

# EAQueue

## Document history:

Date	Version	Author	Comments	Approved
12/03/09	1.0	JC	Initial version	n/a

## Contents:

<a href="#">1. Introduction.....</a>	<a href="#">3</a>
<a href="#">2. Examples of EAQueue.....</a>	<a href="#">4</a>
<a href="#">2.1 Receipt of messages.....</a>	<a href="#">4</a>
<a href="#">2.2 Sending messages.....</a>	<a href="#">5</a>
<a href="#">3. Communication methods.....</a>	<a href="#">6</a>
<a href="#">2.1 UDP datagrams.....</a>	<a href="#">6</a>
<a href="#">2.2 Shared files.....</a>	<a href="#">6</a>
<a href="#">2.3 Message hub.....</a>	<a href="#">7</a>
<a href="#">4. The EAQueue DLL.....</a>	<a href="#">9</a>
<a href="#">4.1 Installation and overview.....</a>	<a href="#">9</a>
<a href="#">4.1.1 Installation of the EAQueue DLL.....</a>	<a href="#">9</a>
<a href="#">4.1.2 Message formats and sizes.....</a>	<a href="#">9</a>
<a href="#">4.1.3 Extra considerations in an EA's start() function.....</a>	<a href="#">10</a>
<a href="#">4.1.4 File locations on Vista and later versions of Windows.....</a>	<a href="#">10</a>
<a href="#">4.2 DLL functions.....</a>	<a href="#">10</a>
<a href="#">4.2.1 CreateClient().....</a>	<a href="#">11</a>
<a href="#">4.2.2 GetConnectionStatus().....</a>	<a href="#">12</a>
<a href="#">4.2.3 FreeClient().....</a>	<a href="#">12</a>
<a href="#">4.2.4 GetQueueMessage().....</a>	<a href="#">13</a>
<a href="#">4.2.5 CountPendingMessages().....</a>	<a href="#">13</a>
<a href="#">4.2.6 WipeThreadQueue().....</a>	<a href="#">13</a>
<a href="#">4.2.7 SendEAMessage().....</a>	<a href="#">14</a>
<a href="#">4.2.8 AsyncSendEAMessage().....</a>	<a href="#">15</a>

---

<a href="#">4.2.9 GetAsyncSendResult()</a> .....	15
<a href="#">4.2.10 ClearEAFileChannel()</a> .....	16
<a href="#">4.2.11 TimeUTC()</a> .....	16
<a href="#">4.3 Parameters for CommMode and Config</a> .....	16
<a href="#">4.3.1 CommMode</a> .....	16
<a href="#">4.3.2 Parameters for Config when using UDP datagrams</a> .....	17
<a href="#">4.3.3 Parameters for Config when using a shared file</a> .....	17
<a href="#">4.3.4 Parameters for Config when receiving messages via the EAQueue message hub</a> .....	18
<a href="#">4.3.5 Parameters for Config when sending messages via the EAQueue message hub</a> .....	18
<a href="#">5. The EAQueue message hub</a> .....	20
<a href="#">6. Sending and receiving from software other than EAs</a> .....	21

## 1. Introduction

EAQueue is a DLL allowing Metatrader EAs to send messages to each other – e.g. in order to implement a master-slave relationship where one EA generates signals and other EAs trade them. EAQueue caters for everything from ad hoc communication between EAs on a single computer through to broadcasting of commercial signals over the internet.

EAQueue can use a variety of communication methods. For simple messaging between EAs on the same computer or local network, it is usually easiest to send data via UDP datagrams or shared files. For communication over the internet, EAQueue can instead use full-blown TCP/IP communication via a message-hub application.

Crucially, the EAQueue library is multi-threaded and non-blocking: EAs do not suspend trading activity while waiting for messages. Any overheads related to file access or socket transmission happen asynchronously, without affecting the EA's execution.

Communication is also event-driven: new incoming messages trigger calls to the EA's start() function even if there are no market ticks. In other words, EAs can receive messages via EAQueue even at weekends when they would normally be in suspended animation because the markets are closed.

All communication using EAQueue is on a broadcast basis: a sender does not receive confirmation that a message has been picked up by a receiver (though you can build this yourself on top of EAQueue by having receivers send response messages).

Messages sent via EAQueue are simply plain text. The format of messages is then entirely up to you – e.g. comma-separated parameters. Messages can be trading signals, news, alerts etc.

