

GatherBird HummingBird Service Bus Programmers Guide

version 0.1 – March 2010



Table of Contents

Basic Principles.....	3
Messages.....	4
Building and Contributing.....	6
Quick Start Example.....	7
Quick Start Example Source Code.....	8
More on the example programs.....	11
HummingBird Configuration and Startup.....	12
HummingBirdConnect.....	13
Connecting.....	13
Send.....	13
Receive (polling).....	13
Receive (Asynchronous).....	13
HummingBirdConnect Utility functions.....	14
Register or Login.....	14
Messages.....	14
Transactions.....	14
HummingBirdConnect Exceptions.....	15
FAQ.....	16
Dictionary.....	17
Broadcast message	17
Client.....	17

Client Messages.....	17
Envelope.....	17
GUID.....	17
Message.....	17
Message Type.....	18
Publish.....	18
Route back message.....	18
Routing.....	20
Transactions.....	20
Tracing.....	20
Command Message Type.....	20
Commit Message Type.....	21
Begin Work Message Type.....	21
Login Message Type.....	21
Subscribe Message Type.....	22
UN-Subscribe Message Type.....	22
Design Decisions.....	24
Version 0.1 – TODO.....	25

Basic Principles

- Clients can connect to the HummingBird through TCP/IP or a named pipe.
- The HummingBird can receive XML messages and send XML messages.
- All XML messages sent to the HummingBird must contain a message type.
- Clients can subscribe to message types.
- The HummingBird routes XML messages to clients that are subscribed to the message type.
- There are 5 message types with special meaning to the HummingBird; subscribe, unsubscribe, login, begin work, and commit.
- Any message with a message type that is not one of the 5 command message types will be routed to clients subscribed clients.

Messages

HummingBird supports five functions. Each one is exercised by sending XML strings to the server like the following:

1 Login. Allows the HummingBird to associate a human readable name to the connected client. Helpful for debugging XML messages as they get passed around between your applications.

```
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_LOGIN</hbc_MessageType>
  <hbc_ClientName>your application name</hbc_ClientName>
</hbc_Message>
```

2 Subscribe. Used when a client wants to receive messages published by other applications. Messages are always categorized by a message type.

```
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_SUBSCRIBE</hbc_MessageType>
  <hbc_SubscribeToThisMessageType>message type</hbc_SubscribeToThisMessageType>
</hbc_Message>
```

3 Unsubscribe. Used when a client no longer wishes to receive certain messages.

```
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_UNSUBSCRIBE</hbc_MessageType>
  <hbc_UnSubscribeFromThisMessageType>your
type</hbc_UnSubscribeFromThisMessageType>
</hbc_Message>
```

4 Begin a transaction. After sending a begin work message, all clients who get routed messages published by this client will be limited to only receiving messages from this client until a commit command is performed.

```
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_BEGINWORK</hbc_MessageType>
  <hbc_TimeOut>10000</hbc_TimeOut>
</hbc_Message>
```

5 Ends a transaction. Clients who were exclusively locked by this client will now receive messages from other clients normally.

```
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_COMMIT</hbc_MessageType>
</hbc_Message>
```

6 Also, any client application can publish it's own message that the HummingBird routes to all clients who have subscribed to the message type.

```
<hbc_Message>
  <hbc_MessageType>your_message</hbc_MessageType>
  <mydata>any extra XML tags and information you want<mydata>
```

</hbc_Message>

Building and Contributing

Email Ben Fields at hummingbird@gatherbird.com if you would like to contribute.

This source code was built using Microsoft Visual Studio 2005. However, the named pipe references is from .NET 3.5. I installed the .NET 3.5 references by way of Visual Studio Express 2008.

The DotNetLibrary files are utility classes intended to generically useful to any .NET application. Nothing HummingBird specific should be included in the RFLib source files.

The HummingBird\BuildAll.bat and \DotNetLibrary\UnitTests\RunUnitTests.bat contain hard coded paths to devenv.exe. This may not work on your development machine.

The .FxCop files are from a code analysis utility from Microsoft. This tool provides some “code standards” guidelines for the HummingBird project. Google for “microsoft fxcop download” to find this tool.

Quick Start Example

Source Code:

