

Advice for writing a custom OPC DA client in Visual Basic.

With particular reference to Rockwell Automation OPC servers (RSlinx Classic & FactoryTalk Gateway) communicating to Logix Controllers.

V1.02



Allen-Bradley • Rockwell Software

Rockwell
Automation

Version History

Version	Date	Description
0.0	15 th March 2012	Draft – For peer review.
0.1	13 th September 2012	Additions and peer review.
1.0	26 th September 2012	First Release
1.01	25 th October 2012	Syntax fix
1.02	14 th October 2013	Review for KB release.

Contents:

About this document	4
Assumed prior knowledge	4
Not included	4
Hello, is there anybody there? (Server Connections).	5
Just Browsing (OPCBrowse Object).....	6
Group together (Group(s) Object).....	7
Properties:	7
Considerations:.....	7
General Advice:	8
All Dressed Up And Nowhere To Go:.....	9
Creator and Destroyer:	9
Shopping List (OPCItem Object).....	10
Adding Items.....	10
Logix Says To Optimize	10
Get a Handle On The Items.....	11
It's Good To Talk.....	12
One At A Time:.....	12
Would You Like To Subscribe?.....	12
Keep in Sync	14
What do you Async about.....	15
Don't Hang Around:	17
Try and Try Again:	17
Clients Log:.....	17

About this document

The intention of this document is to give practical advice for writing OPC DA clients in Visual Basic 6 & Visual Basic .NET, particularly when used with Rockwell Automation Products; RSLinx Classic and FactoryTalk Gateway communicating with ControlLogix controllers. It is assumed the reader is at a point where they can write the client code and are looking for additional advice.

It is not intended to be a definitive guide; it is to provide general guidance based on experience gained whilst writing and troubleshooting clients.

Reference is made to the Rockwell Automation products: RSLinx Classic and FactoryTalk Gateway when used with Rockwell Automation's Logix controllers. The tag based addressing system in Logix controllers provide complete freedom for the Logix programmer and this white paper provides techniques for obtaining better communication performance when writing OPC DA clients.

The symbol  will be used to mark advice points for specific recommendations

Assumed prior knowledge

This document is written with the assumption that the reader has the following prior knowledge:

- Familiarity with the OPC Foundation specification: 'Data Access Automation Interface Standard', available from www.opcfoundation.org
- Understand how to create robust and reliable Visual Basic code for industrial applications, including a full implementation for handling exceptions, trapping errors, responding appropriately to error conditions.
- Familiar with Rockwell Automation RSLinx Classic or/and FactoryTalk Gateway.
- Understanding of Rockwell Automation's Logix controllers tag based symbolic addressing.


Not included

- The document does not include specific code for implementing a Visual Basic client. Sample code can be found in the Rockwell Automation Knowledgebase and during installation of RSLinx Classic (if the install option is selected).
- References to the OPC custom Interface for C programmers is not covered, however most of the techniques discussed still apply.

Hello, is there anybody there? (Server Connections).

When connecting to a server it is possible to do so with a subscription to events the server can raise. For example if the server is going to shut down it can inform subscribed clients of the pending action. However if the server is shutdown in an uncontrolled manner, for example loss of power to server, or network disruption it will not be possible for the server to raise the server events to the subscribed clients.

If the client is in active polling communication with a server, for example the client is using synchronised reads/writes a timeout will occur whilst waiting for the procedure call to complete. However if the client is using subscriptions to data change events it is perfectly possible that the client will not be informed of any server disconnect until a DCOM time out occurs (6 minutes, or more typically 2 x DCOM time out). The OPC specification does not specify an item value data change must occur in a specific time.

 Make a heartbeat for your client to detect if the server has been disconnected without having chance to raise an event. This can be achieved in several ways; the easiest of which is to have the client subscribe to a data point in the controller that is changing regularly. Use an internal timer, if the client has not seen that data point change in the allotted time, it is an indication of a disconnected server.

Alternately if the client is using asynchronous read/write it is advisable to monitor the transaction ID's for read/write not raising complete events in a specified timeout time. More detail on that in section [*Aysnc it is the best method*].

Just Browsing (OPCBrowse Object)

Of all the functionality provided by OPC servers it is the browse object that seems to cause most confusion when writing a client. It is of course not necessary to implement the browse object if item names are known already (ie via an export/import method). However the browse object is invaluable for getting the structure, items and properties of the name space in the end devices.

