and send a message back to the client if needed. It will be able to handle multiple clients connecting and will process requests to all of them at the same time.

The client part of the application will connect to the server and then send a latency request every 5 seconds. The server will respond back and the client will determine how long it takes to send and receive messages from the server and then display this value on the client's screen. When the client presses escape it will disconnect from the server and the server will also exit when it receives a message that the client has disconnected.

We will start the tutorial by looking at the server code first.

**SERVER**

On Linux I use the command line for coding. I use emacs as the editor and then compile from the command line. To compile in Linux we create a file called makefile and put the compiling instructions inside it. After that we type in make at the command line and it will compile the code into an executable binary file. Here is the makefile I used for this project:

Makefile

server: main.o networkclass.o systemclass.o

g++ -o server main.o networkclass.o systemclass.o -pthread

main.o: main.cpp

g++ -c main.cpp

networkclass.o: networkclass.cpp

g++ -c networkclass.cpp

systemclass.o: systemclass.cpp

g++ -c systemclass.cpp

main.cpp

The main() function is the program execution entry point. In the main.cpp file I create a system and network object and then loop calling the system object's Frame function until the network object goes offline. When the network goes offline then the network and system object are shut down and the program exits.

////////////////////////////////////////////////////////////////////////////////

// Filename: main.cpp

////////////////////////////////////////////////////////////////////////////////

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "networkclass.h"

#include "systemclass.h"

int main()

{

SystemClass\* System;

NetworkClass\* Network;

bool result;

// Create the system object.

System = new SystemClass;

if(!System)

{

return 0;

}

// Initialize the system object.

result = System->Initialize();

if(!result)

{

return 0;

}

// Create the network object.

Network = new NetworkClass;

if(!Network)

{

return 0;

}

// Initialize the network object.

result = Network->Initialize();

if(!result)

{

return 0;

}

while(Network->Online())

{

System->Frame();

}

// Release the network object.

if(Network)

{

Network->Shutdown();

delete Network;

Network = 0;

}

// Release the system object.

if(System)

{

System->Shutdown();

delete System;

System = 0;

}

return 0;

}

Systemclass.h

SystemClass is the parent class for the server application where we will do all the server processing. In this tutorial this class will be kept empty but it exists so that you can build on the tutorial yourself if you choose to do so.

////////////////////////////////////////////////////////////////////////////////

// Filename: systemclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SYSTEMCLASS\_H\_

#define \_SYSTEMCLASS\_H\_

////////////////////////////////////////////////////////////////////////////////

// Class name: SystemClass

////////////////////////////////////////////////////////////////////////////////

class SystemClass

{

public:

SystemClass();

SystemClass(const SystemClass&);

~SystemClass();

bool Initialize();

void Shutdown();

void Frame();

private:

};

#endif

Systemclass.cpp

Likewise the SystemClass code is also kept empty to keep the tutorial simple and focused, but it is here if you want to add functionality to this tutorial later.

////////////////////////////////////////////////////////////////////////////////

// Filename: systemkclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "systemclass.h"

SystemClass::SystemClass()

{

}

SystemClass::SystemClass(const SystemClass& other)

{

}

SystemClass::~SystemClass()

{

}

bool SystemClass::Initialize()

{

return true;

}

void SystemClass::Shutdown()

{

return;

}

void SystemClass::Frame()

{

return;

}

Networkmessages.h

The Networkmessages.h header file is used for both the client and the server. It contains the id numbers, message types, and message structures that will be used for communication. This tutorial only requires a couple different message types and all of them will use the same data structure called MSG\_GENERIC\_DATA. When you expand your own network program you can add new message types, ids, and new message structures to this file. Make sure to keep it the same on both the client and the server whenever you make updates.

////////////////////////////////////////////////////////////////////////////////

// Filename: networkmessages.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_NETWORKMESSAGES\_H\_

#define \_NETWORKMESSAGES\_H\_

/////////////////

// NETWORK IDS //

/////////////////

#define ID\_SERVER -2

#define ID\_UNKNOWN -1

///////////////////////////

// NETWORK MESSAGE TYPES //

///////////////////////////

#define MSG\_CONNECT 1000

#define MSG\_ID 1001

#define MSG\_PING 1002

#define MSG\_DISCONNECT 1003

////////////////////////////////

// NETWORK MESSAGE STRUCTURES //

////////////////////////////////

typedef struct

{

short type;

short toId;

short fromId;

}MSG\_GENERIC\_DATA;

#endif

Networkclass.h

The NetworkClass handles all the network related processing. All communication methods are encapsulated here and any further enhancement to the network code should be done inside this class only. Think of this class as the telephone object of the server program. It handles sending and receiving of data but it should not process anything. In this tutorial to simplify things this class will do a little bit of processing but I will also make mention of the proper way it should be done for future changes.