## 2. Examples of EAQueue

The following examples provide a simple introduction to using EAQueue in a Metatrader EA. The code is explained in much greater detail later on, in the section about the [EAQueue DLL](#).

### 2.1 Receipt of messages

The following example shows an EA receiving messages via the file “c:\messages\GBPUSD signals.txt”.

```
#import "EAQueueDll.dll"
// Creates a worker thread. Called in init()
int CreateClient(int hWnd, int CommMode, string Config);

// Deletes the resources associated with the worker thread. Called in deinit()
void FreeClient(int CallerId);

// Gets the oldest message from the queue. Returns a blank string if the queue is empty
string GetQueueMessage(int CallerId);
#import

// Variable with global scope holding the caller ID assigned to this EA by CreateClient()
int glbCallerId;

// EA start routine. Get the DLL to create a worker thread for us
int init()
{
    int hMyWindow = WindowHandle(Symbol(), Period());
    glbCallerId = CreateClient(hMyWindow, 1, "c:\messages\GBPUSD signals.txt");
}

// EA termination. Stop the worker thread and free resources.
int deinit()
{
    FreeClient(glbCallerId);
}

// Called on each tick. Simply extracts all messages received since
// the last call to start() – bearing in mind that the DLL will force a call to
// start() each time a message is received, even if the market is closed
int start()
```

```
{
  // There are potentially multiple, queued messages since the last call to start().
  // Keep extracting messages until the queue is empty, indicated by a blank string.
  string strMsg = GetQueueMessage(glbCallerId);
  while (strMsg != "") {
    Alert("Got a message: " + strMsg);
    strMsg = GetQueueMessage(glbCallerId);
  }
}
```

## 2.2 Sending messages

The following example shows an EA (or script) sending a message, again via the file “c:\messages\GBPUSD signals.txt”.

```
#import "EAQueueDll.dll"
  // Sends a message
  int SendEAMessage(string Message, int CommMode, string Config);
#import

int start()
{
  int err = SendEAMessage("SELL,0.20,MARKET", 1, "c:\messages\GBPUSD signals.txt");
  if (err == 0)
  {
    // Successful
  } else {
    // Message transmission failed
  }
}
```

### 3. Communication methods

EAQueue can be configured to use three different methods of communication (via the parameters passed to the [CreateClient\(\)](#) and [SendEAMessage\(\)](#) functions):

- [UDP datagrams.](#)
- [Shared files.](#)
- [The EAQueue message hub application.](#)

For communication on a single computer or over a local network you will normally want to use shared files (or UDP datagrams, if message delivery is not critical). For communication over the internet, the message hub application provides a reliable, scalable way of handling multiple remote receivers.

#### 2.1 UDP datagrams

Messages are simply sent as UDP datagrams. A receiving EA listens on a UDP port number, and a sending EA sends messages to that port number. The maximum message size is 500 bytes.

**Pros:**

- Simplicity.

**Cons:**

- No logging (i.e. unless the sender/receiver maintain their own logs).
- Each receiver needs a unique ID (a UDP port number), and the sender needs to know this ID. Therefore, a single message cannot be broadcast to a group of receivers.
- Messages are lost if the receiver is not running at the time the message is sent. This includes an EA which has been unloaded while its properties dialog is displayed.
- In theory, delivery of UDP datagrams is not guaranteed (though it is normally 100% reliable on a local network or single computer).
- Not suitable for communication over the internet.

#### 2.2 Shared files

A sending EA writes messages to a file on disk. Receiving EAs simply spot the new additions to the file. The file provides an implicit log of all the messages which have been sent. The file can be on a

shared network directory, allowing communication between different computers. The maximum message size is 10,000 bytes.

**Pros:**

- Simplicity.
- Built-in logging (i.e. the shared-file's contents).
- Senders can broadcast messages to multiple receivers – i.e. multiple EAs can be “listening” to the same file.
- Caters for any number of different communication “channels” – i.e. multiple different files on disk.
- EAs receive messages which were sent before they started running, or during temporary closures.

**Cons:**

- Files must be periodically cleared out, using [ClearEAFileQueue\(\)](#), to prevent EAs receiving large numbers of very old messages whenever they start up.
- You will normally want to include a send-time in messages, and have EAs check this to prevent them acting on outdated information.
- Not suitable for communication over the internet.