One key thing to realise about the OPC browse object is that it provides no visual interface. The GUI (graphical user interface) that is shown in clients have been built by the client developer; populating the content of the GUI by requesting data from the browse object and navigating the browse object through the end device's name space.

A typical way of doing this is to start at the root of the browse object (MoveToRoot) and then to request branches (ShowBranches) and leafs (ShowLeafs). Using the move methods it is then possible to navigate deeper into the namespace structure.

In basic terms, branches are structure and leafs are items or tags in the end device. An object that lends itself very well to this principle of displaying data is the Microsoft control TreeView ([http://msdn.microsoft.com/en-us/library/ms172635\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ms172635(v=vs.80).aspx)). This is one of the reasons it is seen extensively in OPC clients that implement the browser. One note of warning – the TreeView control for VB6 has a limit of 32768 nodes before it does not represent the data reliably; also it becomes very slow at the higher node counts.

☞ Care has to be taken when navigating the browse object, the repetitive nature of Showing Branches, Showing Leafs, Showing Branches, Showing Leafs..... Should not be called from inside a For Each loop – see OPC specification for why not. If it is the intention to browse the whole namespace without visualizing it to a user for navigation the best way to handle it is with recursive calls – each new branch making a recursive call to the next lower level of branches and leafs.

Apply filters for specific properties or names before browsing the namespace. If there are any changes the whole namespace needs to be browsed again from the root.

☞ For RSLinx Classic & FactoryTalk Gateway using a flat hierarchy in the filter will not return the entire name space in a flat format, it will only return the upper level of the namespace.

Group together (Group(s) Object).

The use of groups is the first important decision when designing an OPC client. The second is the choice of communication method as we shall see later.

A group is the mechanism for grouping tags together for communications. Tags are added into one or more groups before communicating with tags.

The most common questions with respect to groups is; how many should I have? Is 1 enough, is 100 too many? The answer to these questions is provided by the functionality and use that the OPC client will provide to the user. In considering how many groups to use, there are two very important properties that must be considered, along with some considerations.

☞ A word of caution here: If the group count is to be very large (for example more than a few hundred) give serious attention to why this is being done, could it be achieved with fewer groups— some advice below. In addition it can make diagnostics much more difficult when trying to fault- find a system and performance can be affected as many more events can be raised when using subscriptions (as the subscriptions are raised from the groups).

For example: consider 100 groups with 1 tag in each, compared against 1 group with 100 tags. The 100 tags changing value in a group update will raise 100 events compared to 1 event for the later example.

Properties:

1. A group is configured with the read update rate for any subscribed tags it contains.
2. A group can be active or inactive, this in effect enables or disables tag communication when subscription based communication is going to be used.

Considerations:

1. A group does not need to be active to be used for writing/reading item values. The group only needs to be active when subscribed item reads are being used.

☞ Don't make a group active if the tags (items) in it are not subscribed. This would put items on scan (ie actively polling for information at the group rate) – but then the item values will not be passed to any clients. This results in a waste of bandwidth and processing power.

2. To reduce loading on the OPC servers and end devices - group items together that need updating at a particular rate. This maybe as simple as; Fast, Medium and Slow.

3. Does the client application have specific functionality that makes sense to be separated? For example if the client is doing: data logging, displaying pages of data and reading/writing recipe values. It may make sense to have individual groups to separate out the client actions. This can assist in writing the client by easily allowing switching on/off communications and help in diagnosis when monitoring communications. However remember the caution above on having very large numbers of groups.

These properties and considerations have to be combined together when making a sensible choice of number of groups to use in the client application.

General Advice:

1. Identify tags that will be used for: reading only or both reading and writing.
2. Tags for reading only: Further separate this group into smaller groups based on:
 - a. Tags that are to be polled (subscribed) at the same rate (ie the fast, mid, slow etc).
 - b. Further split the above if there are significant client functional reasons at the different rates.
 - c. Tags that will be used with sync/async reads only. Should be in separate groups see point 3 below.
 - d. Make different groups for the tags at the different polled (subscribed) rates. If significant tags are in the further functional split then create different groups for them. These groups should be active and subscribed. Avoid making excessive multiple groups at the same subscribed rate.
3. Tags for reading/writing: As a rough guide –

Tags for reading and writing (that are not using subscribed for reading): As a rough guide – if the number of items is less than a few thousand make one group to hold items for writing only. If more than a few thousand consider the following:

- Identify any client functional splits (ie recipe use, commands, etc).
- Count tags to be used for each function.