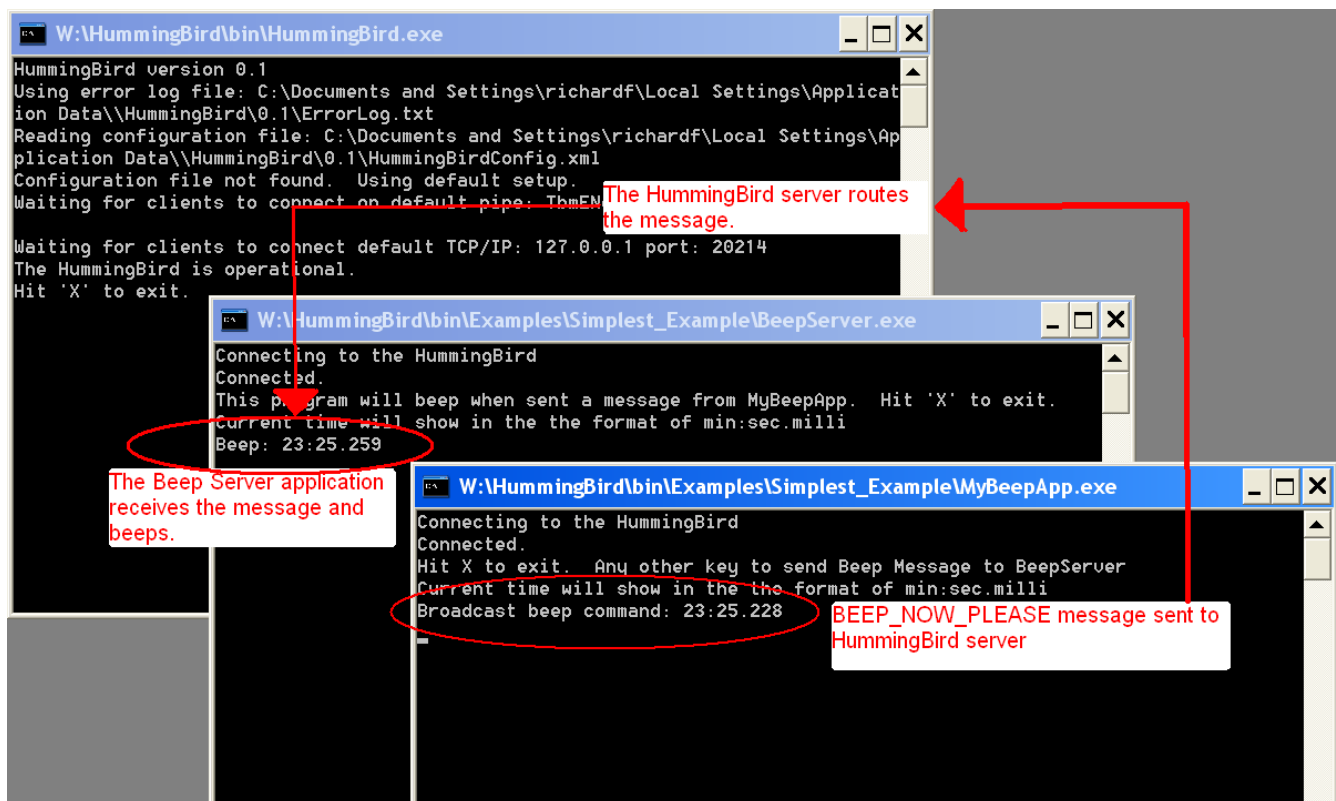
\Examples\Simplest_Example

Run bin\HummingBird.exe. By default, when a HummingBirdConfig.xml file is not found the HummingBird opens one hard coded named pipe and one TCP/IP port where clients can connect.

Run \bin\Examples\Simplest_Example\BeepServer.exe. The Beep Server application will connect to the HummingBird through TCP/IP and subscribed to BEEP_NOW_PLEASE messages.

Run the \bin\Examples\Simplest_Example\MyBeepApp.exe. The MyBeepApp will connect to the HummingBird through a named pipe. When a key is pressed MyBeepApp will publish a BEEP_NOW_PLEASE message to the HummingBird.

All BEEP_NOW_PLEASE messages received by the HummingBird are routed to the BeepServer. The actual beep noise is commented out in the source code because it is annoying.



Quick Start Example Source Code

Before running any example programs you should start the HummingBird.exe server in HummingBird\bin.

“Client” MyBeepApp.exe

Source \Examples\Simplest_Example\MyBeepApp
EXE \bin\Examples\Simplest_Example

Note: The same functionality is duplicated in another example called Simplest_Example_Using_HummingBirdConnectDLL which uses the HummingBirdConnect.dll to remove some of the inconvenience of connecting, receiving, and parsing the XML messages.

Step 1

MyBeepApp connects to the HummingBird using this C# code (or any language that can open a named pipe) like this:

```
NamedPipeClientStream client = new  
NamedPipeClientStream(".", HBC.DEFAULT_SERVER_PIPE_NAME, PipeDirection.InOut,  
PipeOptions.Asynchronous);
```

Or, using the HummingBirdConnect.dll

```
HBC_Connection conn = new HBC_Connection(HBC.DEFAULT_SERVER_PIPE_NAME, 5000, 5000);
```

Step 2

The application then publishes a message by sending XML to the HummingBird using C# code like this:

```
XMLFragment_class writer = new XMLFragment_class();  
writer.WriteRaw(HBC.ENVELOPE_START);  
writer.WriteStartElement(HBC.XML_ELEMENT_MESSAGE);  
  writer.WriteElementString(HBC.XML_ELEMENT_MESSAGE_TYPE, SS_class.BEEP_NOW_PLEASE);  
writer.WriteEndElement();  
writer.WriteRaw(HBC.ENVELOPE_END);  
byte[] sendme_bytes = writer.ToArray();  
client.Write(sendme_bytes, 0, sendme_bytes.Length);
```