### 2.3 Message hub

Messages are sent and received via the EAQueue message-hub application. A sender broadcasts messages to a specific “channel”. Receivers sign up to receive messages via that channel. The maximum message size is 10,000 bytes.

**Pros:**

- Senders can broadcast a message to multiple receivers.
- EAs can receive messages which were sent before they started running, or during temporary closures.
- Caters for any number of different communication “channels” – senders and receivers specify a channel name they want to use when calling [CreateClient\(\)](#) or [SendEAMessage\(\)](#).
- Optional logging by the hub.
- Optional security – i.e. passwords required both for sending and receiving messages.
- Suitable for use over the internet.

**Cons:**

- The sender needs to run the separate message hub application in addition to MT4.
- You will normally want to include a send-time in messages, and have EAs check this to prevent them acting on outdated information.

## 4. The EAQueue DLL

Metatrader EAs send and listen for messages using the EAQueue DLL. Messages can be sent either synchronously or asynchronously. The DLL always receives messages asynchronously, in a separate thread to the EA, and stores pending messages in a queue which the EA then queries when its start() function is next run.

In addition, the DLL *forces* calls to an EA's start() function when a new message is received. This ensures that an EA receives a message even if there are no market ticks and its start() function would not normally run.

### 4.1 Installation and overview

#### 4.1.1 Installation of the EAQueue DLL

For an EA to send or receive messages, the EAQueue DLL (EAQueueDll.dll) must be put either in the experts\libraries directory of an MT4 installation, or in the system directory for the computer. Please note that MT4 is 32-bit software. On a 64-bit version of Windows, the system directory for 32-bit applications is typically \Windows\SysWOW64, not \Windows\System32.

In addition, the "Allow DLL imports" option in MT4 must be turned on, either for each individual EA which uses EAQueue, or as a global setting via Tools/Options/Expert Advisors.

#### 4.1.2 Message formats and sizes

Messages via EAQueue are plain text (no ASCII character values under 32) but can otherwise be in any format you like. The maximum message size is 10,000 bytes when sending via a shared file or the message hub application, or 500 bytes when sending via UDP datagrams.

You will typically want to include a send-time in messages so that receiving EAs can ignore out-dated information. To help with this, the EAQueue DLL provides a [TimeUTC\(\)](#) function which returns UTC time rather than local time, avoiding potential problems with sending messages between computers in different time zones.

For example, the following MQL code builds a comma-delimited message consisting of the current time, a buy instruction, and the current ask price. An EA receiving this message would typically check the time and price, and not act on it if the message was older than a defined threshold or the ask price at the receiver's end was substantially different from that reported by the sending EA.

```
string strMsg = TimeUTC() + ",BUY," + MarketInfo(Symbol(), MODE_ASK);  
SendEAMessage(strMsg, 1, "c:\\messages\\" + Symbol() + " messages.txt");
```

N.B. The need to check a timestamp in messages applies particularly when communicating via a [shared file](#). Messages are stored in the file permanently, until its contents are cleared using [ClearEAFileChannel\(\)](#). Therefore, when an EA first starts up it may receive a large number of very old messages contained in the file.

#### 4.1.3 Extra considerations in an EA's start() function

The EAQueue DLL forces a call to an EA's start() function whenever a new message is received. Therefore, start() can get called even at weekends when the market is closed.

You may need to make some slight modifications to your EA's behaviour to stop it trying to trade during market closures – e.g. additional calls to MQL functions such as IsConnected().

#### 4.1.4 File locations on Vista and later versions of Windows

If you are sending and receiving via [shared files](#), you may need to bear in mind the “file virtualization” in Vista and later versions of Windows.

For example, if you try to communicate via a file inside the Program Files area, Vista may transparently redirect the file operation to another area of disk. Files may not be stored where you expect, and senders and receivers can potentially end up reading and writing from different files.

In short: either turn off file-virtualization (by turning off UAC, or by lowering the security on the folder), or use a directory which is not potentially covered by virtualization.

