// Set up the server that will accept incoming connections

CommunicationsServer server =

new CommunicationsServer(IPAddress.Any, PortNum, new ServerMessageHandler());

server.BeginListening();

...

// Create an outgoing connection, used to connect with the server

OutgoingConnection connectionToServer =

new OutgoingConnection(connectionToServer\_MessageReceived);

// Establish an outgoing connection to the server

bool success = connectionToServer.Connect("localhost", PortNum, true);

// Send some kind of initial hello message

HelloMessage message = new HelloMessage("Hello from client number " + n.ToString());

connectionToServer.IssueCommand(message, 1);

...

// Shut down the server

server.ShutdownServer();

// Show some transmission info

Console.WriteLine("Server sent " + server.BytesSent.ToString() + ", " +

"received " + server.BytesReceived.ToString() + " bytes.");

The Send/Receive Buffer

/// <span class="code-SummaryComment"><summary></span>

/// Retrieves data waiting to be sent, and returns the data in a byte array ready

/// for transmission. The number of bytes is output via output parameter packetLength.

/// <span class="code-SummaryComment"></summary> </span>

internal byte[] ConstructOutgoingPacketFromByteChunks(out int packetLength)

{

byte[] outgoingPacket;

byte[] nextPacket;

int outgoingPacketLength = 0;

// Dequeue a waiting byte chunk

if (\_outgoingByteChunks.Dequeue(out outgoingPacket))

{

outgoingPacketLength = outgoingPacket.Length;

if (outgoingPacketLength < JoinPacketsIfSizeBelow)

{

// If the length of the byte chunk is small, then combine multiple

// byte chunks that may be waiting

using (MemoryStream ms = new MemoryStream(JoinPacketsIfSizeBelow))

{

while (outgoingPacketLength < JoinPacketsIfSizeBelow &&

\_outgoingByteChunks.Dequeue(out nextPacket))

{

ms.SetLength(0);

int nextPacketLength = nextPacket.Length;

// Append packet1 bytes with the bytes of packet2

ms.Write(outgoingPacket, 0, outgoingPacketLength);

ms.Write(nextPacket, 0, nextPacketLength);

outgoingPacket = ms.ToArray();

outgoingPacketLength += nextPacketLength;

}

}

}

}

packetLength = outgoingPacketLength;

return outgoingPacket;

}

## Messages and Fragmentation

/// <span class="code-SummaryComment"><summary></span>

/// Creates a byte array for an object (with appropriate header bytes),

/// allowing a message to be sent across the wire.

/// <span class="code-SummaryComment"></summary></span>

internal byte[] CreateForSend(object o, int transmissionType)

{

byte[] commandBytes;

using (MemoryStream ms = new MemoryStream())

{

// Leave the first 8 bytes empty for now

ms.Seek(8, SeekOrigin.Begin);

// Serialize the object to the memory stream starting at position 8

o.SerializeToCompressedBinaryMemoryStream(ms);

// Determine the byte length of the serialized object

int bytes = (int)ms.Length - 8;

// Write the header to the beginning of the memory stream

ms.WriteHeader(bytes, transmissionType);

commandBytes = ms.ToArray();

}

return commandBytes;

}

## Receiving Message Parts: MessageBuffer

public void AppendNewIncomingBytePackets(ThreadSafeQueue<byte[]> incomingByteChunks)

{

byte[] nextBytes = null;

if (\_incomingMemoryStream == null)

Interlocked.CompareExchange<MemoryStream>(ref \_incomingMemoryStream,

new MemoryStream(), null);

// Go to the end of the incoming memory stream

// and append any bytes that have recently come in

\_incomingMemoryStream.Seek(0, SeekOrigin.End);

while (incomingByteChunks.Dequeue(out nextBytes))

\_incomingMemoryStream.Write(nextBytes, 0, nextBytes.Length);

}

public bool ReadHeader(out bool error)

{

// To read a header, we must have

// at least PacketLengthPrefixSize bytes available

if (\_bytesAvailable - (int)\_binaryReader.BaseStream.Position <=

ConnectionBase.PacketLengthPrefixSize)

return false;

// Read the header - bytes expected and transmission type

\_bytesExpectedTemp = \_binaryReader.ReadInt32();

\_transmissionTypeTemp = \_binaryReader.ReadInt32();

...

