Articles » General Programming » Internet / Network » Client/Server Development

Design and Implementation of a High-performance TCP/IP Communications Library

owen654321, 3 Aug 2008

Rate this:

4.89 (54 votes)

A TCP/IP Communications Library, designed to handle client-server data transmission for a massive multiplayer online game.

Download Communications Library source and demo - 126 KB

Introduction

This article is the second of a multi-part series that will cover the architecture and implementation of components needed to create and maintain a robust, scalable, high performance, massive multiplayer online game server and game engine.

The first article of the series focused on building a Scheduling Engine to drive organized, real-time change in a virtual world. The present article focuses on the design and implementation of a TCP/IP communication component, designed to efficiently handle communications between the game server and remote game clients (players).

Background: BBS Games to MUDs

Back in the 80's, I ran a modem-based bulletin board system (BBS) that let users dial in and leave messages for other users, share files, and play simple multi-player games. BBS-hosted games (anyone remember TradeWars?) were multiplayer in the sense of having a common, persistent virtual world with which multiple players could interact.

Each day, players would dial in to the bulletin board system to use up their quota of daily strategic moves and to see what other players had done in/to the virtual world. Because only one person could play the game at a time, real-time player interaction was not possible.

When I headed off for college in the early 90's, I had to shut down the BBS. Just when I thought my days of online games were over, I found out **my dorm room** had an Ethernet outlet...

A friend from high school, now living hundreds of miles way, introduced me to MUDs (online multiplayer games) as a way to meet up online and have fun. Players connected to MUDs using Telnet to connect to a server via TCP/IP, and then Telnet sent the player's typed commands to the game server, allowing interaction with the game. Best of all, my friend and I could both play and interact in the same virtual world **at the same time**.. Whereas BBS games involved multiple players taking turns, Internet MUDs involved multiple players all playing simultaneously. With Ethernet access in my room and his encouragement, it didn't take me long to get hooked. I started as a player, and when I became bored of playing, I moved on to world creation. When I became bored of world creation, I set up and hosted my own MUD from my dorm room Ethernet, and began making modifications and enhancements to standard MUD code.

While working on MUDs, I spent a lot of time learning the ins and outs of the Berkeley sockets API which handled client communication with the game engine. I'd like to think that my experience with socket communication back in the 90's was not wasted, and in fact, when I started snooping around the modern .NET TCP/IP client- and server-related objects, I found the Berkeley sockets API skeleton hiding in the closet.

So, without further ado, I present to you my own, modernized implementation of a high-performance TCP/IP server, complete with discussion of (and solutions to) various socket-related quirks I encountered along the way.

Terminology - Sockets, and Outgoing / Incoming Connections

> "An Internet socket (or commonly, a socket or network socket), is an end-point of a bidirectional communication link that is mapped to a

computer process communicating on an Internet Protocol-based network, such as the Internet." (Wikipedia).

In my own words, a network socket is an object that, when connected, can send and receive data to/from another computer. A socket can be in a connected or disconnected state, and can send and receive data. Pretty straightforward...

A socket object is created when your computer initiates a connection to a remote host. A connection request is sent to a specific port on a remote host. If the remote host accepts the connection, the socket you created enters a connected state, allowing you to transmit and receive data to/from the remote host. In this article, **an Outgoing Connection refers to a socket object, created locally, used to initiate contact and communicate with a remote host**.