## 4.2 DLL functions

The EAQueue DLL provides functions both for sending and receiving messages. The commonly used functions for receiving messages are as follows:

- [CreateClient\(\)](#)
- [GetConnectionStatus\(\)](#)
- [FreeClient\(\)](#)
- [GetQueueMessage\(\)](#)

The EA also provides receivers with the following functions, but these do not typically need to be used:

- [CountPendingMessages\(\)](#)
- [WipeThreadQueue\(\)](#)

There are functions for sending messages synchronously and asynchronously, plus a utility function for resetting the contents of a shared file:

- [SendEAMessage\(\)](#)
- [AsyncSendEAMessage\(\)](#)
- [GetAsyncSendResult\(\)](#)
- [ClearEAFileChannel\(\)](#)

Finally, the DLL also provides a service function to help callers send time-stamps in messages in a way which is not dependent on both sender and receiver being in the same time zone:

- [TimeUTC\(\)](#)

#### 4.2.1 CreateClient()

The CreateClient() function is normally called in an EA's init(). It tells the DLL to create a worker thread for the EA so that it can receive messages asynchronously.

```
int CreateClient(int hWnd, int CommMode, string Config);
```

The return value is a "caller ID" for the EA which must then be specified when subsequently using [FreeClient\(\)](#), [GetQueueMessage\(\)](#) etc. This caller ID is normally stored in a variable with global scope, accessible throughout the EA's lifetime – see the "glbCallerId" variable in the [example above](#). An EA can make multiple calls to CreateClient(), and store multiple caller IDs, allowing it to listen on multiple "channels".

**N.B.** Calls to CreateClient() cannot fail – unless the computer is critically low on resources – and there is no error return code from the function. Connection to a TCP/IP port or shared file is carried out asynchronously by the worker thread. An EA can subsequently check whether the connection succeeded using the [GetConnectionStatus\(\)](#) function.

The hWnd for CreateClient() parameter is a handle to the EA's chart window so that the DLL can force calls to its start() function when new messages are received. An EA can obtain this handle using

MT4's WindowHandle() function as in the [example above](#). Alternatively, you can pass zero for this parameter if you do not want your EA to be woken by incoming messages.

The CommMode and Config parameters tell the DLL which communication method the EA wants to use, and what parameters to set it up with. These are [explained below](#). For example, to have an EA listen for messages in a shared file, CommMode is set to 1 and Config is simply the filename to watch.

#### 4.2.2 GetConnectionStatus()

The GetConnectionStatus() function checks whether the DLL is currently receiving messages for an EA.

```
int GetConnectionStatus(int CallerId);
```

There are four possible return values from GetConnectionStatus():

Value	Meaning
-1	The DLL's worker thread for the EA is still starting up. Connection is underway.
0	Connection to the communication channel has failed permanently. An error message will be displayed on screen. When communicating via a shared file, it means that the file does not exist and cannot be created. When communicating via datagrams or the message hub, it means that the specified port is already in use or the Config parameters are not valid. If such a fatal error message happens, the EAQueue DLL will display a message on screen.
1	Connection has succeeded.
2	Connection is currently failing, but the DLL will keep retrying. This only applies to communication via the message hub – it indicates, for example, that your internet connection has gone down and the hub cannot currently be contacted. However, it can also indicate that the hub is accepting and then immediately closing the connection from the EA for security reasons, e.g. because the user name and password are wrong.

#### 4.2.3 FreeClient()

The FreeClient() function is normally called in an EA's deinit(). It releases all the resources previously created by a call to [CreateClient\(\)](#).

```
void FreeClient(int CallerId);
```

The parameter to FreeClient() is the "caller ID" allocated by the return value from CreateClient().

#### 4.2.4 GetQueueMessage()

The GetQueueMessage() function checks to see if a new message has been received by the DLL.

```
string GetQueueMessage(int CallerId);
```

The parameter to the function is the “caller ID” allocated by [CreateClient\(\)](#). The return value is a string which either contains a message, or is blank.

All incoming messages are stored by the DLL in a queue, and multiple messages may arrive between calls to the EA’s start() function. Therefore, in its start() function an EA typically keeps calling GetQueueMessage() in a loop, extracting and processing all pending messages, until GetQueueMessage() returns a blank string indicating that there are no more messages in the queue. This is demonstrated by the [example above](#).

Messages can be in any textual format of your choice – e.g. comma-separated values. It is common to include a send-time in a message so that receiving EAs can ignore signals which are significantly out of date.

#### 4.2.5 CountPendingMessages()

The CountPendingMessages() function counts the number of messages sitting in the DLL’s queue and awaiting collection by the EA.

```
int CountPendingMessages(int CallerId);
```

The parameter to the function is the “caller ID” allocated by [CreateClient\(\)](#). The return value is the number of messages which in the queue – i.e. the number of times [GetQueueMessage\(\)](#) can be called before it will return a blank string.