////////////////////////////////////////////////////////////////////////////////

// Filename: networkclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_NETWORKCLASS\_H\_

#define \_NETWORKCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const int MAX\_CLIENTS = 1000;

const unsigned short PORT\_NUMBER = 7531;

//////////////

// INCLUDES //

//////////////

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <string.h>

#include <sys/ioctl.h>

#include <pthread.h>

#include <iostream>

using namespace std;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "networkmessages.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: NetworkClass

////////////////////////////////////////////////////////////////////////////////

class NetworkClass

{

private:

The ClientType structure will be used to represent each client that connects to the server. For now we simply record if they are online or not as well as their address so that we can send them messages.

struct ClientType

{

bool online;

struct sockaddr\_in clientAddress;

};

public:

NetworkClass();

NetworkClass(const NetworkClass&);

~NetworkClass();

Initialize and Shutdown are used to bring the network online and offline. After Initialize is called it is listening for information and sending any information that is needed while the server is online. The other public functions declared here are for the server network listening thread which is passed a pointer to this class object.

bool Initialize();

void Shutdown();

bool Online();

int GetServerSocket();

void ReadNetworkMessage(char\*, struct sockaddr\_in);

private:

bool InitializeClientList();

void ShutdownClientList();

bool InitializeServerSocket();

void ShutdownServerSocket();

void HandleConnectMessage(struct sockaddr\_in);

short GetNextIdNumber();

void HandlePingMessage(short);

void HandleDisconnectMessage(short);

private:

ClientType\* m\_clientList;

bool m\_online;

int m\_socket;

};

The function defined at the end of this class is the thread function which listens for incoming network messages. Thread functions need to be declared outside of the class but I put it inside the NetworkClass since it is entirely related to everything that this class does. Also the thread will only be called by this class.

/////////////////////////

// FUNCTION PROTOTYPES //

/////////////////////////

void\* ServerListenFunction(void\*);

#endif

Networkclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: networkclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "networkclass.h"

The class constructor initializes the client list pointer to null to start with.

NetworkClass::NetworkClass()

{

m\_clientList = 0;

}

NetworkClass::NetworkClass(const NetworkClass& other)

{

}

NetworkClass::~NetworkClass()

{

}

The Initialize function first initializes the client list and then brings the listening server socket online.

bool NetworkClass::Initialize()

{

bool result;

// Initialize the client list.

result = InitializeClientList();

if(!result)

{

return false;

}

// Initialize and start the server socket to listen and process incoming connections.

result = InitializeServerSocket();

if(!result)

{

return false;

}

return true;

}

Shutdown is called after the network has been set offline and is no longer listening for incoming messages. At this point it first releases the client list and then shuts down the server socket.

void NetworkClass::Shutdown()

{

// Release the client list.

ShutdownClientList();

// Close and release the server socket.

ShutdownServerSocket();

return;

}

The InitializeClientList function creates the client list and then sets all the clients to currently be offline. It uses the MAX\_CLIENTS global that is defined in the header file to limit the size of the client list and how many people will be allowed to connect to the server.

bool NetworkClass::InitializeClientList()

{

int i;

// Create the client list.

m\_clientList = new ClientType[MAX\_CLIENTS];

if(!m\_clientList)

{

return false;

}

// Loop through all the clients and initialize them.

for(i=0; i<MAX\_CLIENTS; i++)

{

m\_clientList[i].online = false;

}

return true;

}

ShutdownClientList is used to release all the clients. In this tutorial I don't send all the clients messages that they have been disconnected but this is something you can expand on yourself.

void NetworkClass::ShutdownClientList()

{

int i;

// Let each client know it has been disconnected from the server.

for(i=0; i<MAX\_CLIENTS; i++)

{

}

// Then delete the client list.

if(m\_clientList)

{

delete [] m\_clientList;

m\_clientList = 0;

}

return;

}

InitializeServerSocket is the main function for bringing the network online. It creates a socket for listening for network communication and then calls the thread to loop forever reading for network I/O until the network goes offline. Notice that after creating the socket I set it to non-blocking I/O, this ensures that when the socket is used to read data that it won't wait forever. If it finds no data it will return and not block the program from continuing execution. Because of this I/O method we need to continuously loop and keep reading to check if any new data has come in recently.

bool NetworkClass::InitializeServerSocket()

{

struct sockaddr\_in serverAddress;

int error;

unsigned long setting;

pthread\_t serverThreadId;

// Set the server to be online.

m\_online = true;