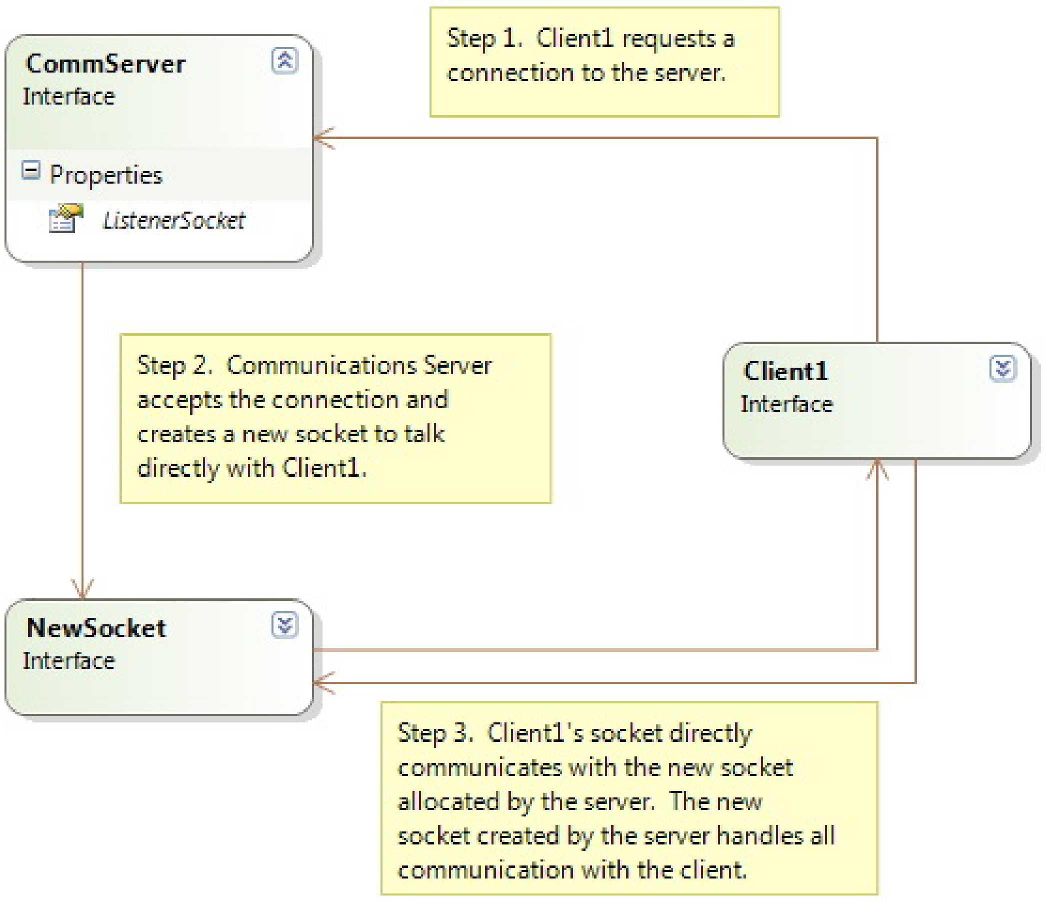
A networked computer also has the capability of listening for incoming connection requests. To listen for incoming connections, a socket is created and then bound (assigned) to a specific port / address. That socket is then instructed to listen for incoming connections. When a remote host requests a connection to the specified port on your computer (and your computer accepts the connection request), an **additional**, dedicated socket object is created on your computer to handle data transmission with that specific remote host. In this article, **an Incoming Connection refers to a connected socket, created in response to fulfilling a remote host's request for a connection with your computer**.

Peer-to-Peer vs. Client/Server

In peer-to-peer communication, a single machine acts as both a server (receiving incoming connections) and as a client (initiating outgoing connections to remote hosts). There are just enough differences between incoming and outgoing connections to make implementing peer-to-peer software annoying. Examples: Firewalls may prevent incoming connections from a remote host, but still allow connection to the same remote host via the use of an outgoing connection request. And, there are other small differences between outgoing and incoming connections: outgoing connections point to a remote host name and port, and if connectivity breaks, you may be able to use the same host name and port information to reestablish the outgoing connection. However, the sockets used to handle incoming connections are assigned their own port number when a listener/server socket accepts an incoming connection. Therefore, if connectivity with an incoming connection is lost, you cannot simply reconnect to the remote machine using the host name and port associated with the (now-broken) incoming connection.

Okay, enough confusion. For my sake and yours, this is not a peer-to-peer project. It's a straightforward client-server TCP/IP implementation. Here is how it works:

- The client is responsible for initiating a connection to the server. Therefore, the client only needs to process and take care of **a single outgoing connection**.
- The communications server listens for connection requests from clients. The communications server must process and take care of **multiple incoming connection requests and incoming connections**.


CommServer
Interface
Properties
ListenerSocket
Step 1. Client1 requests a connection to the server.
Step 2. Communications Server accepts the connection and creates a new socket to talk directly with Client1.
Client1
Interface
NewSocket
Interface
Step 3. Client1's socket directly communicates with the new socket allocated by the server. The new socket created by the server handles all communication with the client.