Make more than one group – with a common sense rough guide (not a technical one) that a group not to have less than 5% of the total tags.

For these tags (sync/async reads/writes) create a separate group or multiple groups if there are significant numbers or functional splits. These groups should be inactive and unsubscribed.

It is not good practice to mix the communication methods described below in individual groups, for example subscribing to a tag and also reading the tag with a read method. The last value received by a client by which ever means is considered the last value sent to the client.

All Dressed Up And Nowhere To Go:

- Never make a group active and unsubscribed. This causes data to be polled and thrown away as no clients will receive updates.
- Care should be taken in the client application that error trapping is implemented correctly and not forgetting to tidy up the connections to the OPC server. For example if a client has 20,000 tags on subscription and an error it is important not to abandon the OPC server connections as the 20,000 tags will remain on scan with no client to connect to. The client restarting or recovering incorrectly from the error trapping routine will not reconnect to the existing groups but make new groups and tags.

Creator and Destroyer:

The groups should be created on client start-up after successful connection and removed as the client is shutdown. Creating and destroying groups 'on the fly' for frequent communications can make troubleshooting difficult and if done frequently enough can introduce delays. This is also true for adding and removing tags.

If the communication rate with the tags is very low, then consider making a group with a slow update rate eg 10 seconds or more. Then if the tag values are needed rapidly use an Async refresh on the group to force immediate update.

Communication methods are covered in the section ('It's good to talk')

Shopping List (OPCItem Object).

Once groups have been created, items need to be added before communication with the items can start. There are two key considerations here:

Adding Items

Items can be added either individually (AddItem) to an OPC group or added in a collection (AddItems). It is more efficient to use the AddItems method – this also gives RSLinx Classic and FactoryTalk Gateway all the tags at once and hence more efficient optimised packets can be built when communicating with Logix controllers.

Logix Says To Optimize

Adding items into a group initiates a 'dialogue' with the controller to produce what are called ControlLogix Optimised Packets. This proprietary algorithm working between the controller and RSLinx ensures that the minimum numbers of optimised packets are created for the tags requested. The creation of these packets uses memory in the controller to store the packet information. This can be seen by comparing the free memory in the controller with no communications and with communications. Sufficient memory needs to be available to enable efficient communications.

Using optimised packets delivers the very fastest communications available with a data server talking to a Logix controller. However there is something important to be aware of. The initial creation of the packets takes time. This is not usually an issue it is simply a short delay before receiving the first data value update. However if your client repeatedly adds and deletes items the optimised packets will be constantly created and destroyed. This could lead to believing the data server is not performing at a high level. Hence the recommendation is not to repeatedly add and remove tags constantly. Rather leave them added until no longer required, i.e. at client close down

Get a Handle On The Items

An item has two handles associated with it, Client & Server. The client handle is allocated by you when adding the item. The server handle is allocated by the OPC server when an item is added. These handles are important as they are used for communication with the OPC server, it is not the item name (ItemID)(Tag Name) that is used.

When adding items (tags) to groups, request the server handles immediately. This eliminates the need to repeatedly request server handles when reading & writing items. Keep an association between the ItemID (tag name), Client handle and Server handle.

The OPC server allocates a Server handle but you can choose a client handle as your client adds the items. This could be a simple incrementing number; however it is good practice to make the client handle work for you as it is the client handles that are returned from the OPC server for subscribed items (see next section on item subscriptions).

Example of client handle use:

The client handle is a long (signed) (2^{32}) how does this help? It means you have a large number range to use.

Let's say for example your application will display tag values on a number of separate displays (e.g. 20). You decide each display can hold 180 tag values in a grid arrangement of 30 x 6. Hence total items to display = $20 * 30 * 6 = 3600$ items.

You could just allocate client handles of 1 to 3600. Then when RSLinx Classic or FactoryTalk Gateway informs your application that a tag value has changed, say client handle 2061 (note: no item name, just client handle). Your application will then have to look up where that tag value is located, display and position.

Or you could allocate client handles that identified the display number and the grid location of the tag value directly. For example -

Client handle = (display number * 2^{16}) OR (row * 2^8) OR (column) *(the OR is a logical OR).*

So here the client handle inherently represents the display, row and column.


It's Good To Talk

At this point we have connection to the server, groups created and items (tags) added to them. The next decision is what method to use to read and write values to the items. To help you decide, this section will describe the options available including the pros and cons of each method.