Or, using the HummingBirdConnect.dll

```
byte[] message = MSU.CreatePublishMessage(SS_class.BEEP_NOW_PLEASE, "");  
conn.HBC_Send(message);
```

This is the XML produced above

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE  
<hbc_Message>  
  <hbc_MessageType>BEEP_NOW_PLEASE</hbc_MessageType>
```

```
</hbc_Message>  
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Note: The special GUIDs are required by HummingBird to mark the beginning and ending of every XML message.

“Service” side BeepServer.exe

Step 1

BeepServer.exe connects to the the HummingBird using C# code (or any language) like this:

```
TcpClient client = new TcpClient();  
NetworkStream netstream = client.GetStream();
```

Or, using the *HummingBirdConnect.dll*

```
HBC_Connection conn = new HBC_Connection(HBC.DEFAULT_TCPIP_ADDRESS,  
HBC.DEFAULT_TCPIP_PORT, 5000, 5000);
```

Step 2

BeepServer.exe then sends XML (a [message](#)) to the HummingBird to subscribe to the BEEP_NOW_PLEASE message type using C# code like this:

```
XMLFragment_class riter = new XMLFragment_class();  
writer.WriteRaw(HBC.ENVELOPE_START);  
writer.WriteStartElement(HBC.XML_ELEMENT_MESSAGE);  
writer.WriteElementString(HBC.XML_ELEMENT_MESSAGE_TYPE,  
HBC.MESSAGE_TYPE_COMMAND_SUBSCRIBE);  
writer.WriteElementString(HBC.XML_ELEMENT_SUBSCRIBE_TO,SS_class.BEEP_NOW_PLEASE);  
writer.WriteEndElement();  
writer.WriteRaw(HBC.ENVELOPE_END);  
netstream.Write(writer.ToArray(), 0, writer.Length());
```

Or use the *HummingBirdConnect.dll*

```
byte[] message = MSU.CreateSubscribeToMessage(SS_class.BEEP_NOW_PLEASE);  
conn.Send(message);
```

This is the XML produced above

```
MjAskpBWkD4zctG2TPPdAS3n6ySc4GE  
<hbc_Message>  
  <hbc_MessageType>HBC_COMMAND_SUBSCRIBE</hbc_MessageType>  
  <hbc_SubscribeToThisMessageType>BEEP_NOW_PLEASE</hbc_SubscribeToThisMessageType>  
</hbc_Message>  
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

The HBC_COMMAND_SUBSCRIBE message type is a HummingBird command message (it does not get routed).

BeepServer.exe has now subscribed to BEEP_NOW_PLEASE messages which will be published by other clients.

Step 3

BeepServer.exe can receive messages from the HummingBird using code like this

```
const int MAX_MESSAGE_LENGTH = 1024;
byte[] bytebuffer = new byte[MAX_MESSAGE_LENGTH];
int bytesread = netstream.Read(bytebuffer,0,MAX_MESSAGE_LENGTH);
```

Or use the *HummingBirdConnect.dll*

```
byte[] message = conn_p.HBC_GetMessage();
```

BeepServer.exe can expect to receive messages like this

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_AddedByHummingBird>
    <hbc_ThisMessageIDForRouteBack>
      ea9bTpkpwzEAPLr4z7DqW55EzjtHn2C
    </hbc_ThisMessageIDForRouteBack>
    <hbc_ThisMessageWasCreatedBy>Simple Beep Client</hbc_ThisMessageWasCreatedBy>
  </hbc_AddedByHummingBird>
  <hbc_MessageType>BEEP_NOW_PLEASE</hbc_MessageType>
</hbc_Message>
9zA8bzLDDrRBFqRgmGZGEFAfqMfnb8KHq
```