Example Use

Before delving into the design and implementation details, I'd like to present several snippets of code that demonstrate how the communications library can be used:

⊟ Collapse | Copy Code

```
// 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.");
```

Design and Implementation

For the sake of article length, I'm going to jump right into the design and implementation. The class `OutgoingConnection` is used by the game client to establish an outgoing connection to the game server. When the connection is accepted, the game server creates an instance of the class `IncomingConnection` to communicate with the connected client.

Both `OutgoingConnection` and `IncomingConnection` inherit from the common base class, `ConnectionBase`. This base class provides communication-related functions (i.e., sending and receiving data) that are common to both incoming and outgoing connections.

The interface `ICommunicationsBase` is implemented by both incoming and outgoing connections. Elements common to both incoming and outgoing connections include:

- a socket (the connection or pipeline used to send and receive messages),
- an event that will be raised whenever a new message is received from the pipeline,
- a method that can transmit data across the pipeline to the remote host, and
- a way of closing the socket if/when the connection needs to be terminated.

(The interface also inherits from `ITrackBytes`, which tracks bytes sent and received for diagnostic purposes.)

⊟ Collapse | Copy Code

```
public interface ICommunicationsBase : ITrackBytes
{
   ICommunicationsProtocol TcpClientConnection { get; set; }
   event EventHandler<ObjectEventArgs> MessageReceived;
   void SendMessage(object o, int transmissionType);
   void CloseConnection();
}
```

Outgoing Client Connection to the Server

The client's connectivity with the server is defined by the `OutgoingConnection` class. In addition to inheriting the basic communication capabilities from class `ConnectionBase`, it also contains the following:

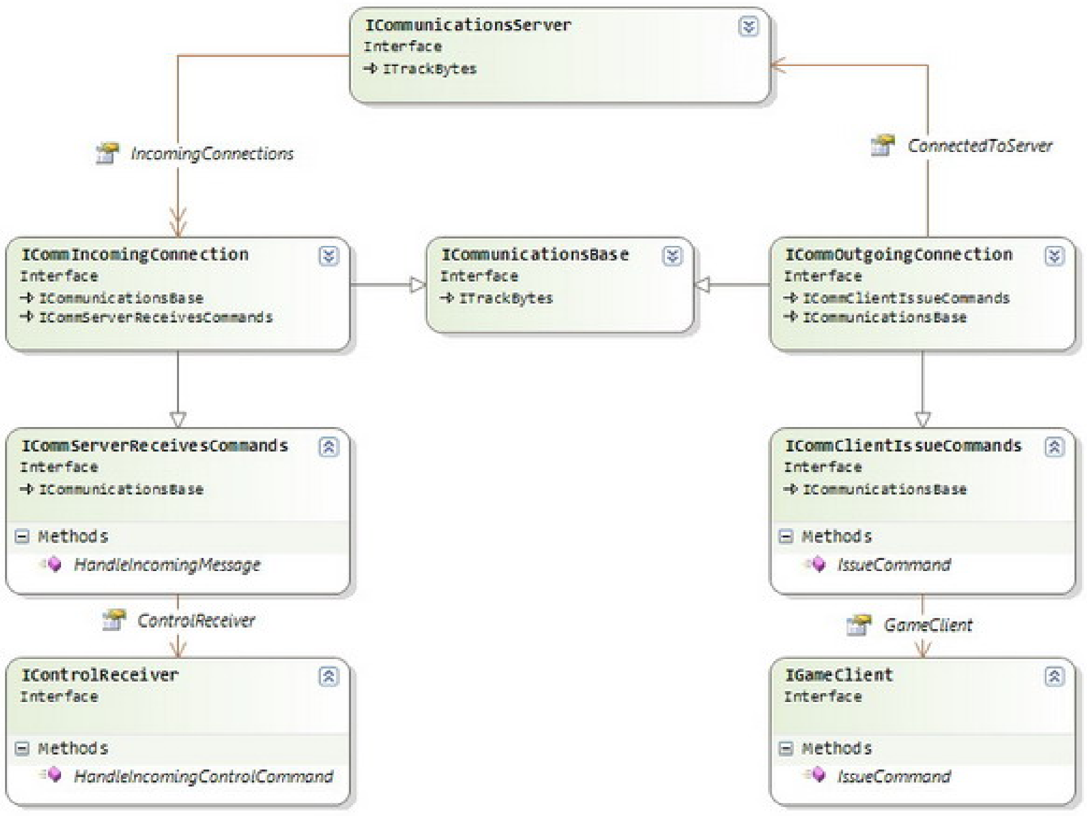
1. A method, `public bool Connect(string hostName, int portNumber)`, that is used to initiate a connection with the communications server.
2. Three threads, one dedicated to receiving incoming data, another dedicated to sending outgoing messages, and the third dedicated to extracting messages from the incoming data stream.

