



OPC UA Data Acquisition of Arrays of Scalar Values for Software Engineers

Rockwell Automation Publication OPCUA-WP001A, December 2019

David A. Levine, Senior Software Engineer, Rockwell Automation

This paper discusses using OPC UA arrays in control systems, focusing on how an OPC UA client can implement data acquisition with arrays in UA servers, including the costs and benefits of using different strategies. Additionally, it provides background information on a UA server's address space and the objects within it used for data acquisition.

This paper was written to version 1.04 of the *OPC UA Specification*. Some details may have changed since this was written.

Introduction

This paper is targeted primarily at software engineers who are planning to or are already adopting OPC UA into their products. Control systems engineers, support personnel and product managers will also find the information useful to better understand the strengths and weaknesses of arrays in OPC UA, its capabilities, potential support issues, and how it can fit into existing products.

This paper assumes the reader is familiar with OPC UA; background material on nodes, NodeIds, Variable nodes, and node attributes is provided.

OPC UA is gaining market acceptance and is anticipated to be widely adopted as a result of its official approval by *Industrie 4.0*, *Made in China 2025*, and other international organizations and efforts. OPC UA supports features that were nearly impossible to build using older technology (such as methods and transactions), includes built-in security, is multi-platform and is technology agnostic. An advantage of particular note is the extensible type system which makes it possible to create domain specific extensions that are industry specific. New types can be created by a single organization or created by a consortium and made public and available for an entire industry (for example, MDIS - undersea oil & gas). With the recent addition of publish and subscribe (pub/sub) and communications to the cloud it supports Industrial Internet of Things (IIoT).

The focus of this paper is on data acquisition from a control system, where the control system contains programmable logic controllers (PLCs) which the UA server uses as its data source. The data types discussed are limited to acquisition of arrays of scalar values – structures and arrays of structures are not discussed except as background information.

The practices recommended for OPC UA arrays are not limited to systems based on PLCs – they can be applied to any system – but their emphasis is on operations and data supported by PLCs.

Caveats

This paper is written from the perspective of a UA client, not a UA server. It does not discuss what a UA server must do to support arrays, it focuses on the client side. There are pitfalls with reading and writing arrays and this paper discusses those that were discovered when writing a UA client.

The OPC UA Specification is over 1000 pages and complex. Inevitably different engineers in different parts of the world will interpret the same specification in many different ways, leading to incompatibilities, inconsistencies, and other unexpected defects, faults, or flaws in a program. To help mitigate this the OPC Foundation has a certification process to help ensure that OPC UA clients and servers are in compliance with the specification. This certification process helps heterogeneous applications from different vendors achieve interoperability.

Even with interoperability and certification a general-purpose client may find it difficult to support all servers and still achieve a high level of performance. Servers, ranging from low-end embedded devices to high-end desktop servers, are built to support different profiles and feature sets, and will operate at different levels of performance – all of these can impact throughput available to a client. This paper is intended to help developers create OPC UA clients which use strategies that are best suited to achieve their goals, both functional and for performance.

In many cases the OPC UA Specification was so well written that this paper reprints some sections directly from it.

Background

This section summarizes key points about the OPC UA address space and the objects it contains. For the full definition, refer to the *OPC UA Part 3 – Address Space Model*.

Address space

A UA server's address space is an object-oriented model of the underlying system; different components of the model are represented in the address space as nodes and references between nodes. The purpose of the OPC UA model is to provide a standard way for servers to represent objects to clients. The objects are presented to clients as *nodes*.

One of the differences between the nodes in an OPC UA server and the data variables in a PLC is how the address space is organized. Most PLCs have a relatively simple hierarchy – a single starting point (the root or parent) with a single path through the hierarchy (children) to reach *DataVariables*. There are no cycles (loops) allowed, meaning a child cannot have a link to a node that would lead back to the child's parent.

OPC UA is more flexible and complex. It too has a single starting point (*Root* node) along with 3 nodes that are always below it – *Objects*, *Types*, and *Views*. There is a default address space that almost all servers must contain which includes diagnostics, information about the server's capabilities, and the base type system.

Beyond this the server has a lot of flexibility in how it defines its address space. It can have loops and relationships between nodes other than the simple parent-child relationship present in the PLC. This allows the server to model the underlying system more closely and more meaningfully than is possible with a simple hierarchical address space.