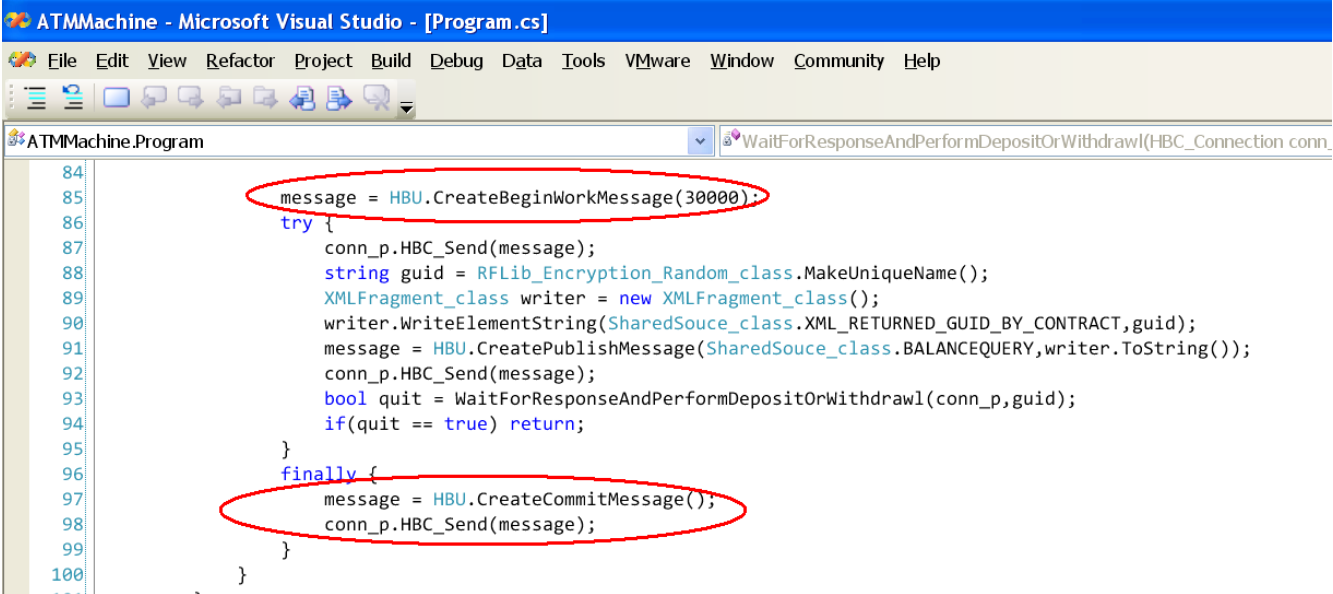
The HummingBird inserts XML into each routed message. The inserted XML allows the subscribed client to send a response back to the originator of the message using the `hbc_ThisMessageIDForRouteBack` tag. Also the originator of the message is identified in the `hbc_ThisMessageWasCreatedBy` tag.

More on the example programs

\bin\Examples\Transactions

An example to test transactions is represented by a Banks server and ATM machines. In this example it is intended that one instance of BankOfAntiochServer.exe should be run. Multiple instances of ATMMachine.exe should be run.

To see how the ATM machines would fail without transaction support from the HummingBird comment out these lines in the ATMMachine source file



```
84
85     message = HBU.CreateBeginWorkMessage(30000);
86     try {
87         conn_p.HBC_Send(message);
88         string guid = RFLib_Encryption_Random_class.MakeUniqueName();
89         XMLFragment_class writer = new XMLFragment_class();
90         writer.WriteElementString(SharedSouce_class.XML_RETURNED_GUID_BY_CONTRACT, guid);
91         message = HBU.CreatePublishMessage(SharedSouce_class.BALANCEQUERY, writer.ToString());
92         conn_p.HBC_Send(message);
93         bool quit = WaitForResponseAndPerformDepositOrWithdrawl(conn_p, guid);
94         if(quit == true) return;
95     }
96     finally {
97         message = HBU.CreateCommitMessage();
98         conn_p.HBC_Send(message);
99     }
100
101
```

\bin\Examples\Logging

The logging example is in a beta stage. The potential benefits of a good logging are almost without limit. Future version of the HummingBird server will focus on logging.

HummingBird Configuration and Startup

When the HummingBird first starts it needs to establish a working directory. This is where it looks for the HummingBirdConfig.xml configuration file and may create an ErrorLog.txt file which is used to log catastrophic errors.

The default working directory In Windows XP is:

C:\Documents and Settings\

In Windows Vista and 7 the default working directory is:

C:\Users\

The default working directory can be changed by using the -d command line parameter. For example

HummingBird.exe -dC:\mydata

The HummingBirdConfig.xml file specifies the TCP/IP addresses and named pipes that the HummingBird will open and allow connections from clients. If the HummingBirdConfig.xml file can not be found then the HummingBird uses a default TCP/IP and named pipe which will match this configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<HummingBird>
  <HummingBirdINIFileVersion0.1>
    <Listen>
      <TCPIP>127.0.0.1:20214</TCPIP>
      <Pipe>TbmENdCYwCcaEseCHSetyztzmmRHp7e27</Pipe>
    </Listen>
  </HummingBirdINIFileVersion0.1>
</HummingBird>
```

It is possible to configure the HummingBird to listen on multiple TCP/IP ports and multiple named pipes.