Incoming Client Connection on the Server

On the server side of things, an instance of the IncomingConnection class is used to track and communicate with **each connected client**. In addition to inheriting the basic communication capabilities from ConnectionBase, it also contains the following:

1. A pointer to its parent communications server object.
2. An ID which can be used to associate the connection with an Entity in the game.
3. A pointer to an incoming message handler, IIncomingMessageHandler, that can be used to transmit incoming commands from the client to the associated in-game Entity.

A visual representation of the relationship between OutgoingConnection, IncomingConnection, and ConnectionBase is presented below:



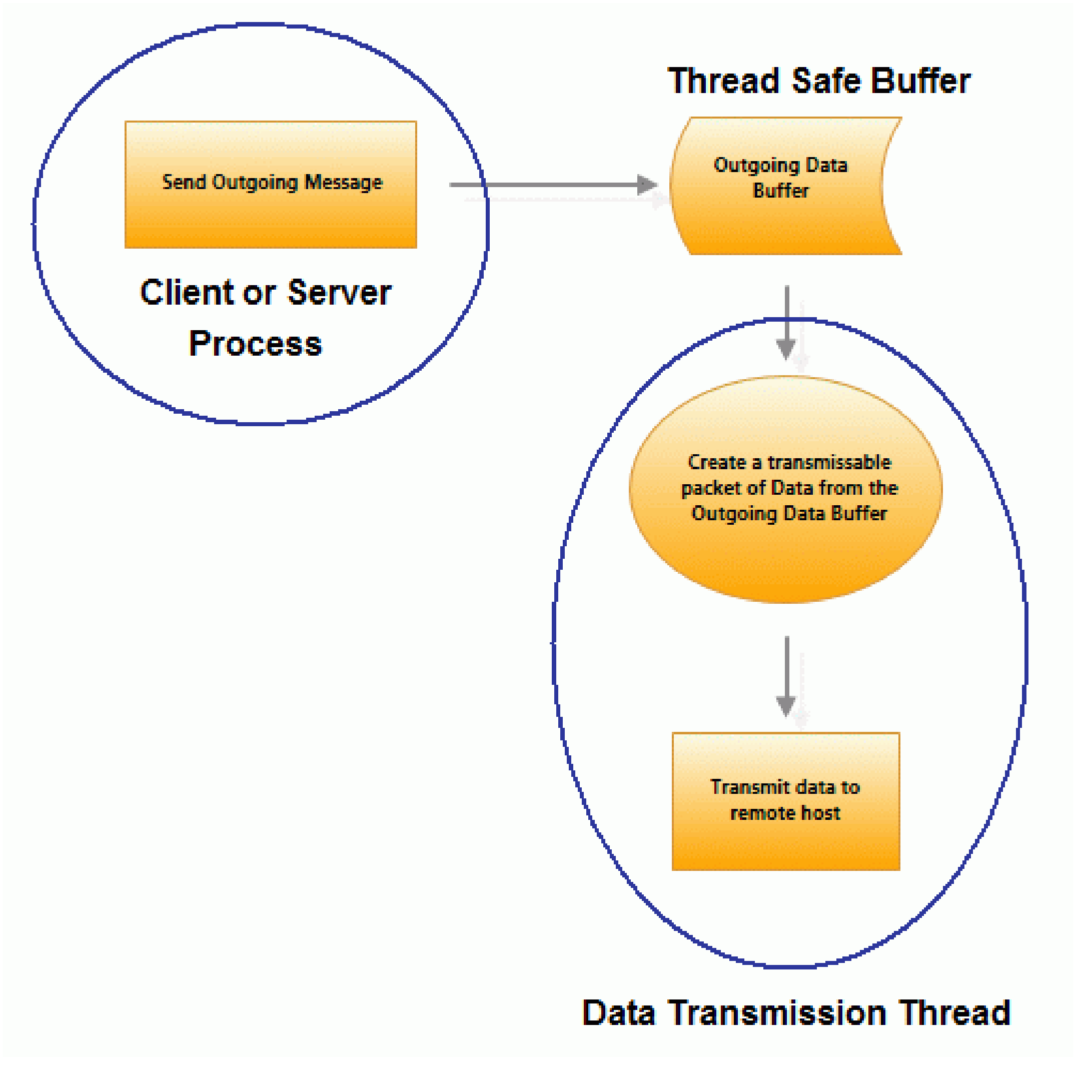
Transmitting Messages

At the heart of the client-server communication is the transmission and receipt of **messages**. A message is a **complete** parcel of information - when packaged up, sent, received, and then unwrapped, a message contains the same (complete) information as sent by the sender.