}

public bool ReadMessage(out object o, out int transmissionType)

{

o = null;

transmissionType = 0;

if ((int)\_binaryReader.BaseStream.Position +

\_bytesExpectedTemp > \_bytesAvailable)

{

// The message is not complete (i.e. data may

// be coming in on the next incoming message).

// Rewind the stream back to the beginning of the message header

if (\_binaryReader.BaseStream.Position != 0)

\_binaryReader.BaseStream.Seek(-ConnectionBase.PacketLengthPrefixSize,

SeekOrigin.Current);

return false;

}

// If we're here, we have a full message.

// Read the message bytes and deserialize the message

byte[] messageBytes = \_binaryReader.ReadBytes(\_bytesExpectedTemp);

try

{

o = messageBytes.DeserializeFromCompressedBinary();

transmissionType = \_transmissionTypeTemp;

}

catch (Exception ex)

{

o = null;

// let this fall through and return true.

// The next message might be good.

}

return true;

}

/// <span class="code-SummaryComment"><summary></span>

/// Checks to determine if a full message has been received. If so, raises the

/// MessageReceived event.

/// <span class="code-SummaryComment"></summary> </span>

internal bool GetNextFullMessage()

{

bool fullMessageReceived = false;

bool error = false;

// No new data received, so no need to check for a new message

if (\_incomingByteChunks.Count <= 0) return false;

// Append the new data received to the incoming message buffer

\_incomingMessageBuffer.AppendNewIncomingBytePackets(\_incomingByteChunks);

// Records the number of bytes we have available

// and sets up the message buffer for reading

\_incomingMessageBuffer.RewindAndCreateBinaryReader();

while (\_incomingMessageBuffer.ReadHeader(out error))

{

object o;

int transmissionType;

// Try to read the complete message. If the message is incomplete,

// ReadMessage returns false and we exit the loop

if (!\_incomingMessageBuffer.ReadMessage(out o, out transmissionType))

break;

fullMessageReceived = true;

// Raise an event to signal a full message was received

if (o != null && MessageReceived != null)

{ MessageReceived(this, new ObjectEventArgs(o, transmissionType)); }

}

if (error == true)

{

// Clear the incoming byte chunks - see if

// we can start over with a fresh slate

\_incomingByteChunks.Clear();

}

\_incomingMessageBuffer.RemoveReceivedMessagesFromBuffer();

return fullMessageReceived;

}

## Receiving Messages

/// <span class="code-SummaryComment"><summary></span>

/// Handles incoming messages received by the server

/// <span class="code-SummaryComment"></summary></span>

public class ServerMessageHandler : IIncomingMessageHandler

{

public static int \_messagesReceived = 0;

public void HandleIncomingMessage(ICommIncomingConnection connection,

ICommunicationsMessage message)

{

Interlocked.Increment(ref \_messagesReceived);

// We could also use the "type" parameter integer and avoid reflection/etc

Type messageType = message.MessageContents.GetType();

if (messageType == typeof(HelloMessage))

HandleHelloMessage(message.MessageContents as HelloMessage);

else if (messageType == typeof(SomeGameCommand))

HandleSomeGameCommand(message.MessageContents as SomeGameCommand);

else

Console.WriteLine("Unknown command type: " + messageType.Name);

}

public void HandleSomeGameCommand(SomeGameCommand command)

{

Console.WriteLine(

"Server received game command number " +

command.CommandNumber.ToString() +

" with message " + command.Message +

" and " + command.AdditionalParameters.Count.ToString() +

" additional parameters");

}

public void HandleHelloMessage(HelloMessage hello)

{

Console.WriteLine("Server received hello message: " + hello.Message);

}

}

### Detecting Dropped Connections

List<Socket> sockets = new List<Socket>();

foreach (IncomingConnection connection in incomingConnections)

{

sockets.Add(connection.TcpClientConnection.Socket);

}

// Check read

Socket.Select(sockets, null, null, 10000);

if (sockets.Count > 0)

{

// Check write

Socket.Select(null, sockets, null, 10000);

if (sockets.Count > 0)

{

// For these sockets, both read

// and write are bad - client dropped connection

foreach (Socket s in sockets)

{

foreach (IncomingConnection connection in incomingConnections)

{

if (connection.TcpClientConnection.Socket.Handle.ToInt32() ==

s.Handle.ToInt32())

connection.NeedsClosing = true;

}

}

}

}