Client browsers must be aware that the same node can be reached through many different paths through the hierarchy. When uploading/browsing the address space it must guard against evaluating the same node multiple times and getting stuck in an infinite loop.

Nodes

An OPC UA server exposes objects of interest to an OPC UA client as *nodes*. An object is defined in terms of *Variables* and *Methods* and can have relationships to other nodes. Nodes come in a variety of types, called a *NodeClass*. *NodeClasses* are defined in terms of *Attributes* and *References* that are given values when a node is created in the address space. Each node in the address space is an instance of one of these *NodeClasses*.

NodeId

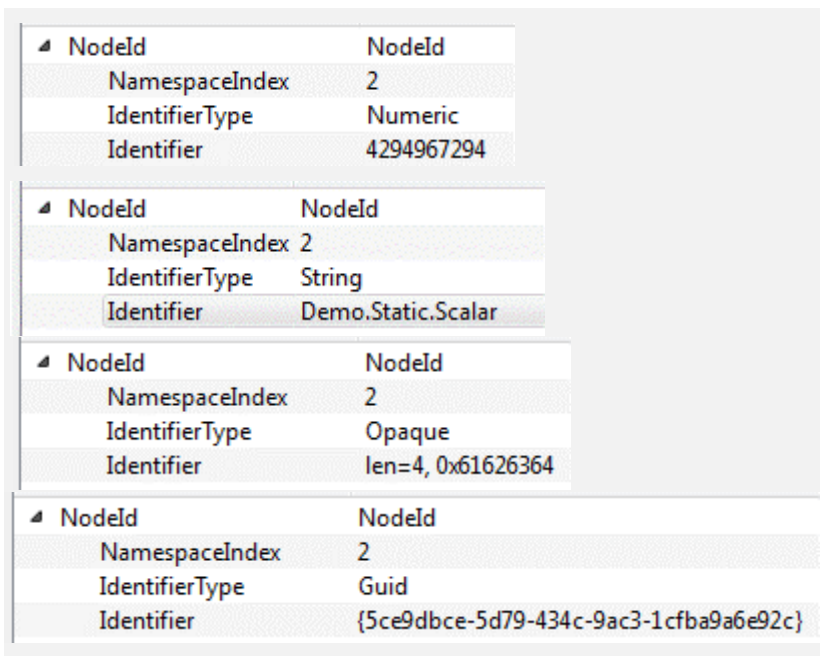
The *NodeId* is formally defined in the *OPC UA Part 3*, section **8.2 NodeId**

Nodes are unambiguously identified by its *NodeId*, a built-in data type. Every item in a UA server is represented by a *node* and all nodes are identified by a *NodeId*, including objects, variables, references, data types, and so on. All nodes within a UA server must have a unique *NodeId*.

A *NodeId* consists of three fields: a *NamespaceIndex*, an *IdentifierType*, and an *Identifier*. The *Identifier* portion can be one of four basic data types: an integer, a string, a GUID, or a byte string (opaque stream of bytes). The type used is determined by the value of *IdentifierType*.

Here are examples of each type:

Figure 1: Screen shots of all four types of NodeIds



▲ NodeId	NodeId
NamespaceIndex	2
IdentifierType	Numeric
Identifier	4294967294

▲ NodeId	NodeId
NamespaceIndex	2
IdentifierType	String
Identifier	Demo.Static.Scalar

▲ NodeId	NodeId
NamespaceIndex	2
IdentifierType	Opaque
Identifier	len=4, 0x61626364

▲ NodeId	NodeId
NamespaceIndex	2
IdentifierType	Guid
Identifier	{5ce9dbce-5d79-434c-9ac3-1cfba9a6e92c}

The *NamespaceIndex* is an index into a table of URIs constructed by the server each time it starts. This is used to identify the *naming authority* which assigned the identifier in the NodeId. The namespace index can change from session to session but the URI it points to cannot change. The UA server typically does not change the indexes unless the address space has changed. It is the client's responsibility to fixup NodeIds each time it creates a new session with a server.

The combination of the fields *Index* and *Identifier* make it unique. It is legal and valid for multiple nodes at the same hierarchal level to have the same *Identifier* – as long as each has a different *Index* it is unique.

For example, there may be two nodes under the *Objects* folder with NodeIds set to (index=4; identifier="MyObject") and (index=5; identifier="MyObject"). Even though they each use the same identifier value, each NodeId is unique because of the different index value.