It is not good practice to mix the communication methods described below in individual groups, for example subscribing to a tag and also reading the tag with a read method. The last value received by a client by which ever means is considered the last value sent to the client.

One At A Time:

It is possible to read and write individual items by using the OPCItem object. For each read/write a blocking call is made to the OPC server. This is a very inefficient method for communications with a large number of tags and should really only be used for small item quantities and then only infrequently.

 This is only available via the Automation Interface, not the custom interface.

Would You Like To Subscribe?

Communication by subscription to items is one of the most popular ways to use OPC for acquiring data values. It requires relatively little coding for preparation and processing of received values.

To receive subscriptions the group(s) that contain the tags have to have been dimensioned in the VBA code with events. In addition the group(s) have to have properties

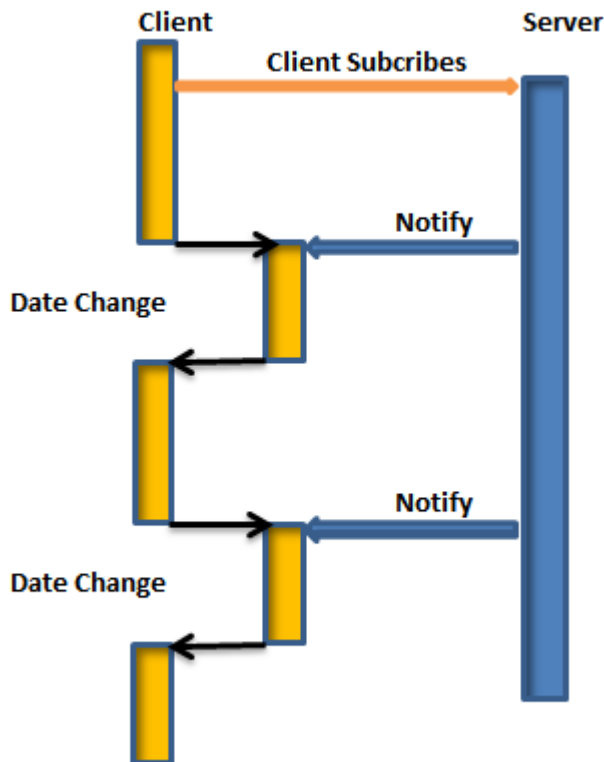
1. IsActive enabled.
2. IsSubscribed enabled.

The advantage of this method is that once the client has added the items to the group, it is the OPC server what will poll the controller (end device) at the UpDateRate of the group. If no value has changed the OPC server will not send the values to the client. This means that the client is free to continue the client task without interruption. If item values do change between poll (UpDateRates) then those items are sent to the client as an event. Only the items that have changed will be sent not all items in the group. The client will then be interrupted by the event and can process the sent item values.

In the diagram below the client has added a group (with events) and tags (items) and the group is active and subscribed. At that point the OPC server will poll the controller at the group update rate. If any of the tags have a changed value between the previous poll and the current one a notification is raised to the client (Data Change event). The client can then process the received values with the time stamps and quality flags.

The popularity of this method is evident as it leaves the client free to run client code, only being interrupted when tag values have changed.

Subscription Read Communications



Each group has its own associated data change event. In addition the event can be raised as a global data change. In this latter case the group name is included in the data change event. This may be preferable if the data change event client code is the same for all tags and groups. However the preferred method is to use the individual group data changes.

It is not good practice to have the same tag(s) in multiple groups and not at all good practice to have the same tag(s) in multiple groups with different update rates.

Consider enabling/disabling the `IsActive` group property to control the item subscriptions being raised by the server. For example if 20,000 items are subscribed and they all change value between poll cycles. The server will raise the 20,000 to the client. The client needs to be able to handle the expected maximum item changes. By setting the `IsActive` to false the subscriptions will not be raised to the client. This is one of the reasons the groups created by the client should be carefully considered to make use of this technique. Don't set the `IsSubscribed` to false and leave the `IsActive` true. Whilst this would stop the subscriptions being raised to the client, the server will continue to poll the controller which is simply wasting bandwidth and resources.