#### 4.2.6 WipeThreadQueue()

The WipeThreadQueue() function tells the DLL to discard any messages which are awaiting collection by the EA.

```
void WipeThreadQueue(int CallerId);
```

The parameter to the function is the “caller ID” allocated by [CreateClient\(\)](#). After the call to WipeThreadQueue(), the [CountPendingMessages\(\)](#) function will return zero and [GetQueueMessage\(\)](#) will return a blank string.

#### 4.2.7 SendEAMessage()

The SendEAMessage() function sends a message to a broadcast channel.

```
int SendEAMessage(string Message, int CommMode, string Config);
```

The parameters are the textual message to send, and the communication method to use and its configuration. For example, to send a message to a shared file, CommMode is set to 1 and Config is simply set to the file name.

The return value from SendEAMessage() is zero if successful, or an error code if there is a problem. Please note that all communication via EAQueue is on a broadcast basis: successful sending of a message does not imply that any EA has received it.

Transmission using SendEAMessage() is synchronous, and the EA's execution is halted while sending takes place. To send messages asynchronously, without blocking, use [AsyncSendEAMessage\(\)](#).

Possible error return codes from SendEAMessage() are as follows:

Value	Meaning
<b>General errors</b>	
2	Message is too long – more than 10,000 bytes for shared-files or the hub, or 500 bytes for UDP datagrams.
3	Message contains non-textual characters – i.e. ASCII values below 32.
<b>Shared-file errors (CommMode = 1)</b>	
10	Unable to open the specified file for writing.
11	File opened, but write failed.
<b>UDP datagram errors (CommMode = 0)</b>	
20	Winsock initialization failed
21	Unable to create a datagram socket
22	Unable to send a datagram (commonly because Config contains an address such as www.bob.com:22222 and www.bob.com is not valid)
<b>Message hub errors (CommMode = 2)</b>	
30	Invalid number of parameters – SendEAMessage() requires six comma-separated values for Config when sending via the hub
31	Winsock initialization failed
32	Unable to create a TCP/IP socket
33	Unable to connect to the specified sending port on the hub server
34	Timeout while waiting for the hub to accept new messages
35	Error while sending data to the hub
36	Timeout while waiting for a response from the hub
37	No data received from the hub – usually indicates that the hub has closed the

	connection without responding, for security reasons. In other words: check the user name and password being sent.
38	Hub has declined the message. Check the channel name, and check that the message is valid.
39	Corrupt response data received from the hub

#### 4.2.8 AsyncSendEAMessage()

AsyncSendEAMessage() is identical to [SendEAMessage\(\)](#), except that the message is sent asynchronously and the EA's execution does not block if there is a delay in transmission.

```
int AsyncSendEAMessage(string Message, int CommMode, string Config);
```

The parameters are the same as for SendEAMessage() – message, mode, and configuration. However, the return value is different: it is an ID for the asynchronous transmission. The success or failure of the asynchronous operation can then be checked using [GetAsyncSendResult\(\)](#).

For example:

```
int IAsyncId = AsyncSendEAMessage("My message", 0, "20000");
[... EA continues immediately ...]
int IResult = GetAsyncSendResult(IAsyncId, false);
```

N.B. AsyncSendEAMessage() allocates memory which is not freed until GetAsyncSendResult() is called. However, the amount of memory is small. If you don't care about the success/failure of an asynchronous send, you may decide not to worry about calling GetAsyncSendResult()...

#### 4.2.9 GetAsyncSendResult()

The GetAsyncSendResult() function checks the progress of an asynchronous message transmission which was previously started using [AsyncSendEAMessage\(\)](#).

```
int GetAsyncSendResult(int AsyncId, bool FreeMemoryIfComplete);
```

The return value from GetAsyncSendResult() is -1 if the transmission is still underway. Otherwise, if complete, the return value is the same as for [SendEAMessage\(\)](#): zero if successful, or one of the error codes listed above.

The AsyncId parameter is the ID for the asynchronous transmission which was returned by AsyncSendEAMessage().

The `FreeMemoryIfComplete` parameter can be used to instruct the DLL to free the memory allocated for the transmission if it has finished (and therefore if the return value from the function is  $\geq 0$ ). If you request `FreeMemoryIfComplete` and the transmission has indeed finished, then any further calls to `GetAsyncSendResult()` with the same `AsyncId` will cause a crash – because the memory indicated by `AsyncId` no longer exists.