// Create a UDP socket.

m\_socket = socket(AF\_INET, SOCK\_DGRAM, 0);

if(m\_socket == -1)

{

cout << "Error: Could not create UDP socket." << endl;

return false;

}

// Fill in the address information for binding the socket to and have the kernel set the IP address.

memset((char\*)&serverAddress, 0, sizeof(serverAddress));

serverAddress.sin\_family = AF\_INET;

serverAddress.sin\_port = htons(PORT\_NUMBER);

serverAddress.sin\_addr.s\_addr = htonl(INADDR\_ANY);

// Bind the socket to the address.

error = bind(m\_socket, (struct sockaddr\*)&serverAddress, sizeof(serverAddress));

if(error == -1)

{

cout << "Error: Could not bind the socket." << endl;

return false;

}

// Set the socket to non-blocking I/O.

setting = 1;

error = ioctl(m\_socket, FIONBIO, &setting);

if(error == -1)

{

cout << "Error: Could not set socket to non-blocking I/O." << endl;

return false;

}

// Create a thread to listen for and accept incoming connections from clients.

error = pthread\_create(&serverThreadId, NULL, ServerListenFunction, (void\*)this);

if(error != 0)

{

cout << "Error: Could not create thread." << endl;

return false;

}

return true;

}

The ShutdownServerSocket function is called after the network is no longer online. It will close the socket that was used for network communication.

void NetworkClass::ShutdownServerSocket()

{

int error;

// Close the server socket.

error = close(m\_socket);

if(error != 0)

{

cout << "Error: Could not close socket correctly." << endl;

}

return;

}

Online is a public function that is used to query whether the network is online or not.

bool NetworkClass::Online()

{

return m\_online;

}

GetServerSocket is another public function that will be used by the network reading thread to get access to the private socket inside this class.

int NetworkClass::GetServerSocket()

{

return m\_socket;

}

ServerListenFunction is the thread function that will be called by this class to do the network reading. It is passed a pointer to this class as input so that it can communicate back to this class. The functionality is pretty simple, it will loop while the network is online and continuously read for network data on the server socket. If it finds any network data then it sends that message back into the class by calling the ReadNetworkMessage function. Note that this program is using multithreading when this function is in use, and since it loops non-stop it will use a good amount of cpu to ensure we don't miss reading any messages.

void\* ServerListenFunction(void\* ptr)

{

NetworkClass\* networkClassPtr;

int bytesRead;

char recvBuffer[4096];

struct sockaddr\_in clientAddress;

unsigned int clientLength;

// Get a pointer to the calling object.

networkClassPtr = (NetworkClass\*)ptr;

// Set the size of the address.

clientLength = sizeof(clientAddress);

while(networkClassPtr->Online())

{

// Check if there is a message from a client.

bytesRead = recvfrom(networkClassPtr->GetServerSocket(), recvBuffer, 4096, 0, (struct sockaddr\*)&clientAddress, &clientLength);

if(bytesRead > 0)

{

networkClassPtr->ReadNetworkMessage(recvBuffer, clientAddress);

}

}

return 0;

}

ReadNetworkMessage is called when the I/O reading thread receives network data. The network data is passed back into the class through this function. Ideally the function should then put that data on a buffer to be processed later, but for this tutorial I have put in the three functions for processing the different types of messages. Note that this will block the thread from reading new network data until the processing is complete. If you expand on this tutorial make sure to change this function accordingly.

void NetworkClass::ReadNetworkMessage(char\* recvBuffer, struct sockaddr\_in clientAddress)

{

char\* ipAddress;

MSG\_GENERIC\_DATA\* message;

// Store the IP address of the person who sent a message.

ipAddress = inet\_ntoa(clientAddress.sin\_addr);

// Coerce the message into a generic format to read the type of message.

message = (MSG\_GENERIC\_DATA\*)recvBuffer;

switch(message->type)

{

case MSG\_CONNECT:

{

cout << ipAddress << " sent a connect message." << endl;

HandleConnectMessage(clientAddress);

break;

}

case MSG\_PING:

{

HandlePingMessage(message->fromId);

break;

}

case MSG\_DISCONNECT:

{

HandleDisconnectMessage(message->fromId);

break;

}

default:

{

break;

}

}

return;

}

HandleConnectMessage is called whenever we receive a connect message from a new client. We get a new ID number for that client and store their address so we can send them messages whenever we need to. After that we form a message with their ID number that they need to use when communicating with the server. The message is then sent to the client.

void NetworkClass::HandleConnectMessage(struct sockaddr\_in clientAddress)

{

short newId;

MSG\_GENERIC\_DATA message;

int bytesSent;