The Send/Receive Buffer

The thread-safe SendReceiveBuffer is used to buffer incoming and outgoing message data. Within the class, bytes waiting to be sent and incoming bytes waiting to be processed are stored in ThreadSafeQueue<byte[]> _outgoingByteChunks and ThreadSafeQueue<byte[]>

_incomingByteChunks, respectively.



When data is received from a remote client, it is enqueued into _incomingByteChunks, where it sits, waiting to be combined and processed. When data is ready to be sent, bytes are dequeued from _outgoingByteChunks for transmission. Because it's inefficient to transmit small chunks of data multiple times, small byte chunks are combined prior to transmission. In this way, multiple messages may be sent via a single TCP/IP data packet:

Collapse | Copy Code

```
/// <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

Blocking vs. Non-blocking Sockets

In blocking mode, sending data to a remote client halts the current thread's execution until **all data** is sent. Similarly, attempting to receive data from a remote client halts the current thread's execution **until data is received**.

When sending data to a remote client in non-blocking mode, **part or all of the data is transmitted**, and the send operation immediately returns. When receiving data, if the client has not sent any data, `receive()` will return **a value indicating that no data has yet been received**. If the client has sent data, **part or all of the transmitted data** will be read.

We don't want the communications server waiting around for send and receive operations to complete; we want to read available data that has been sent from a client and then immediately move on to read available data from the next connected client. Similarly, we want to send as much waiting data as possible to a client, and then immediately move on to send waiting data to the next client. Therefore, the current communications library implementation uses non-blocking sockets.

Dealing with Partial Data Transmission

A message is defined as a **complete** parcel of information - when packaged up, sent, received, and then unwrapped, a message contains the same (complete) information as sent by the sender. When data is transmitted via TCP/IP in non-blocking mode, complete messages may be broken apart (fragmented) into multiple packets of data for transmission. Therefore, messages may need to be reassembled into complete parcels of information when received by the recipient.

The technique most frequently used to ensure complete receipt of a message (and to help identify the type of message being transmitted) is to prefix the message with a header. In the presented solution, a message header consists of two integers (each 4 bytes). The first integer specifies the size of the complete, transmitted message, and the second integer specifies the transmitted message type:

⊟ Collapse | Copy Code