#### 4.2.10 `ClearEAFileChannel()`

The `ClearEAFileChannel()` function clears the contents of a shared file being used for message transmission.

```
int ClearEAFileChannel(string Filename);
```

The parameter is simply the filename to clear – e.g. “c:\\messages\\GBPUSD signals.txt” (bearing in mind that backslashes in MQL code need to be escaped).

The return value is zero if successful, or 1 if the file could not be written to in order to clear it.

#### 4.2.11 `TimeUTC()`

`TimeUTC()` is a service function for returning the current UTC/GMT time. It is equivalent to MT4’s built-in `TimeLocal()` function except that it returns UTC/GMT time rather than local time:

```
datetime TimeUTC();
```

### 4.3 Parameters for `CommMode` and `Config`

The following section describes the values for the `CommMode` and `Config` parameters for the [CreateClient\(\)](#) and [SendEAMessage\(\)](#) functions.

#### 4.3.1 `CommMode`

The `CommMode` parameter for [CreateClient\(\)](#) and [SendEAMessage\(\)](#) determines whether send/receipt is via UDP datagrams, a shared file, or the EAQueue message hub:

<b>CommMode</b>	<b>Meaning</b>
0	UDP datagrams
1	Shared file
2	Message hub

### 4.3.2 Parameters for Config when using UDP datagrams

When communicating via UDP datagrams ([CommMode](#) = 0), the Config parameter holds the UDP port number to send to or receive from. For example, an EA can listen for messages on UDP port 12345 using the following call:

```
glbCallerId = CreateClient(hMyWindow, 0, "12345");
```

Another EA then sends messages to that port number using the following command:

```
int err = SendEAMessage(strMyMessage, 0, "12345");
```

If the sender is on a different computer to the receiver, then the sender can specify the receiver's IP address using the usual server:port syntax:

```
int err = SendEAMessage(strMyMessage, 0, "192.168.9.238:12345");
```

Please note that delivery of UDP datagrams is not guaranteed, particularly between different computers rather than on a single computer. In addition, a receiving EA must be running at the time a message is sent. If the receiver is currently closed – including having been unloaded to display/change its properties – the message will be permanently lost.

### 4.3.3 Parameters for Config when using a shared file

When communicating via a shared file ([CommMode](#) = 1), the Config parameter simply specifies the file to use. The file is always created if it does not already exist – a receiver of messages can connect to a file before any messages are sent to it.

The only other point to note is that the location of the file – i.e. drive letter and so on – may be different for sender and receiver if the file is on a network drive. For example, a receiver may monitor the local file "c:\messages\GBPUSD signals.txt":

```
glbCallerId = CreateClient(hMyWindow, 1, "c:\\messages\\GBPUSD signals.txt");
```

If the sender is on another computer, and accessing the "messages" directory via the mapped drive J:, then the parameters for SendEAMessage() would be as follows:

```
int err = SendEAMessage(strMyMessage, 1, "j:\\GBPUSD signals.txt");
```

Message delivery is quickest if the file is on the same computer as the receiver. Therefore, if the sender and receiver are on different computers, and there is only a single receiver of messages rather than a group, you should use a file on the receiver's computer, not the sender's.

N.B. Backslashes in file names in MQL must be escaped – i.e. double \\ rather than single \. Otherwise, MQL interprets them as an escape sequence such as \t = tab.

#### 4.3.4 Parameters for Config when receiving messages via the EAQueue message hub

When listening to messages via the EAQueue message hub, the Config parameter for [CreateClient\(\)](#) needs to contain six comma-separated values:

1. Server name or IP address of hub
2. Listening port on hub
3. Channel name
4. Receive old messages
5. User name
6. Password

The “receive old messages” parameter indicates whether the EA is interested in receiving old messages which were sent before the EA started, and is expressed as either zero or one.

For example, the following call connects to port 14444 on the local computer, listening for new messages on the channel called “News”. The EA is asking to receive any old messages which have already been transmitted.

```
glbCallerId = CreateClient(hWnd, 2, "localhost,14444,News,1,myusername,mypass");
```

#### 4.3.5 Parameters for Config when sending messages via the EAQueue message hub

When sending messages via the EAQueue message hub, the Config parameter for [SendEAMessage\(\)](#) needs to contain six comma-separated values – slightly different to the [parameters for receiving messages](#) via the hub.