Attributes

A node consists of attributes and a table of references to other nodes. There are attributes common to all node classes, four of which are mandatory, and the others are optional. Each node class has additional attributes specific to that class, some of which are mandatory, and some are optional – the set of additional attributes varies with the *NodeClass*.

There are eight different *NodeClasses*. Some types are used to help organize the address space, some to represent objects, such folders and data, and some to identify data types, such as integers, structures, and custom data types.

The set of *NodeClasses* cannot be extended and each node in the UA server must be one of these classes. They are all derived from a common base class – the base class itself is never directly instantiated.

An attribute is a chunk of data which has a name, data type, and a value – the value may be a scalar or a structure. Some attributes, such as references, may have multiple entries. Some attributes are optional - for a given attribute refer to the OPC UA to determine whether it is *Mandatory* or *Optional*. If an attribute is listed as optional and not implemented in the server then it cannot be accessed by the client - attempts to access it will fail. If the attribute is optional and the server implements it then it may be accessed the same way that all attributes are accessed.

A reference can be in either direction, to another node or from another node, so references can function like a doubly linked list. The *type* of reference defines the relationship – for example, an *Object* can contain a *Variable*; the *Object* will contain a forward reference such as *HasComponent* and the *Variable* will contain a reverse reference, e.g. *ComponentOf*. References are optional – a node may not have any references at all.

The base class from which all other classes are derived is formally defined in *OPC UA Part 3 – Address Space Model*, section 5.2.1. *Table 7 – Base NodeClass* defines its attributes.

The *Mandatory* attributes that all nodes possess are:

- *NodeClass*
- *NodeId*
- *BrowseName*
- *DisplayName*

The *Optional* attributes that all nodes may possess are:

- *Description*
- *WriteMask*
- *UserWriteMask*
- *RolePermissions*
- *UserRolePermissions*
- *AccessRestrictions*

NodeClass defines what type of object the node is.

NodeId defines how to unambiguously refer to the object.

BrowseName defines an alternative way to refer to the object, but it is not unambiguous – there may be multiple objects at the same hierarchal level with the identical *BrowseName*. Clients can use a service to translate relative symbolic paths (referred to as a *BNF* path) into a *NodeId* and must be prepared to handle multiple *NodeIds* for the same *path* + *BrowseName*. This field is not localized and does not change with the locale.

DisplayName: contains a string and a locale. The intended usage for this field is that client applications use the contents of this field to display to end users instead of any other data, such as the *BrowseName*. The content of this field may change with the locale.

Attributes are data fields that can potentially be read, written and monitored for changes – the server may not support all operations for all attributes and optional attributes may not be present.

Data acquisition is interested in *DataItems*. This is an abstract term which refers to “live” automation data.

There are several concrete types of *DataItems*, including *AnalogItem* and *DiscreteItem*. *DataItems* are represented in the address space by nodes whose *NodeClass* attribute is set to *Variable*. Variable nodes are the only *NodeClass* containing a *Value* attribute.

Variable nodes contain both mandatory and optional attributes that used in data acquisition - here is a partial listing:

Mandatory attributes:

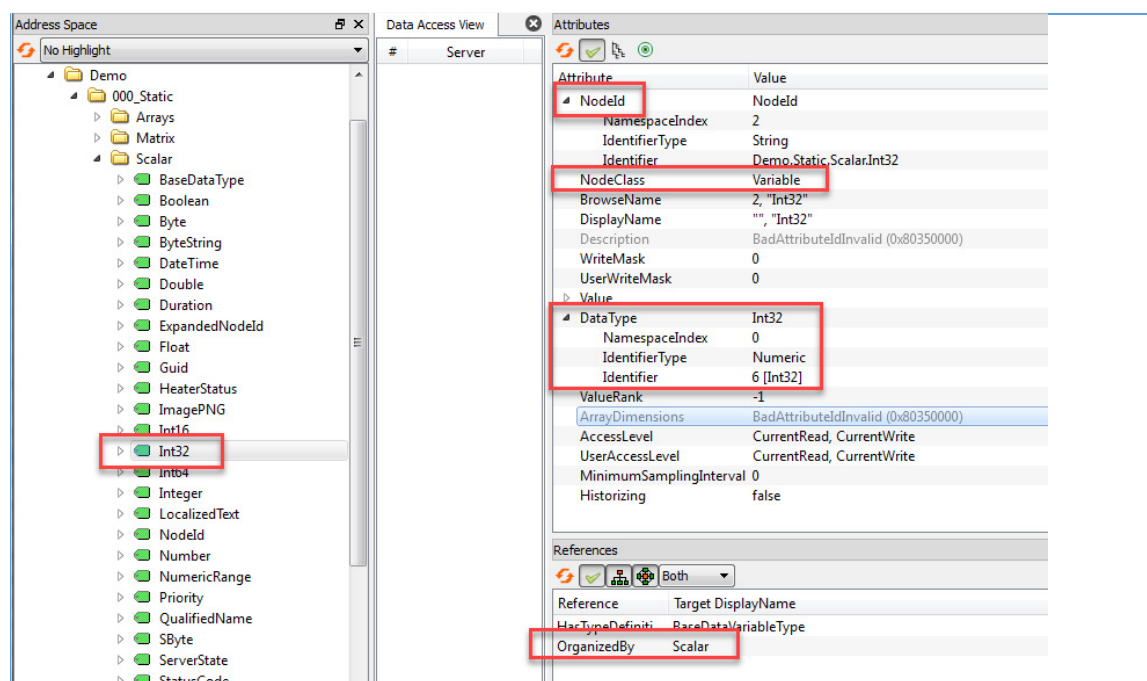
- *Value*: this can be simple or complex and can be of any data type as specified by the *DataType* attribute
- *DataType*: the *NodeId* of the *DataType*
- *ValueRank*: defines if it is a scalar, an array, or either.

Optional attributes:

- *ArrayDimensions*:

These are later discussed in detail. A complete listing of the *Variable* class's attributes is in *OPC UA Part 3*, section **5.6.2 Variable NodeClass**.

Figure 2: Screenshot of a Variable node, its attributes and references



This screenshot shows a node and its relative location in the address space. The *NodeId*, *NodeClass*, *DataType* and *References* to other nodes are highlighted.

Metadata

Variables can have *Properties* that act as *metadata* applied to the value of the *Variable* and characterize what the value represents. A *Property* is when a *Variable* node has a *HasProperty* reference to another *Variable* node whose *Value* is the metadata.

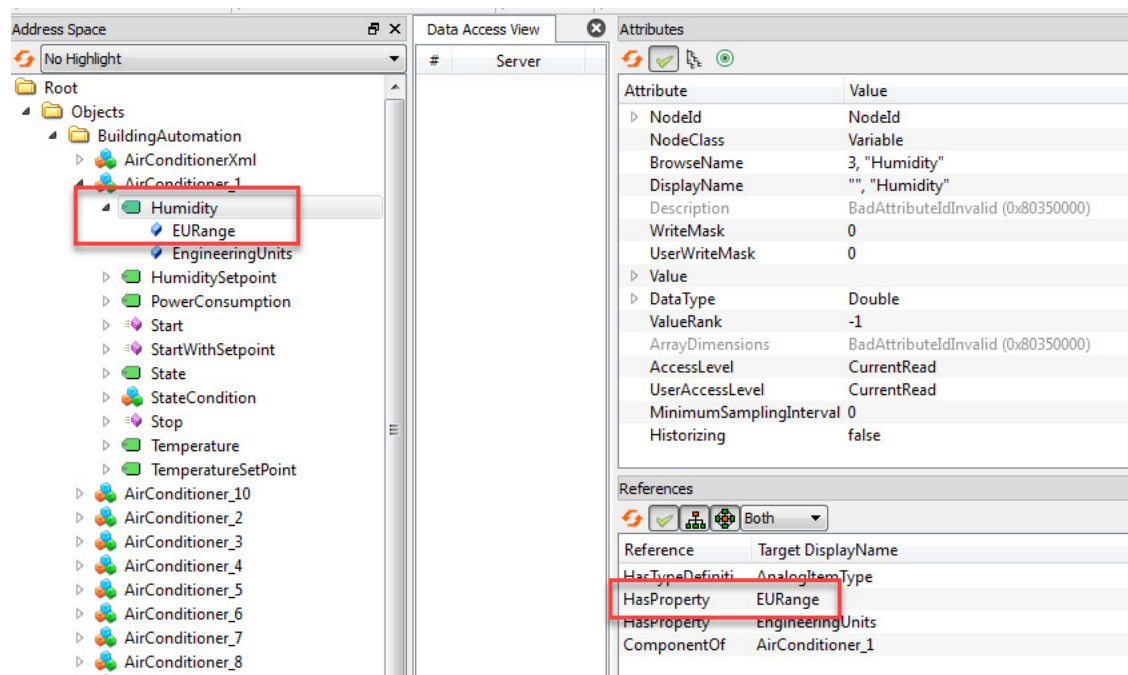
This definition of metadata allows domain specific metadata to be created. A limited set of properties is defined by the OPC UA Specification. These are all structures that must be read or written as a single transaction - it is not possible to read/write sub-elements without reading or writing them all.