```
<?xml version="1.0" encoding="utf-8"?>
<HummingBird>
  <HummingBirdINIFileVersion0.1>
    <Listen>
      <TCPIP>74.125.95.99:21215</TCPIP>
      <TCPIP>74.125.95.99:21216</TCPIP>
      <Pipe>MyNamedPipe1</Pipe>
      <Pipe>MyNamedPipe2</Pipe>
    </Listen>
  </HummingBirdINIFileVersion0.1>
</HummingBird>
```

Note: There is no need to add \\.\pipe\ to the beginning of the named pipe names.

HummingBirdConnect

The HummingBirdConnect.dll (source in \HummingBirdConnect) provides an easy way to connect to and use the facilities of the HummingBird.

Each of these examples use the HummingBirdConnect.dll

\Examples\Simplest_Example_Using_HummingBirdConnectDLL

\Examples\Simplest_Example_Asynch_Using_HummingBirdConnectDLL

\Examples\Logging

\Examples\Transactions

```
namespace HummingBirdConnect_NS
public class HBC_Connection : IDisposable
```

Connecting

```
public HBC_Connection(string pipename_p, int connection_timeout_milliseconds_p, int
cleanup_threads_timeout_milliseconds_p)
```

```
public HBC_Connection(string address_p, int port_p, int
connection_timeout_milliseconds_p, int cleanup_threads_timeout_milliseconds_p)
```

Send

```
public void Send(byte[] message_p)
```

Receive (polling)

```
public byte[] GetMessage ()
```

```
public byte[] GetMessageWait (WaitHandle breakout_p)
```

Receive (Asynchronous)

```
public void AsyncReceiveDelegate (ReceiveDelegate call_p, ErrorDelegate
errorcall_p, Object pass_p, int cleanup_threads_timeout_milliseconds_p)
```

```
public delegate void ReceiveDelegate (byte[] message_p, Object pass_p);
public delegate void ErrorDelegate (Exception ex_p, Object pass_p);
```

HummingBirdConnect Utility functions

Utility Functions provided by the HummingBirdConnect

```
namespace HummingBirdUtility_NS  
public static class MSU
```

Register or Login

```
static public byte[] CreateRegisterNameMessage (string application_name_p)
```

Messages

```
static public byte[] CreateSubscribeToMessage (string  
subscribe_to_this_message_type_p)
```

```
static public byte[] CreateUnSubscribeFromMessage (string  
unsubscribe_from_this_message_type_p)
```

```
static public byte[] CreatePublishMessage (string message_type_p, string more_xml_p)
```

```
static public byte[] CreatePublishReplyMessage (string message_type_p, string  
replying_to_this_XML_message_p, string more_xml_p)
```

Transactions

```
static public byte[] CreateBeginWorkMessage (int timeoutinmilliseconds_p)
```

```
static public byte[] CreateCommitMessage ()
```

HummingBirdConnect Exceptions

```
using HummingBirdConnectException_NS;
```

Typically more information is provided in the InnerException.

```
public class HummingBirdConnect_APIUsageException : Exception
```

GetMessage and GetMessageWait will throw this exception if the AsyncReceiveDelegate has been called. You can not simultaneously call the polling GetMessage functions and receive messages asynchronously.

AsyncReceiveDelegate will throw this exception if it is called twice. It will also throw this exception if the [ReceiveDelegate](#) or [ErrorDelegate](#) are passed as null.

```
public class HummingBirdConnect_CanNotConnectException : Exception
```

Both the TCP/IP and named pipe version of HBC_Connection will throw this exception. The InnerException will contain more a more specific exception like [TimeoutException](#), [RFLibProgrammerErrorException](#), or [SocketException](#).

```
public class HummingBirdConnect_DisconnectException : Exception
```

Send, GetMessage, and GetMessageWait will throw this exception if the connection is lost.

Also, the [ErrorDelegate](#) passed to the AsyncReceiveDelegate function will receive this exception when the connection is lost.

```
public class HummingBirdConnect_InternalErrorException : Exception
```

All function may throw this exception. This exception would indicate that there is a problem that should be fixed in the HummingBird source code.

Also, the [ErrorDelegate](#) passed to the AsyncReceiveDelegate function will receive this exception.

```
public class HummingBirdConnect_CorruptDataErrorException : Exception
```

GetMessage, and GetMessageWait will throw this exception if data is received that is not enclosed properly in a beginning or ending envelope.

Also, the [ErrorDelegate](#) passed to the AsyncReceiveDelegate function will receive this exception.

FAQ

My application needs know when another application which is connected to the Humming has disconnected (gone down).

The best way to do this is have your application subscribe to the client disconnect trace message like this

```
byte[] message =  
HBU.CreateSubscribeMessage(HBT.MESSAGE_TYPE_TRACE_CLIENTDISCONNECT);  
conn.HBC_Send(message);
```

then when the application receives the client disconnect message determine what client disconnected with code like this