The parameters are:

- Server name or IP address of the hub
- Port number for sending
- Channel name

- Storage mode
- User name
- Password

The “storage mode” parameter indicates whether or not the message should be stored and subsequently transmitted to clients who connect to the hub later. A value of 1 means “yes”; a value of zero means “no”.

For example, the following call connects to port 14443 on the local computer, and sends the message on a channel called “News”. The message is not stored; any EAs which connect to the hub in future will not receive it when they start up.

```
int err = SendEAMessage("Test!", 2, "localhost,14443,News,0,senduser,sendpass");
```

## 5. The EAQueue message hub

The EAQueue message hub is simply an application which listens on two TCP/IP ports: a sending port, and a receiving port. The hub then implements an unlimited number of virtual “channels”. When a receiver connects, it specifies the channel it is interested in (via the Config parameter to the [CreateClient\(\)](#) function). Similarly, when a sender transmits a message, it specifies which channel to broadcast it on (via the Config parameter to the [SendEAMessage\(\)](#) function). The hub then repeats the sent message out to all the receivers who have signed up for that channel.

Senders can specify whether a message should be retained and sent out to new receivers when they connect in future, or whether a message is disposable and only applies to receivers which are currently running. In addition, receivers can override this setting and choose to ignore any retained messages which were sent in the past. (In both cases these behaviours are specified by the Config parameter to the [CreateClient\(\)](#) and [SendEAMessage\(\)](#) functions.)

The hub can be supplied in two forms:

- As a .NET assembly which you then incorporate in your own wrapper application. The assembly provides the underlying mechanics of communication. The wrapper which you build is responsible for handling logging and user authentication, by responding to events raised by the assembly.
- As a complete, compiled executable. You specify the logging and authentication you want to use, and we build a wrapper application for you on a bespoke basis.

## 6. Sending and receiving from software other than EAs

The EAQueue DLL can potentially be used from *any* software with the ability to call DLLs, not just from EAs. In other words, you can send and receive messages from custom software written in almost any programming language.

The definitions of functions simply need to be translated from the MQL format [used in the documentation above](#). For example, the CreateClient() function can be declared and called from VB.NET as follows:

```
Private Declare Function CreateClient Lib "EAQueueDll.dll" (ByVal hWnd As Integer, ByVal  
CommMode As Integer, ByVal Config As System.Text.StringBuilder) As Integer  
[...]  
glbCaller = CreateClient(Me.Handle, 2, New StringBuilder("localhost,14444,News,1,,"))
```

The same code in VB6 would be as follows (bearing in mind that 32-bit integers are type Long in VB6, not type Integer):

```
Private Declare Function CreateClient Lib "EAQueueDll.dll" (ByVal hWnd As Long, ByVal  
CommMode As Long, ByVal Config As String) As Long  
[...]  
glbCaller = CreateClient(Me.hWnd, 2, "localhost,14444,News,0,,")
```

The only potential complication is receiving messages. The [GetQueueMessage\(\)](#) function passes back a C-style null-terminated string in a block of memory which the caller is responsible for freeing. Languages such as VB6 or VBA cannot treat this return value as a String. Instead, you have to allocate a buffer, and copy from the string returned by EAQueue into your buffer. For example, VB6/VBA code should retrieve messages using this VB6\_GetQueueMessage() rather than GetQueueMessage(). Please note that the return value from the imported GetQueueMessage() function is declared as Long, not as String.

‘ N.B. Return value from GetQueueMessage is declared as Long, not as String

```
Private Declare Function GetQueueMessage Lib "EAQueueDll.dll" (ByVal CallerId As Long) As  
Long
```

```
Private Declare Function lstrlen Lib "kernel32" Alias "lstrlenA" (ByVal lpString As Long) As Long  
Private Declare Function LocalFree Lib "kernel32" (ByVal hMem As Long) As Long  
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" (ByVal StringBuffer  
As String, ByVal Source As Long, ByVal Length As Long)
```

```
Private Function VB6_GetQueueMessage(ByVal CallerId As Long) As String  
Dim pString As Long, lenString As Long  
pString = GetQueueMessage(CallerId)
```

```
lenString = strlen(pString)
If lenString = 0 Then
    VB6_GetQueueMessage = ""
Else
    Dim strBuffer As String
    strBuffer = Space$(lenString + 1)
    CopyMemory strBuffer, pString, lenString
    VB6_GetQueueMessage = Left(strBuffer, lenString)
End If
LocalFree pString
End Function
```