// Get the next free ID number and assign it to this client.

newId = GetNextIdNumber();

if(newId != -1)

{

// Store the client address information so that we can send them messages when we need to.

m\_clientList[newId].clientAddress = clientAddress;

}

message.type = MSG\_ID;

message.toId = newId;

message.fromId = ID\_SERVER;

bytesSent = sendto(m\_socket, (char\*)&message, sizeof(MSG\_GENERIC\_DATA), 0, (struct sockaddr\*)&clientAddress, sizeof(clientAddress));

if(bytesSent != sizeof(MSG\_GENERIC\_DATA))

{

cout << "Error: Could not send ID message." << endl;

}

else

{

cout << "Sent ID number " << newId << " to client." << endl;

}

return;

}

GetNextIdNumber searches the list of clients and finds a free ID number. It also sets that ID number to online. If it can't find an ID number then -1 is returned indicating the server is full and can't accept new clients.

short NetworkClass::GetNextIdNumber()

{

short idNumber;

bool done;

int i;

// Initialize the loop variables.

idNumber = -1;

done = false;

i=0;

// Loop through all the clients and assign an ID that is not online.

while(!done)

{

if(m\_clientList[i].online == false)

{

idNumber = i;

m\_clientList[idNumber].online = true;

done = true;

}

else

{

i++;

}

// Check if we have exceeded the maximum number of clients.

if(i == MAX\_CLIENTS)

{

done = true;

}

}

return idNumber;

}

HandlePingMessage is called whenever the server receives a network latency request from a client. We form a ping message and send it back to the client which they will then use to calculate their speed for communicating with the server.

void NetworkClass::HandlePingMessage(short fromId)

{

MSG\_GENERIC\_DATA message;

int bytesSent;

message.type = MSG\_PING;

message.toId = fromId;

message.fromId = ID\_SERVER;

bytesSent = sendto(m\_socket, (char\*)&message, sizeof(MSG\_GENERIC\_DATA), 0, (struct sockaddr\*)&m\_clientList[fromId].clientAddress,

sizeof(m\_clientList[fromId].clientAddress));

if(bytesSent != sizeof(MSG\_GENERIC\_DATA))

{

cout << "Error: Could not send ping message." << endl;

}

else

{

cout << "Sent ping message to client " << fromId << "." << endl;

}

return;

}

HandleDisconnectMessage is called whenever a client sends a messages that they have disconnected from the server. In this tutorial we actually have it also shut down the server at the same time. If you are going to expand this network code then you would want to remove setting the server offline and change it to release the client ID and continue on instead.

void NetworkClass::HandleDisconnectMessage(short fromId)

{

cout << "Disconnect meesage from " << fromId << ", shutting server down also." << endl;

// Set the server to be offline.

m\_online = false;

return;

}

**CLIENT**

The client side code is written for the Windows operating system. We will start by looking at the NetworkClass equivalent for the Windows client.

Networkclass.h

///////////////////////////////////////////////////////////////////////////////

// Filename: networkclass.h

///////////////////////////////////////////////////////////////////////////////

#ifndef \_NETWORKCLASS\_H\_

#define \_NETWORKCLASS\_H\_

The first global is for turning the network on or off. This allows you to debug the program without the network being on and using if statements before all your network code. The second and third globals are the IP address and port number of the server that the client will be connecting to. Note that you will probably have to open the port on the Linux server otherwise the firewall will block incoming connections to it. Change the IP address to your own and change the port to whatever you like (and above 1024 preferably).

/////////////

// GLOBALS //

/////////////

const bool NETWORK\_ENABLED = true;

static char SERVER\_ADDRESS[] = "192.168.1.102";

const unsigned short SERVER\_PORT = 7531;

Windows uses what is called the WinSock library for networking. The following library and header file is required for the code to compile. I also include the winmm.lib for the windows timer that will be used in determining the speed of data transfer.

/////////////

// LINKING //

/////////////

#pragma comment(lib, "ws2\_32.lib")

#pragma comment(lib, "winmm.lib")

//////////////

// INCLUDES //

//////////////

#include <winsock2.h>

#include <mmsystem.h>

The networkmessages.h is the same file that was included on the server side code. This file has to be kept consistent between both client and server always.

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "networkmessages.h"

The NetworkClass definition is similar to the server side. The code as well will be fairly similar except for the differences in the windows socket library, for example most things on Linux return -1 when they fail and the windows socket would instead return SOCKET\_ERROR and then you would have to query another object for the exact error. Other than those minor differences it works the exact same.