An example of a property that acts as metadata is *EURange*. *EURange* defines a range of values the *Variable* can have and consists of two elements: *low* and *high*. Its datatype is *Range*, which is defined in *OPC UA Part 8 DataAccess*, section 5.6.2.

Although *DataVariables* and *Properties* are used for different purposes and have different constraints they are both instances of the *Variable NodeClass*.

The next screenshot shows an example of a *DataItem* named *Humidity* with metadata defined using *EURange*.

Figure 3: A Variable with Metadata attached to it: EURange is metadata applied to Humidity



Data Types

Before tackling arrays, the UA client must handle scalars, but this does not mean that it must handle all the different scalar data types. When initially developing a component in a new technology start with what is needed (start small), get that working correctly, and build on that.

One way to limit the scope of the initial development effort is to support only the data types required by your product. When interacting with PLCs this typically means restricting data types to integers, from 8 to 64 bit signed and unsigned, floats, doubles, and strings.

UA defines many native structured data types, such as *LocalizedText* (contains 2 strings, text and locale),

QualifiedName (1 integer, 1 string), and many others. If your system does not directly support structures or all the other data types, you can support a subset of these. Another option is to encode the values of unsupported types as a string so it can be displayed or logged – this may not be directly usable in the application, but it might provide useful diagnostic information.

Mapping between OPC UA and OPC DA

Many existing and successful products were built to the original OPC DA specification and are being updated to work with OPC UA. It's relatively easy to go from OPC DA to OPC UA – UA is a superset of DA - but the reverse is not true.

Mapping numeric values is the easy part but many UA data types cannot be directly supported. There are also many error and quality codes in OPC UA that do not map directly into OPC DA and must be mapped into a few quality codes. This results in generic error and warning messages instead of detailed messages that help users diagnose and fix problems.

One of the harder parts of mapping OPC UA into OPC DA is identifying items of interest – converting from a string-based identification system (DA) to a system based on *NodeIds* (UA). Doing this for a UA server whose data source is a DA-based system is relatively easy – the string identifier used in the DA system is made the identifier portion of the *NodeId*.

UA Clients that map into DA systems cannot do this because the client does not define the *NodeIds* – the server does. Clients need a way to map UA *NodeIds* defined by a server into DA string identifiers consumed by the DA client and back again. If the OPC DA system preserves the DA string identifier from the string displayed to the user, then this mapping process becomes easier. If the OPC DA system uses one string for both, then this process is harder – displaying *NodeIds* directly to the user instead of human-readable text is user-hostile and should be avoided.

Operations

OPC UA clients used for data acquisition primarily read and write data to and from Variable nodes – almost all other operations are related to connection and configuration activities and are not relevant to the subject of arrays.

A UA Variable node can expose a single scalar value, an array of scalars, a structure, an array of structures, and an array of both scalars and structures, where each element can be a different type.

A UA server may support none, some, or all of these, in increasing degrees of sophistication and complexity. Each of these requires a client at the same level of sophistication in order to work correctly and efficiently with these.

A UA server may expose an array from its underlying data source in three different ways. It can:

- Create a folder that represents the array and create a *Variable* node for each element and assign each a *NodeId* - the clients access each element as individual nodes.
- Create a *Variable* node whose *Value* is the array. Clients must access it as a UA array.
- Both.

Exposing each element individually makes it compatible with unsophisticated clients that do not support UA arrays. However, this requires more operations to read/write the entire array and may degrade performance, and it increases the size of the address space, which may become so large it causes undesirable side effects. This consumes additional memory and CPU cycles on both the client and

server.

Exposing arrays as UA arrays may make it incompatible with some clients. Its benefits include reduced memory consumption and higher performance – a single operation can be performed on multiple elements.

A server can do both – expose the *Variable* as an array for high performance clients and expose each element in the array as individual nodes for unsophisticated clients.

Commercial servers have implemented all three options.