```
if(value == HBT.MESSAGE_TYPE_TRACE_CLIENTDISCONNECT) {  
    string clientname = findme["<" + HBC.XML_ELEMENT_MESSAGE + "><" +  
        HBT.XML_ELEMENT_TRACE_REGISTER_CLIENT_NAME + ">"];  
}
```

Dictionary

Broadcast message

A message sent from a client to the HummingBird containing a message type that does not match a pre-defined HummingBird [command](#) message type. After receiving a broadcast message the HummingBird routes the message to all clients that have subscribed to the message type.

Client

A process or thread that establishes a connection to the HummingBird through TCP/IP or a named pipe. A client does not have to be running on the same computer as the HummingBird. From the perspective of the HummingBird any process that connects to it is considered a client. From the perspective of the distributed application this client may be considered a server. For example the Bank of Antioch server in example ZZZ is considered a server from the applications perspective but is just a client to the HummingBird.

Client Messages

Messages created by clients and sent to the HummingBird which are not [command](#) messages are called client messages. These messages will either be routed by the HummingBird to all clients subscribed to that message type (publish messages) or the message will be routed to a single client (route back messages).

Envelope

An envelope is a special character or string of characters used frequently in applications that send and receive data streams. The HummingBird uses a string of characters in the format of a [GUID](#) to mark the beginning and ending of messages in a data stream. The beginning and ending envelope GUID characters are pre-defined by the HummingBird as

MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE and 9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq

GUID

Globally unique identifier. GUIDs created by the HummingBird are typically a 32 byte long character string containing numeric and mixed case characters. The HummingBird creates GUIDs for the purpose of identifying clients that need to receive a [route back](#) message.

Message

XML formatted text enclosed in a begin and end [envelope](#). All messages sent to the HummingBird at least contain the `<hbc_Message>` `<hbc_MessageType>` XML tags.

Example of a [command](#) message sent by a [client](#) to the HummingBird enclosed by the beginning [envelope](#) characters (MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE) and ending envelope characters (9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq).

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_SUBSCRIBE</hbc_MessageType>
  <hbc_SubscribeToThisMessageType>customer location
  query</hbc_SubscribeToThisMessageType>
</hbc_Message>
```

```
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Message Type

After receiving a [message](#) the HummingBird will determine what action to take according to the message's message type. The message type is specified by the `<hbc_MessageType>` XML tag. There are two categories of messages, those with a [command message type](#) and those with a [client message type](#). The HummingBird treats client message types as either a message to [broadcast](#) to other clients or as a route [back message](#) to be sent to a single client. It is an error to send the HummingBird a message without a message type.

Example of a *command message* sent by a [client](#) to the HummingBird indicating that client wants to start working within a [transaction](#). The message type (HBC_COMMAND_BEGINWORK) is predefined (reserved) by the HummingBird.

```
MjAskrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_BEGINWORK</hbc_MessageType>
  <hbc_TimeOut>30000</hbc_TimeOut>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Example of a *client message* sent to the HummingBird. In this case an ATM banking machine want to send a message to the Bank of Antioch's server indicating that 50 dollars should be deposited in an account.

```
MjAskrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>DEPOSIT_FUNDS_INTO_BANK_ANTIOCH</hbc_MessageType>
  <Amount>50</Amount>
  <Account Number>5123232210</Account Number >
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Publish

When a client sends a [client message](#) to the HummingBird it is referred to as publishing the message. The client is sometimes referred to as the "sending client".

Route back message

Routeback messages are created by clients who received a message from another client and wish to send a response back to that particular client only. The HummingBird knows who to route the message to by the `<hbc_ThisMessageIDForRouteBack>` guid which was inserted into the original message by the HummingBird.

A typical scenario of an ATM (Automated Teller Machine) and a server at a Bank.

By convention the two clients use `<My_Amount>` and `<My_GUID>` to contain specific application data. The `<My_Amount>` value indicates how much money will be deposited. The Bank of Antioch server echoes back the `<My_GUID>` data element in its response message so that

Automated Teller Machine can match the the response to the original message that was published. The Automated Teller Machine sends this message to the HummingBird.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>DEPOSIT_FUNDS_INTO_BANK_ANTIOCH</hbc_MessageType>
  <My_Amount>50</My_Amount>
  <My_GUID>jWskz7WFPk92jnFmXLpWWmRWJ25CgE9g</My_GUID>
</hbc_Message>
```

Bank A has previously subscribed to DEPOSIT_FUNDS_INTO_BANK_ANTIOCH messages so the HummingBird routes this message to the Bank of Antioch client. The HummingBird has inserted the XML tag `<hbc_ThisMessageIDForRouteBack>` into the message. The GUID contained in this tag will be used by the Bank of Antioch server to send a route back message back to only the ATM machine that original published the DEPOSIT_FUNDS_INTO_BANK_ANTIOCH message.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_AddedByHummingBird>
    <hbc_ThisMessageIDForRouteBack>DwCSN3CcC2FwQsaDyXzZafgZqQwyLrtM</hbc_ThisMessageIDForRouteBack>
    <hbc_ThisMessageWasCreatedBy>Automated Teller 1</hbc_ThisMessageWasCreatedBy>
  </hbc_AddedByHummingBird>
  <hbc_MessageType>DEPOSIT_FUNDS_INTO_BANK_ANTIOCH</hbc_MessageType>
  <My_Amount>50</My_Amount>
  <My_GUID>jWskz7WFPk92jnFmXLpWWmRWJ25CgE9g</My_GUID>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