///////////////////////////////////////////////////////////////////////////////

// Class name: NetworkClass

///////////////////////////////////////////////////////////////////////////////

class NetworkClass

{

public:

NetworkClass();

NetworkClass(const NetworkClass&);

~NetworkClass();

bool Initialize();

void Shutdown();

void Frame();

int GetLantency();

bool ConnectToServer(char\*, unsigned short);

void SetThreadActive();

void SetThreadInactive();

bool Online();

SOCKET GetClientSocket();

void ReadNetworkMessage(char\*);

private:

bool InitializeWinSock();

void ShutdownWinsock();

void ProcessLatency();

void SendPing();

void HandlePingMessage();

void SendDisconnectMessage();

private:

SOCKET m\_clientSocket;

int m\_addressLength;

struct sockaddr\_in m\_serverAddress;

short m\_idNumber;

bool m\_online;

bool m\_threadActive;

int m\_latency;

unsigned long m\_pingTime;

};

Just like the server code we have a thread function for the windows client which will handle reading data sent from the server running at all times on a separate thread.

/////////////////////////

// FUNCTION PROTOTYPES //

/////////////////////////

void NetworkReadFunction(void\*);

#endif

Networkclass.cpp

///////////////////////////////////////////////////////////////////////////////

// Filename: networkclass.cpp

///////////////////////////////////////////////////////////////////////////////

#include "networkclass.h"

NetworkClass::NetworkClass()

{

m\_online = false;

}

NetworkClass::NetworkClass(const NetworkClass& other)

{

}

NetworkClass::~NetworkClass()

{

}

Initialize will call the InitializeWinSock function to setup sockets and prepare us for connecting to the server.

bool NetworkClass::Initialize()

{

bool result;

// Initialize winsock for using window's sockets.

result = InitializeWinSock();

if(!result)

{

return false;

}

return true;

}

Shutdown does a couple things in a very specific order. First it sends a message to the server that we are disconnecting from them. After it sends that message it then sets the m\_online flag to false. Since the thread function keeps checking that flag it will exit on its own once it sees it is set to false. When the thread exits it lets this class know by setting m\_threadActive to false through one of the public functions this class offers that thread for use. We loop until the thread does exit and then we close the socket and shutdown the windows socket functionality.

void NetworkClass::Shutdown()

{

// If it never went online then no need to shut it down.

if(!m\_online)

{

return;

}

// Send a message to the server letting it know this client is disconnecting.

SendDisconnectMessage();

// Set the client to be offline.

m\_online = false;

// Wait for the network I/O thread to complete.

while(m\_threadActive)

{

}

// Close the socket.

closesocket(m\_clientSocket);

// Shutdown winsock.

ShutdownWinsock();

return;

}

As we are keeping this tutorial simple the Frame function will only be processing the network latency which will also be displayed on the screen.

void NetworkClass::Frame()

{

ProcessLatency();

return;

}

The InitializeWinSock function sets up the ability for the program to use windows sockets. We request version 2.0 of the socket library and request the use of the UDP and TCP protocols. We won't use the TCP protocol in this tutorial but it makes this function ready for future tutorials.

bool NetworkClass::InitializeWinSock()

{

unsigned short version;

WSADATA wsaData;

int error;

unsigned long bufferSize;

WSAPROTOCOL\_INFOW\* protocolBuffer;

int protocols[2];

// Create a 2.0 macro to check versions.

version = MAKEWORD(2, 0);

// Get the data to see if it handles the version we want.

error = WSAStartup(version, &wsaData);

if(error != 0)

{

return false;

}

// Check to see if the winsock dll is version 2.0.

if((LOBYTE(wsaData.wVersion) != 2) || (HIBYTE(wsaData.wVersion) != 0))

{

return false;

}

// Request the buffer size needed for holding the protocols available.

WSAEnumProtocols(NULL, NULL, &bufferSize);

// Create a buffer for the protocol information structs.

protocolBuffer = new WSAPROTOCOL\_INFOW[bufferSize];

if(!protocolBuffer)

{

return false;

}

// Create the list of protocols we are looking for which are TCP and UDP.

protocols[0] = IPPROTO\_TCP;

protocols[1] = IPPROTO\_UDP;

// Retrieve information about available transport protocols, if no socket error then the protocols from the list will work.

error = WSAEnumProtocols(protocols, protocolBuffer, &bufferSize);

if(error == SOCKET\_ERROR)

{

return false;

}

// Release the protocol buffer.

delete [] protocolBuffer;

protocolBuffer = 0;

return true;

}

ShutdownWinsock just calls WSACleanup which cleans up and releases the windows socket api.

void NetworkClass::ShutdownWinsock()

{

WSACleanup();

return;