Data Encodings and Transport Protocols.

All communications between clients and servers is based on the exchange of messages. The format of a message is specified by the encoding and transport. Each combination of an encoding and transport (plus serialization and security) comprise a stack.

A data encoding describes how to construct a message. OPC UA supports 3 types of encoding: UA Binary, UA XML, and UA JSON. These bind to different transport protocols: UA TCP, HTTPS, and AMQP. The combination of encoding and transport is called a stack.

UA JSON and AMQP are used for publish-subscribe messaging to the cloud and JavaScript clients; high speed data acquisition uses UA Binary and TCP.

Arrays should be supported in all encodings and transports, however there may be restrictions imposed by some stacks.

Reading Data

All attributes, including values, are accessed using the same set of services – this is a great feature because it unifies the service set. The same read command will read the value of a *DataItem* and any other attribute's value as well; all that changes in the request is the attribute's identifier. This holds true for all items in the address space.

Multiple requests can be combined into a single packet sent to the UA server. Each request is an individual operation which arrives in the same packet over the network. The order of processing matches the order within the packet and the replies match the requests by position within the packet.

To read data the client constructs an array of *ReadValueIds* which it sends to the server.

Each of these contains

- The *NodeId* of the target node
- The identifier of the attribute to read (a non-zero unsigned integer)
- An optional *IndexRange*

When determining if the node is an array the client reads the *ValueRank* and *ArrayDimensions* attributes, usually during configuration and validation. When running data acquisition operations, the client reads the *Value* attribute.

Reading attributes and values can be more difficult than expected because most data objects exchanged between the server and client are encoded as a *Variant*, which contains both the data and its data type. The safest approach to reading the data value is to extract the data using the type information contained in the reply data (if present), not necessarily what the specification or configuration data calls for. It may not work correctly if the data is extracted based on assumed knowledge of the data type. Some servers are dynamic (or have unexpected behaviors) that cause unanticipated changes to the data type.

Unless you are writing your own OPC UA stack (not recommended!) you are using an OPC UA toolkit. It may have methods to extract data from a *DataValue* using APIs that are specific to a data type. If so, then if the API does not match the actual type you will get unexpected results. The technology you are using may also impact this. Developer beware!

Writing Data

To write data the client constructs an array of *WriteValues* and sends it to the server. Each *WriteValue* instance contains:

- The *NodeId* of the target node
- The identifier of the attribute to write
- An optional *IndexRange*
- The value(s) to write

Subscriptions and Monitoring Items

To monitor items for data changes the client first creates a subscription which defines the update rate and then adds items to the subscription.

Each item contains:

- The *NodeId* of the target node to monitor
- Optional *IndexRange*
- Optional Filters

Multiple items can be added to the subscription at the same time.

Constraints on Reading, Writing, and Other Operations

There are limits, both server and system imposed, on the amount of operations and data that can be requested or transmitted in a single packet. There is support in OPC UA for runtime discovery of some constraints, but many of these are optional and may not be present.

When a client initially connects to the OPC UA server it should read the nodes under folder *Root/Objects/Server/ServerCapabilities* and *Root/Objects/Server/ServerCapabilities/OperationLimits* for constraint information. These are defined in *OPC UA Part 5 – Information Model*, section 6.3.2 and *OPC UA Part 5 – Information Model*, section 6.4.11 respectively.

The *ServerCapabilities* object is a mandatory object and contains nodes which can be useful for data acquisition. For example, node *MaxBrowseContinuationPoints* is mandatory and specifies the maximum number of parallel continuation points the Browse Service supports – this is used when uploading the address space.

The property *ServerCapabilities/MaxArrayLength* indicates the maximum overall number of elements in all dimensions that may be contained in an array, regardless whether it is a one or multi-dimensional array. The server may put further restrictions on individual variables and may even support a larger size than is specified by this property. A client should be flexible and adapt to handle whatever the server supports.

The nodes below the *OperationLimits* node are useful but all are optional and may not be present. If a constraint is present, then the client should read the value and use them.

Some of these nodes to be aware of are *MaxNodesPerRead*, *MaxNodesPerWrite*, *MaxNodesPerBrowse*, and *MaxMonitoredItemsPerCall*. If these nodes are present and have a positive value then the client must chunk its requests to stay within these limits, otherwise the call will probably fail.