```
  4 bytes    4 bytes     n bytes
[Msg Size][Msg Type][Compressed Msg]
```

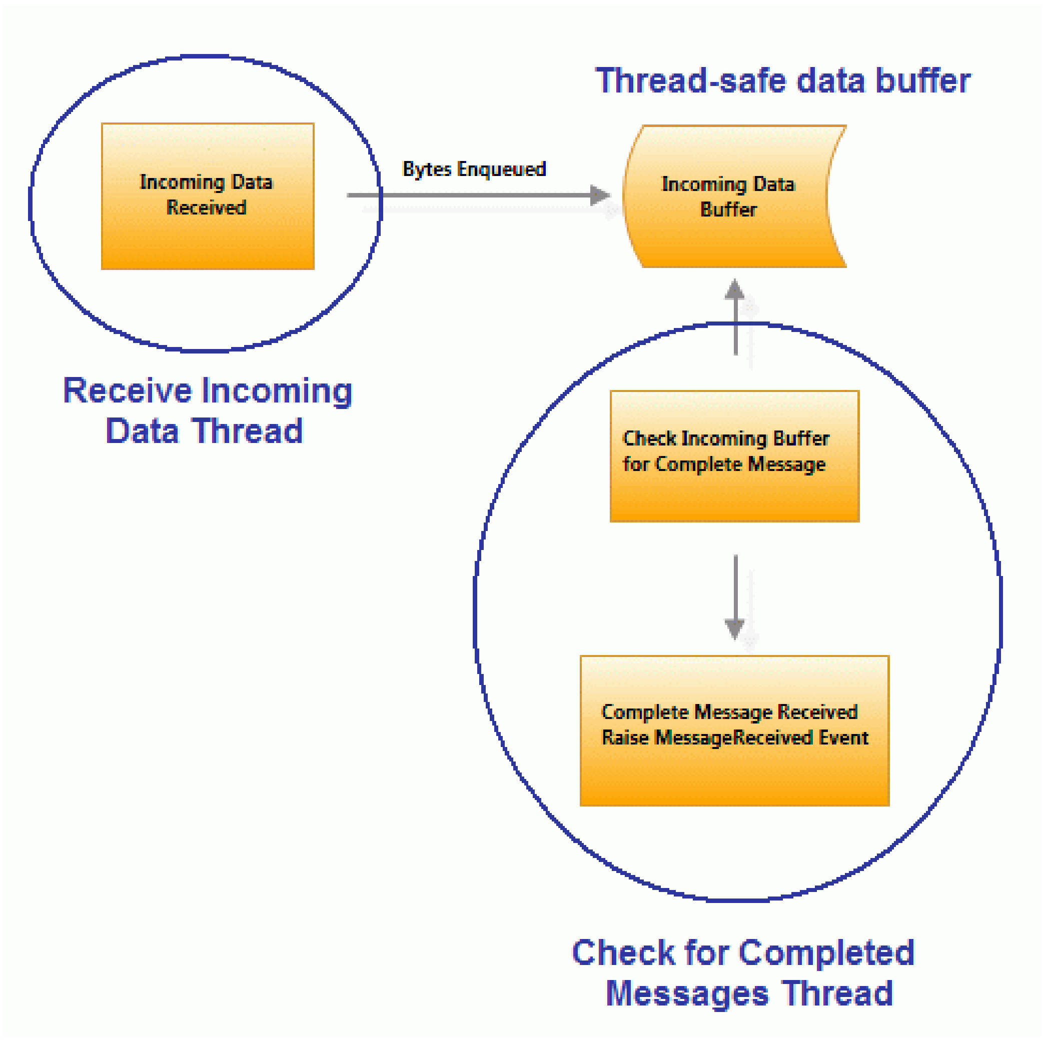
For flexibility, it would be nice to transmit **any instance of any class** across the wire. Although using binary serialization to serialize and then compress objects may not be the most efficient method of transmitting data between client and server, its flexibility and ease-of-use made it the method of choice for data transmission for this sample project. To maximize efficiency and speed in a real-world scenario, custom serialization should be implemented and used. The following method is used to convert a message (from any serializable object) into a byte array suitable for transmission.

⊟ Collapse | Copy Code

```
/// <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

Because messages may be received in multiple pieces, the message header (the first four bytes of which contain the total size of the message) is used to determine when a full message has been received. The `MessageBuffer` class is used to determine if a full message has been received and, if so, read the message from the incoming data stream. The class `MessageBuffer` uses a `MemoryStream` to store incoming data (which may or may not contain a complete message). In this way, the `MessageBuffer` class **extracts concrete messages from a continuous stream of data**.


Thread-safe data buffer
Incoming Data Received
Bytes Enqueued
Incoming Data Buffer
Receive Incoming Data Thread
Check Incoming Buffer for Complete Message
Complete Message Received Raise MessageReceived Event
Check for Completed Messages Thread

The `MessageBuffer` method `AppendNewIncomingBytePackets(ThreadSafeQueue<byte[]> incomingByteChunks)` appends newly-received data packets to `MessageBuffer`'s `MemoryStream`:

Collapse | Copy Code

```
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);
}
```

The method `bool ReadHeader(out bool error)` attempts to read a message header (8 bytes in length). If the `MessageBuffer` doesn't contain at least 8 bytes, the method will return `false`. Otherwise, the `MessageBuffer` reads and stores the header information (including the expected message size and the transmission type).

⊟ Collapse | Copy Code

```
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();
    ...
}
```

The method `bool ReadMessage(out object o, out int transmissionType)` is used to read the next message from the `MessageBuffer`, if a full message exists in the buffer. The method returns `false` if no full message is yet available (as defined by the message size in the header); otherwise, it deserializes the message into `object o` and populates the message's transmission type.