}

ConnectToServer is called when the client wants to connect to the server. It will create a UDP socket and then use that to connect to the input IP address and port number. Once it sends a connect message to the server it then waits for a reply with the ID number that it will be assigned. After it receives the ID number it then starts the thread which will listen for messages from the server until the client disconnects.

bool NetworkClass::ConnectToServer(char\* ipAddress, unsigned short portNumber)

{

unsigned long setting, inetAddress, startTime, threadId;

MSG\_GENERIC\_DATA connectMessage, \*message;

int error, bytesSent, bytesRead;

bool done, gotId;

char recvBuffer[4096];

HANDLE threadHandle;

// Create a UDP socket.

m\_clientSocket = socket(AF\_INET, SOCK\_DGRAM, 0);

if(m\_clientSocket == INVALID\_SOCKET)

{

return false;

}

// Set the client socket to non-blocking I/O.

setting = 1;

error = ioctlsocket(m\_clientSocket, FIONBIO, &setting);

if(error == SOCKET\_ERROR)

{

return false;

}

// Save the size of the server address structure.

m\_addressLength = sizeof(m\_serverAddress);

// Setup the address of the server we are sending data to.

inetAddress = inet\_addr(ipAddress);

memset((char\*)&m\_serverAddress, 0, m\_addressLength);

m\_serverAddress.sin\_family = AF\_INET;

m\_serverAddress.sin\_port = htons(portNumber);

m\_serverAddress.sin\_addr.s\_addr = inetAddress;

// Setup a connect message to send to the server.

connectMessage.type = MSG\_CONNECT;

connectMessage.toId = ID\_SERVER;

connectMessage.fromId = ID\_UNKNOWN;

// Send the connect message to the server.

bytesSent = sendto(m\_clientSocket, (char\*)&connectMessage, sizeof(MSG\_GENERIC\_DATA), 0, (struct sockaddr\*)&m\_serverAddress, m\_addressLength);

if(bytesSent < 0)

{

return false;

}

// Record the time when the connect packet was sent.

startTime = timeGetTime();

// Set the boolean loop values.

done = false;

gotId = false;

while(!done)

{

// Check for a reply message from the server.

bytesRead = recvfrom(m\_clientSocket, recvBuffer, 4096, 0, (struct sockaddr\*)&m\_serverAddress, &m\_addressLength);

if(bytesRead > 0)

{

done = true;

gotId = true;

}

// Check to see if this loop has been running for longer than 2 seconds.

if(timeGetTime() > (startTime + 2000))

{

done = true;

gotId = false;

}

}

// If it didn't get an ID in 2 seconds then the server was not up.

if(!gotId)

{

return false;

}

// Coerce the message into a generic message type.

message = (MSG\_GENERIC\_DATA\*)recvBuffer;

// Ensure it was an ID message.

if(message->type != MSG\_ID)

{

return false;

}

// Ensure the ID number was not a server full message.

if(message->toId == -1)

{

return false;

}

// Store the ID number for this client for all future communication with the server.

m\_idNumber = message->toId;

// Set the client to be online now.

m\_online = true;

// Initialize the thread activity variable.

m\_threadActive = false;

// Create a thread to listen for network I/O from the server.

threadHandle = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)NetworkReadFunction, (void\*)this, 0, &threadId);

if(threadHandle == NULL)

{

return false;

}

// Initialize the network latency variables.

m\_latency = 0;

m\_pingTime = timeGetTime();

return true;

}

GetLantency gives calling functions the network latency that is calculated from the ping messages. The user interface will call this function so that it can display the speed on network screen.

int NetworkClass::GetLantency()

{

return m\_latency;

}

The SetThreadActive and SetThreadInactive public functions are used by the network I/O thread to notify this class when the thread is active or not active. The class can then use this information to know if the thread has exited or not.

void NetworkClass::SetThreadActive()

{

m\_threadActive = true;

return;

}

void NetworkClass::SetThreadInactive()

{

m\_threadActive = false;

return;

}

The Online public function is used to query whether this class object is online or not. For this tutorial the I/O thread will use this function to check if the network has gone offline or not. If it sees the network is offline then it shuts the thread down also.

bool NetworkClass::Online()

{

return m\_online;

}

GetClientSocket is another public function that will be used by the I/O thread to get access to the private client socket.