As the server only raises the data changes event to the client when a value of a tag changes, it is not possible to tell if 'no communication' means nothing is changing or there is a problem with the server connection. Hence it is advisable for the client to make a

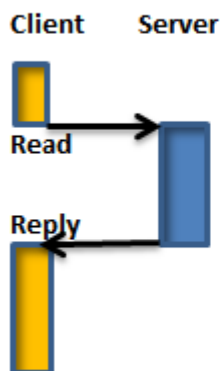
handshake call to the server. This can be via one of the other methods discussed in this section or a tag that is programmed to change every time period, hence the client can detect no updates arriving in the time period.

Keep in Sync

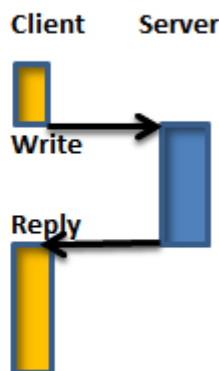
The subscription methods above is a very popular method of reading tags as it requires relatively little code to implement and gives maximum time to the client to execute its own code. However when it comes to writing tags there is no equivalent of the subscription methods. To write tags there are two methods: synchronous calls (or Sync) and asynchronous calls (Async). These two methods are also available to perform reads, ie Sync Read and Async Read. In this section sync calls are discussed.

The sync read/write methods are not a very popular method in clients. This is for one very good reason, because a sync read/write call is a 'blocking' call. That means that when a client sends a read or write request, the client code is paused until the server returns the read values or the write confirmations. Or if the server is having problems connecting to the controller or there is a network disruption between server and client, a timeout will occur. This timeout could be many minutes, during which time the client code is not executing. Hence this is not a popular method to use.

Sync Read



Sync Write

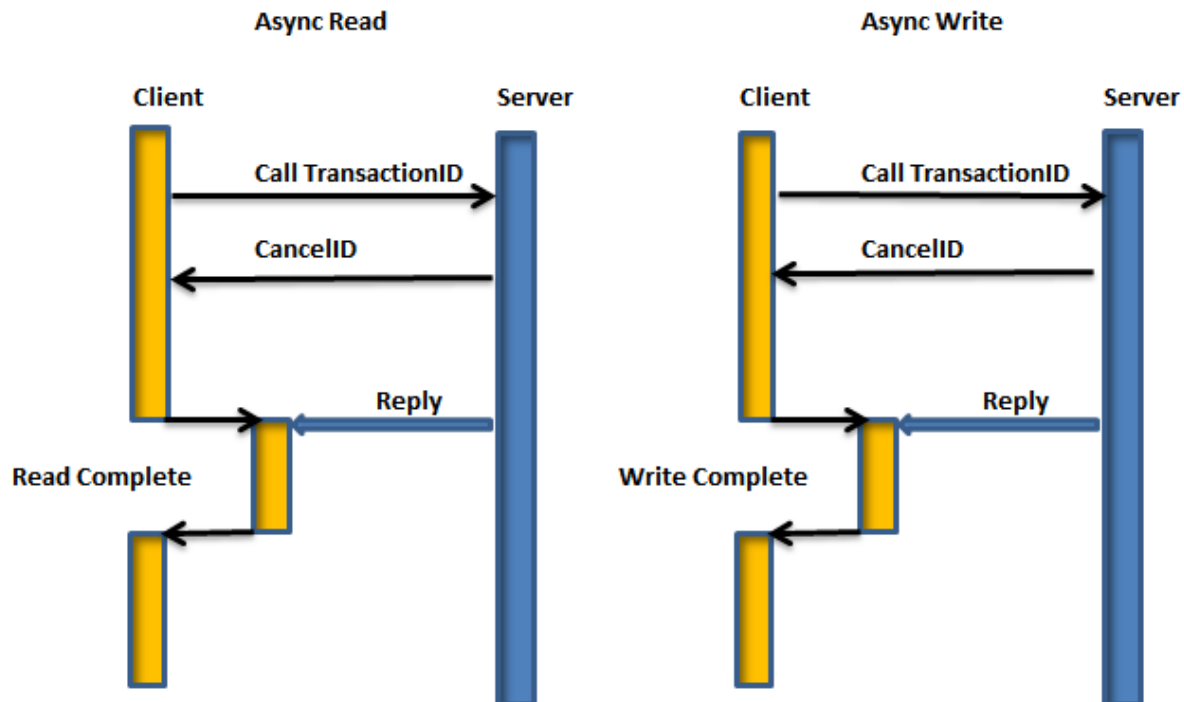


However this does not mean that the sync methods do not have their uses:

- The first one is that the sync methods are relatively easy to code in the client. Add group, then add tags, then call sync read/write methods. So it can be a quick to put together a client.
- A second reason to use the sync methods would be if the client should not proceed with any code until it receives values from the controller, and that is the purpose of the client. Ie only purpose of the client is to read values(s), process the values, which then results in the next action. Which may be a further read/write. Or another example – client should not continue until confirmation of values has been read/written.