⊟ Collapse | Copy Code

```
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;
}
```

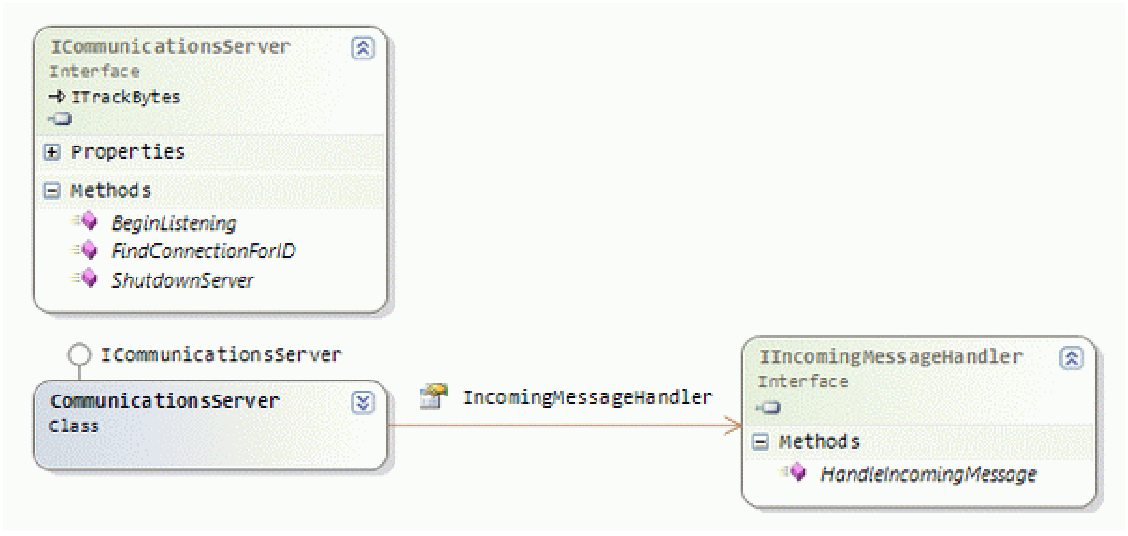
Full use of the `MessageBuffer` class is demonstrated in the method `bool`

GetNextFullMessage(). The method either reads new incoming messages or returns false if no messages (or only a partial message) are available:

Collapse | Copy Code

```
/// <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



When a complete, incoming message is received by the server, the server uses an implementation of `interface IIncomingMessageHandler` to interpret the message. In the demo project, the `ServerMessageHandler` class handles the server's receipt of incoming messages:

Collapse | Copy Code

```
/// <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);
    }
}
```

TCP/IP Tricks and Tips

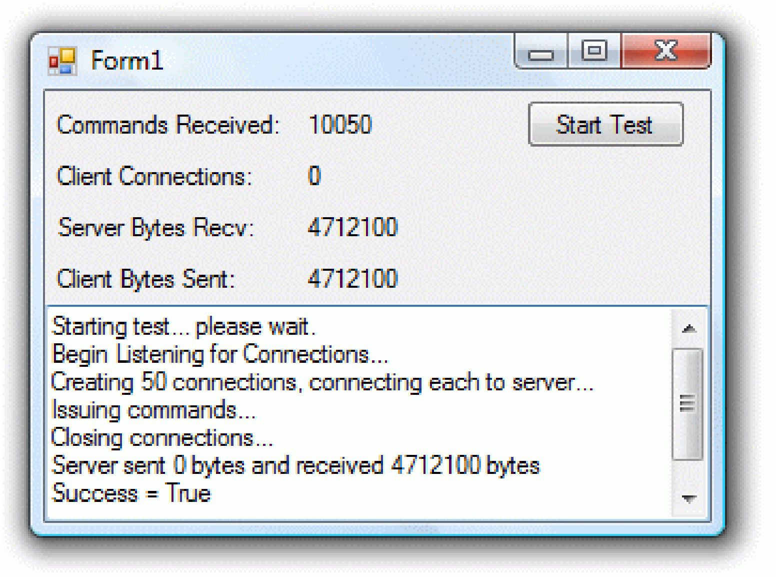
Detecting Dropped Connections

When writing a TCP/IP client and server, a frequently encountered problem is the issue of clients dropping their connections. Specifically, when a client drops its connection to the server in an abnormal manner, the server can still use the partially closed socket to "send data to the client". In other words, after a client drops his/her connection, the server may blissfully continue sending data into a black hole, seemingly unaware that the client is no longer fully connected. This actually isn't a bug; TCP/IP is a full-duplex network protocol, which means the connection can be closed "half way". Therefore, additional work is required to determine if the client's connection is fully connected (capable of both sending and receiving data to/from the server). In the presented communications library, the following (quick and dirty) code executes occasionally to determine which connections have been "dropped":

⊟ Collapse | Copy Code