SOCKET NetworkClass::GetClientSocket()

{

return m\_clientSocket;

}

The NetworkReadFunction is the network I/O reading thread. It loops forever reading for data from the server until the network goes offline. When it does get data it sends it back to the class object so that it can be processed. Notice that both the client and server read a maximum of 4096 bytes, the reason for this is that it is the maximum UDP packet size.

void NetworkReadFunction(void\* ptr)

{

NetworkClass\* networkClassPtr;

struct sockaddr\_in serverAddress;

int addressLength;

int bytesRead;

char recvBuffer[4096];

// Get a pointer to the calling object.

networkClassPtr = (NetworkClass\*)ptr;

// Notify parent object that this thread is now active.

networkClassPtr->SetThreadActive();

// Set the size of the address.

addressLength = sizeof(serverAddress);

// Loop and read network messages while the client is online.

while(networkClassPtr->Online())

{

// Check if there is a message from the server.

bytesRead = recvfrom(networkClassPtr->GetClientSocket(), recvBuffer, 4096, 0, (struct sockaddr\*)&serverAddress, &addressLength);

if(bytesRead > 0)

{

networkClassPtr->ReadNetworkMessage(recvBuffer);

}

}

// Notify parent object that this thread is now inactive.

networkClassPtr->SetThreadInactive();

return;

}

ReadNetworkMessage is called whenever the network I/O thread receives data from the server. Ideally this function should put the message on a buffer and then let the Frame function process the messages so we don't block the thread from reading data (since the thread is blocked until this function returns). However to keep things simple in this tutorial we will process the ping message since we aren't expecting anything else to come in. But do remember receiving data and not missing packets is paramount for any network application.

void NetworkClass::ReadNetworkMessage(char\* recvBuffer)

{

MSG\_GENERIC\_DATA\* message;

// Coerce the message into a generic format to read the type of message.

message = (MSG\_GENERIC\_DATA\*)recvBuffer;

switch(message->type)

{

case MSG\_PING:

{

HandlePingMessage();

break;

}

default:

{

break;

}

}

return;

}

ProcessLatency is called by the Frame function. The function will check if 5 seconds have passed since the last time it sent a ping message. If so it will send another ping message which will update the latency speed of the client again.

void NetworkClass::ProcessLatency()

{

// If 5 seconds is up then send a ping message to the server.

if(timeGetTime() >= (m\_pingTime + 5000))

{

m\_pingTime = timeGetTime();

SendPing();

}

return;

}

SendPing sends ping messages to the server. It sets up the message structure and then uses the client socket to send it to the server address. Also note I haven't set up any logging of errors if packets don't get sent, I have left that open for how you would like to deal with it.

void NetworkClass::SendPing()

{

MSG\_GENERIC\_DATA message;

int bytesSent;

// Create the ping message.

message.type = MSG\_PING;

message.toId = ID\_SERVER;

message.fromId = m\_idNumber;

bytesSent = sendto(m\_clientSocket, (char\*)&message, sizeof(MSG\_GENERIC\_DATA), 0, (struct sockaddr\*)&m\_serverAddress, m\_addressLength);

if(bytesSent != sizeof(MSG\_GENERIC\_DATA))

{

}

return;

}

Whenever we receive a return ping message from the server we call HandlePingMessage to process the time it took for the round trip. This then updates the network latency until the next time we send a ping message.

void NetworkClass::HandlePingMessage()

{

m\_latency = timeGetTime() - m\_pingTime;

return;

}

When the client wants to disconnect from the server it will call SendDisconnectMessage to send a disconnect message to the server letting it know it will no longer be communicating with it.

void NetworkClass::SendDisconnectMessage()

{

MSG\_GENERIC\_DATA message;

int bytesSent;

// Create the ping message.

message.type = MSG\_DISCONNECT;

message.toId = ID\_SERVER;

message.fromId = m\_idNumber;

bytesSent = sendto(m\_clientSocket, (char\*)&message, sizeof(MSG\_GENERIC\_DATA), 0, (struct sockaddr\*)&m\_serverAddress, m\_addressLength);

if(bytesSent != sizeof(MSG\_GENERIC\_DATA))

{

}

return;

}

Systemclass.h

The SystemClass has been changed to incorporate the new NetworkClass.

////////////////////////////////////////////////////////////////////////////////

// Filename: systemclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SYSTEMCLASS\_H\_

#define \_SYSTEMCLASS\_H\_

///////////////////////////////

// PRE-PROCESSING DIRECTIVES //

///////////////////////////////

#define WIN32\_LEAN\_AND\_MEAN

//////////////

// INCLUDES //

//////////////

#include <windows.h>

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "graphicsclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "networkclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: SystemClass

////////////////////////////////////////////////////////////////////////////////

class SystemClass

{

public:

SystemClass();

SystemClass(const SystemClass&);

~SystemClass();

bool Initialize();

void Shutdown();

void Run();

LRESULT CALLBACK MessageHandler(HWND, UINT, WPARAM, LPARAM);

private:

bool Frame();

void InitializeWindows(int&, int&);

void ShutdownWindows();

private:

LPCWSTR m\_applicationName;

HINSTANCE m\_hinstance;

HWND m\_hwnd;

InputClass\* m\_Input;

GraphicsClass\* m\_Graphics;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

NetworkClass\* m\_Network;

};

/////////////////////////

// FUNCTION PROTOTYPES //

/////////////////////////

static LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

/////////////

// GLOBALS //

/////////////

static SystemClass\* ApplicationHandle = 0;

#endif

Systemclass.cpp

I will just cover the changes in the SystemClass which were used to incorporate the NetworkClass into the frame work.