After the Bank of Antioch server processes the message it will send a response back to Automated Teller 1. By convention the Automated Teller Machine expects a response containing the new account balance. To send the response back to only Automated Teller 1 the GUID from the `<hbc_ThisMessageIDForRouteBack>` tag in the original message is placed in the tag `<hbc_RespondingToThisMessage>`. The `<My_GUID>` tag value echoed back in the response message so the Automated Teller machine can match this message to the original DEPOSIT_FUNDS_INTO_BANK_ANTIOCH message it sent earlier.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
  <hbc_Message>
    <hbc_MessageType>ACCOUNT_INFO_FROM_ANTIOCH</hbc_MessageType>
    <My_Amount>350</My_Amount>
    <My_GUID>jWskz7WFPk92jnFmXLpWWmRWJ25CgE9g</My_GUID>
    <hbc_RespondingToThisMessage>DwCSN3CcC2FwQsaDyXzZafgZqQwyLrtM</hbc_RespondingToThisMessage>
  </hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

The HummingBird receives and routes this message to Automated Teller 1. Notice the `<hbc_RespondingToThisMessage>` is stripped out of the message because it is no longer useful. Automated Teller 1 knows the message is a response to the original DEPOSIT_FUNDS_INTO_BANK_ANTIOCH because it matches the `<My_GUID>` value to the original.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>ACCOUNT_INFO_FROM_ANTIOCH</hbc_MessageType>
  <My_Amount>350</My_Amount>
  <My_GUID>jWskz7WFPk92jnFmXlpWwRWJ25CgE9g</My_GUID>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Routing

Routing is the term used to describe the HummingBird sending messages to all the clients that have subscribed to a message type. The term Routing should not be confused with the term routback message.

Transactions

A set of messages sent by a single client where each receiving client is blocked from receiving messages from other clients until a commit message is sent. To indicate the beginning of a transaction a begin work message must be sent to the HummingBird. The transaction ends when a commit message is sent or the time out period elapses.

Tracing

HummingBird activity can be monitored by a client by subscribing to pre-defined tracing messages produced by the HummingBird. These tracing messages contain information that is useful for debugging and obtaining the system wide status of the distributed application.

Example of a message sent by a client who wishes to receive a message when a transaction has timed out.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_SUBSCRIBE</hbc_MessageType>
  <hbc_SubscribeToThisMessageType>HBC_TRACE_BEGIN_WORK_TIMEOUT</hbc_SubscribeToThisMessageType>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

The client will now receive messages of message type HBC_TRACE_BEGIN_WORK_TIMEOUT, for example

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_TRACE_BEGIN_WORK_TIMEOUT</hbc_MessageType>
  <hbc_TraceServerTime>8/23/2009 6:22:48 AM</hbc_TraceServerTime>
  <hbc_TraceClientName>Money Transfer Service 3</hbc_TraceClientName>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Command Message Type

A command message is a [message](#) sent from a client to the HummingBird whose message

type matches a predefined HummingBird message type. The term “command message type” and “command message” can be used interchangeably. Unlike client messages, command messages are not [routed](#) to other clients.

There are five command messages supported by the HummingBird; SUBSCRIBE, UNSUBSCRIBE, BEGIN WORK, COMMIT, and LOGIN.

Commit Message Type

To indicate the end of a transaction (started by sending a begin work command message) a commit command message must be sent. Rollback messages are not currently supported.

This is an example of a commit command message sent from a client to the HummingBird.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_COMMIT</hbc_MessageType>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Begin Work Message Type

The HummingBird supports a basic form of [transactions](#). After a client sends a begin work command message to the HummingBird the clients gains exclusive access any client who gets routed its messages. In other words, once a client receives a message sent by a client who performed a begin work command it will only receive message from that client until the sending client performs a commit command. The message blocking is terminated after the client who originally sent the begin work command message sends a [commit](#) command message to the HummingBird or the timeout expires.

This is an example of a begin work command message sent from a client to the HummingBird.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_BEGINWORK</hbc_MessageType>
  <hbc_TimeOut>45000</hbc_TimeOut>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Login Message Type

After a [client](#) establishes a connection to the HummingBird using TCP/IP or a named pipe it has the option of sending a LOGIN command message. The purpose of the login is to establish a human friendly name which enables the HummingBird to provide better tracing (debugging) information and better information on routed messages.

An example of a LOGIN message sent by a client to the HummingBird.

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
```

```

<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_LOGIN</hbc_MessageType>
  <hbc_ClientName>My Logging Application</hbc_ClientName>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq

```

The HummingBird will now be able to include the client name (My Logging Application) when sending trace messages and when routing messages to other clients.

Subscribe Message Type

When a [client](#) wants to receive messages of a particular message type sent by other clients it must subscribe to that message type. The subscribe command is defined by using a `<hbc_MessageType>` XML element of (HBC_COMMAND_SUBSCRIBE) which is pre-defined by the HummingBird. The XML element `<hbc_SubscribeToThisMessageType>` tells the HummingBird what kind of message type the client wishes to receive.

Example of a subscribe command message sent by a client to the HummingBird. In this case the client that sent this message to the HummingBird wishes to receive any messages matching `Transfer Funds` that may be published by other clients.