```
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;
            }
        }
    }
}
```

Demo Project



```
Commands Received:   10050
Client Connections:  0
Server Bytes Recv:   4712100
Client Bytes Sent:   4712100

Starting test... please wait.
Begin Listening for Connections...
Creating 50 connections, connecting each to server...
Issuing commands...
Closing connections...
Server sent 0 bytes and received 4712100 bytes
Success = True
```

The demo/test project, TestCommunicationsWinforms, is a Windows Forms application that tests both the server and the client functionality of the communications library. The application performs the following:

1. Sets up a server socket to start listening for incoming connections on port 4006.
2. Creates 50 outgoing connections. Connects each outgoing connection to the server socket.
3. After the 50 connections have been established, each sends a Hello message to the server.
4. Each outgoing connection simultaneously sends 200 mock game commands to the server.
5. Connections are closed and the server shuts down.
6. Server received message count is compared with the number of commands sent.

Points of Interest

I hope you've enjoyed the second installment in the series (or at least picked up some useful information or snippets of code). Upcoming article topics are listed below:

- Entities, Abilities, Intentions, Actions, and States
- The Virtual World: Interaction and Change
- Data Storage and Virtual World Persistence
- Plug-ins and Extensibility

All comments/suggestions are welcome.

- owen@binarynorthwest.com
- Member BrainTechLLC

License

This article, along with any

Share

EMAIL

About the Author

owen654321

Software Developer (Senior) Troppus Software
United States

Currently working as a Senior Silverlight Developer with Troppus Software in Superior, CO. I enjoy statistics, programming, new technology, playing the cello, and reading codeproject articles.

You may also be interested in...

Gartner: Magic Quadrant for On-Premises Application Platforms

Expanding active decision-making: The power of integrating business rules and events

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

Profile popups Spacing Relaxed Noise Medium Layout Normal Per page 25 Update

First Prev Next

Advantages over ZeroMQ?	**pt1401**	**4-Jan-13 11:59**
hi	**Pankaj_Pundir**	**22-Apr-12 1:55**
[Bug Reporting] I think this code should be modified.	**creatinova**	**3-Nov-10 21:57**
nice work	**duyao**	**2-Jul-10 21:53**
AssociatedID missing a unique value?	**Lucradev**	**3-May-10 4:57**
Re: AssociatedID missing a unique value?	owen654321	3-May-10 7:16
Re: AssociatedID missing a unique value?	Lucradev	3-May-10 12:33
event ClientConnecting or ClientConnected	**dim.grey**	**24-Sep-09 5:23**
What am I doing wrong?	**Kennetho**	**21-Jun-09 6:33**
Re: What am I doing wrong? [modified]	M-Dawg	24-Jul-09 9:29
Update...	**owen654321**	**21-May-09 4:33**
[Message Deleted]	**it.ragester**	**2-Apr-09 21:43**
error report!	**happyokok**	**4-Dec-08 19:16**
Re: error report!	owen654321	21-May-09 4:25

Re: error report!	owen654321	22-Jul-09 6:42
How did you make this?!	**secutos1**	**6-Nov-08 20:58**
Re: How did you make this?!	Ngan Pham	29-Nov-08 12:46
Thanks for the exellent article! I can't drop any word to describe how cool it is? Thanks again :)	**Sacxophone**	**5-Oct-08 21:13**
A better way	**sergiols**	**27-Sep-08 11:15**
I m trying to do a client - server standalone console applications (HELP!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!).	**[Dolphin]**	**15-Sep-08 6:26**
Re: I m trying to do a client - server standalone console applications (HELP!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!).	owen654321	15-Sep-08 12:16
Re: I m trying to do a client - server standalone console applications (HELP!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!).	[Dolphin]	16-Sep-08 22:01
Re: I m trying to do a client - server standalone console applications (HELP!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!).	czllfy	19-Aug-10 20:39
serializing objects	**Shukaido**	**7-Aug-08 9:08**
Re: serializing objects	owen654321	7-Aug-08 12:17

Last Visit: 31-Dec-99 18:00 Last Update: 8-Sep-14 0:01 Refresh 1 2 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.