SystemClass::SystemClass()

{

m\_Input = 0;

m\_Graphics = 0;

m\_Fps = 0;

m\_Cpu = 0;

The class constructor now initializes the network object to null.

m\_Network = 0;

}

bool SystemClass::Initialize()

{

int screenWidth, screenHeight;

bool result;

// Initialize the width and height of the screen to zero before sending the variables into the function.

screenWidth = 0;

screenHeight = 0;

// Initialize the windows api.

InitializeWindows(screenWidth, screenHeight);

// Create the input object. This object will be used to handle reading the keyboard input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

m\_Input->Initialize();

// Create the graphics object. This object will handle rendering all the graphics for this application.

m\_Graphics = new GraphicsClass;

if(!m\_Graphics)

{

return false;

}

// Initialize the graphics object.

result = m\_Graphics->Initialize(screenWidth, screenHeight, m\_hwnd);

if(!result)

{

return false;

}

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

During the initialization of the SystemClass object we now start the network if it is enabled (which is defined in the NetworkClass header file). If it is enabled we initialize the Windows socket library and then attempt to connect to the server. The server address and port number are also defined in the NetworkClass header file.

// If the network is enabled then create and initialize the network object.

if(NETWORK\_ENABLED)

{

m\_Network = new NetworkClass;

if(!m\_Network)

{

return false;

}

result = m\_Network->Initialize();

if(!result)

{

MessageBox(m\_hwnd, L"Could not initialize the network object.", L"Error", MB\_OK);

return false;

}

result = m\_Network->ConnectToServer(SERVER\_ADDRESS, SERVER\_PORT);

if(!result)

{

MessageBox(m\_hwnd, L"Could not connect to server.", L"Error", MB\_OK);

return false;

}

}

return true;

}

void SystemClass::Shutdown()

{

If the network was enabled then we shut down the network object during the SystemClass shut down.

// If the network was enabled then release the network object.

if(NETWORK\_ENABLED)

{

if(m\_Network)

{

m\_Network->Shutdown();

delete m\_Network;

m\_Network = 0;

}

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the graphics object.

if(m\_Graphics)

{

m\_Graphics->Shutdown();

delete m\_Graphics;

m\_Graphics = 0;

}

// Release the input object.

if(m\_Input)

{

delete m\_Input;

m\_Input = 0;

}

// Shutdown the window.

ShutdownWindows();

return;

}

bool SystemClass::Frame()

{

bool result;

// Update the system stats.

m\_Fps->Frame();

m\_Cpu->Frame();

If the network is enabled then we call the NetworkClass::Frame function during each frame of the SystemClass to do all network related processing.

// If the network is enabled then do frame processing for it.

if(NETWORK\_ENABLED)

{

m\_Network->Frame();

}

// Set the system stats

result = m\_Graphics->SetFps(m\_Fps->GetFps());

if(!result)

{

return false;

}

result = m\_Graphics->SetCpu(m\_Cpu->GetCpuPercentage());

if(!result)

{

return false;

}

If the network is enabled then we also update the network latency in the user interface.

if(NETWORK\_ENABLED)

{

result = m\_Graphics->SetLatency(m\_Network->GetLantency());

if(!result)

{

return false;

}

}

// Do the frame processing for the graphics object.

m\_Graphics->Frame();

// Finally render the graphics to the screen.

m\_Graphics->Render();

return true;

}

Summary

We have now seen how to do a simple communication setup between multiple clients and a single server using UDP sockets.

To Do Exercises

1. Recompile the code and run the program. Let it run for 10 seconds or so and quit. Check the server output to see the detailed connection information and packets that were received.

2. Change the server and client code to be buffered when receiving messages in the network I/O threads.

3. Create some of your own message types such as when a key is pressed send that data to the server and display the key press on the server output screen.

Source Code

Visual Studio 2008 Project/Linux Code: [nwtut01.zip](http://web.archive.org/web/20140722013916/http:/rastertek.com/nwtut01.zip)

Source Only: [nwsrc01.zip](http://web.archive.org/web/20140722013916/http:/rastertek.com/nwsrc01.zip)