```

MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>HBC_COMMAND_SUBSCRIBE</hbc_MessageType>
  <hbc_SubscribeToThisMessageType>Transfer Funds</hbc_SubscribeToThisMessageType>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq

```

After subscribing, the client will receive any [published](#) message with a `<hbc_MessageType>` of `Transfer Funds`. For example, the subscribing client will now receive this message which was created and sent to the HummingBird by another client. Note: The HummingBird adds some additional information by inserting it into the `<hbc_AddedByHummingBird>` tag.

```

MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
<hbc_Message>
  <hbc_MessageType>Transfer Funds</hbc_MessageType>
  <my_amount_in_US_dollars>100</my_amount_in_US_dollars>
  <hbc_AddedByHummingBird>
    <hbc_ThisMessageIDForRouteBack>DwCSN3CcC2FwQsaDyXzZafgZqQwyLrtM</hbc_ThisMessageIDForRouteBack>
    <hbc_ThisMessageWasCreatedBy>Accounting Server 2</hbc_ThisMessageWasCreatedBy>
  </hbc_AddedByHummingBird>
</hbc_Message>
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq

```

UN-Subscribe Message Type

Similar in construction to the SUBSCRIBE message type. When the HummingBird receives a message from a client with an `HBC_COMMAND_UNSUBSCRIBE` message type it no longer routes messages to the client for the type defined in the `<hbc_UnSubscribeFromThisMessageType>` XML element. It is not an error for a client to un-subscribe to a message type of which it never has subscribed.

Example of an UN-Subscribe command message sent by a client to the HummingBird. The

client will no longer receive messages with the client application specific message type of (LogWarnings).

```
MjAskrrpBWkD4zctG2TPPdAS3n6ySc4GE
```

```
<hbc_Message>
```

```
  <hbc_MessageType>HBC_COMMAND_UNSUBSCRIBE</hbc_MessageType>
```

```
  <hbc_UnSubscribeFromThisMessageType>LogWarnings</hbc_UnSubscribeFromThisMessageT  
ype>
```

```
</hbc_Message>
```

```
9zA8bzLDdRBFqRgmGZGEFAfqMfnb8KHq
```

Design Decisions

The required XML tags, element names, and envelopes are very verbose. This causes the messages to be longer than otherwise would be minimally needed. The purpose of this is to make it clear to new developers the functions being performed by the HummingBird.

Most server applications are designed to recover as best as possible from unexpected exception errors. Some exception errors in HummingBird are treated as fatal. A process is in place to log and report exception errors to a website.

Version 0.1 – TODO

An example of streaming .NET classes through the HummingBird should be created.

The source code for the HummingBird is compiled in Visual Studio 2005. However, the System.IO.Pipes from .NET 3.5 is being used for named pipes. The HummingBird should be converted to Visual Studio 2008 or 2010.

Support running the HummingBird under MONO on other operating systems.

http://www.mono-project.com/Main_Page

There is no project file compiling the HummingBird to compiled as a Windows Service.

There is no example of an application that compiles the HummingBird code internally inside its own EXE.

A better logging/tracing experience.

Any bug fixes.

Anything to increase the simplicity of using the HummingBird. Improving examples, simplifying the API in HummingBirdConnect, and simplifying the XML messages/parsing.

Other TODO ideas:

Even though there is 'sanity checking' on sent and received XML messages th HummingBird version does not make full amends for deliberate or accidentally malicious clients. There is no protection for DNS attacks. Password protected secure logins, Windows Authentication are not supported.

Once the HummingBird obtains a certain maturity it could be entirely or partially ported to c++ for speed purposes.

The HummingBird will throw exceptions on catastrophic errors and shut down. This typically won't occur but the rule for an enterprise level application would be that the HummingBird should never go down.

There is no login/password functionality for client connections. The Login function is currently the place holder for login functionality. Currently the Login function is only useful to simplify tracing information by replacing GUIDs with more human friendly names.

The HummingBird contains many hard coded XML tags, element names, and envelopes. An option to configure these to smaller strings to reduce the message sizes, therefore improving the speed, could be allowed. Currently, the XML tags are verbose to help with learning the system and debugging.

Messages are sent to and from the HummingBird through either TCP/IP or named pipes. These methods could be decoupled from the HummingBird and created as "adapters". For

instance, a client may want to send information to the HummingBird through batch files and create a batch file adapter.

Add the ability to run multiple connected HummingBirds on separate machines. The multiple HummingBirds would work in unison but be indistinguishable from a single running HummingBird in functionality.