What do you Async about

The most popular methods for reading and writing tags in OPC clients are the Async methods; Async Read and Async write. These methods give you the best of both worlds, communication initiated and controlled by the clients (as for Sync), but with the benefit that the Async calls are 'non-blocking'. This means the client can continue running client code and not be troubled by being blocked waiting for a server response. If there is a downside to using Async methods (and this should certainly not stop a developer using them) is that there is more code to write to implement Async communications correctly.



When using Async methods these points should be considered to ensure correct use of the read/write methods.

1. When the client makes an Async call it passes (along with the tag details, see spec) a TransactionID to the server. The server will return this TransactionID when it replies by raising the read or write complete event. Hence it is good practice that the client generates unique TransactionIDs to help ensure the raised call has had a response from the server.
2. The server will return immediately after the call a CancelID. This should be held by the client in case it is necessary to cancel a request that had not had a timely response from the server.
3. As can be seen from the above diagram the client can resume its code immediately after making a Async call. This code can (and probably will) include further calls to the server, before receiving replies to the previous calls. It is good practice to keep a running counter of outstanding requests. Each time a request is sent to the server increment the counter by one, each time a valid reply is received from the server decrement the counter by one.

Why is this useful?

A counter implemented correctly in this way can give indications of problems and loadings on the server. For example:

- a) The counter should never go negative, if it does it implies that more replies are being received than have been requested or that replies are not being handled correctly.
- b) In normal operation the counter should have a value within a normal operations window of values. This can be estimated from the code or from testing the client/server configuration. The reason for this is, as the operator/system carries out tasks the loading will peak and minimise. Hence values that fall significantly outside this normal operation window can be used to indicate problems in the system.
 - i. Lower values in the counter than expected. This could occur if the client is not requesting the data at the normal speed. This could be caused by several things, amongst them could be:
 - The client requesting data is based on some other component in the system, ie a database. If that part of the system slows down the client request rate for data could also slow down.
 - The server or end device (controller) is responding faster than normal. This is of course a good thing – but if nothing has been deliberately changed it is probably worth investigating. This could also indicate that the controller has been put into programming mode, ie the controller is responding faster to communication traffic in this instance.
 - ii. Higher values in the counter than expected. This could occur if the server stops responding or the controller is powered off or disconnected. Could also indicate the controller (communication module) bandwidth is not sufficient for the number of requests being sent.
 - iii. For both of the above cases the rate of change in the counter can indicate the nature of the issue. Fast rates of change indicating a major issue, slower rates of change indicating some other variable being introduced into the system.
- c) The counter can also be used by the client to cap how many requests are sent in a particular period. RSLinx Classic and FactoryTalk Gateway will queue requests that occur too quickly to send the controller at one time. However it is good practice to be able to control this value if it is known that the controller is being highly loaded. This can be as simple as checking the count before sending next requests or more complicated by implementing a client side queuing system.

4. It is good practice for the client to keep a check on timeouts for requests. When a request is made and the CancelID is received the client should be monitoring how long that request has been unfulfilled. If the time is exceeded a Cancel request (AsyncCancel) should be issued before requesting the same data. Both RSLinx Classic and FactoryTalk Gateway have internal time out for request unfulfilled, however it is good practice to implement in the client also.

Don't Hang Around:

When the client receives any of the notification events; Data Change, Async Read/write complete, Cancel complete etc. it is important for the client to process and complete the call quickly. Delays in processing can lead to locks occurring in subsequence raised events. This is particularly prevalent when using a .NET wrapper to interface between a .NET client and a COM data server.

The advice is to quickly use/store the return data and continue further processing outside of the raised event.

Try and Try Again:

It is the clients responsibility to respond to any errors received from the server and check OPC quality. It is good practice for the client to automatically retry reads/writes as some errors are inevitable, particularly in high capacity communication systems.

Clients Log:

It is good practice for the client to keep a log of actions, events and errors. This could be an option to turn on/off if debugging is required. Combining this with the logs generated by RSLinx Classic and FactoryTalk Gateway can be invaluable in isolating issues in the products or client implementation.