Many servers do not implement these optional properties or do implement them but set the value to 0 or to a very large value (such as 65536). A missing property or a property with a 0 or large value *may* mean that it can handle unlimited requests, but not always. The OPC UA Specification does not define what a client can assume when the property is missing or has a value of 0 or an extremely large value.

The Basics of Reading/Writing Arrays of Scalars

An OPC UA server exposes nodes to UA clients. Nodes can be constructed to contain scalars or arrays. It can even be either of these and its type must be discovered at runtime.

Arrays come in an infinite variety of sizes and dimensions. The OPC UA Specification does not impose any limits on the number of dimensions or size of a dimension – this is totally up to the server. It can range from an array with 1 element to millions of elements and beyond, from 1 dimension to 1000 and beyond – the spec does not require servers to be practical. ☺

Array elements use 0-based indexing, so the 1st element's index in a dimension is always equal to 0.

The contents of the array are entirely up to the server. Arrays can consist of elements of the same data type, either a scalar (single value) or a structure (many values). Arrays can consist of Variants where each element can be a different datatype from the other elements – one element can be a scalar and the next can be a structure.

This paper assumes arrays of scalars where all elements are the same data type; arrays whose elements contain different data types, structures and arrays, are not addressed.

When a node is defined as “either” a scalar or an array then the server must be queried at runtime to discover what it is for the current session. The node can change to the other type on a new session - this is up to the server.

Determination of a node's array characteristics is done by reading attributes from the node. The attributes associated with arrays are *DataType*, *ValueRank*, *ArrayDimensions*, and *Value*.

The meaning of these attributes is defined in *OPC UA Part 3 – Address Space Model*, **section 5.6.2**, *Table 13 – Variable NodeClass* and is summarized here.

- *DataType*: for scalars this is the actual data type in each element. If it can vary then it is a *Variant* data type.
- *ValueRank*: Defines whether the node contains a scalar, an array, or if it can be either.
- *ArrayDimensions*: the number and size of each dimension. This is optional and may not be present.
- *Value*: the data content of the Variable node. This includes the source and server timestamp, the status code, and the scalar or the full contents of the array.

Data Type

This is a required attribute and must be present. For standard types, this field is the *NodeId* assigned by the OPC UA Foundation to represent that data type.

Most servers contain the complete set of all data types used within its address space, located beneath the *Root/Types* node. Clients use this to decode all objects and types within the server's address space. Servers are constructed to match a *Profile*, which is a minimum set of required features. Nano and Micro profiles are not required to contain this information.

OPC UA base data type definitions are located within each UA server at *Root/Types/DataTypes/BaseDataType*.

- Numeric data types are located at: *Root/Types/DataTypes/BaseDataType/Number*.
- Integer types are located at: *Root/Types/DataTypes/BaseDataType/Number/Integer*, and include *Int16*, *Int32*, *Int64*, and *SByte*.

A Variable node that contains simple scalar values has a *DataType* attribute which contains a *NodeId* which refers to one of these types.

ValueRank

This is a required attribute and must be present. The possible values are:

- $n \geq 1$: An array with 1 or more dimensions as specified by the value of n . (The specification distinguishes between 1 dimension and more than 1, but does not explain the rationale for this distinction)
- 0: value is an array with one or more dimensions. (the actual number must be determined by reading the contents of the array)
- -1: the value is not an array (a scalar).
- -2: the value may be a scalar or an array of any number of dimensions. (the only way to determine if it is a scalar or array is to read the entire value and examine the result to see what the server sent)
- -3: the value may be a scalar or a one-dimensional array (to find out do same as $n == -2$)

All datatypes are considered to be scalar even if they have array-like semantics, like a *String* or *ByteString*

The values of *ValueRank* that require runtime evaluation are $n == 0$, -2 , or -3 . Any other value is either invalid or directly provides the information. The client must read the *Value* attribute at runtime to determine what it is dealing with. The *Value* is required to not change its nature during a session but can for each new session.

If the array is not fully specified by the *ValueRank* and *ArrayDimensions* fields then the client should evaluate the *Value*'s actual type and array dimensions each time it opens a session with the server to get the missing details.

For example, if the *ValueRank* = 3 you know the number of dimensions of the array (3), but you do not yet know the size of each dimension. If that is important (and it probably is) the client has more work to do.

If your product needs to know what the node is at runtime then it should read the *DataType*, *ValueRank* and *ArrayDimensions* attributes and, if necessary, the *Value*. If the node is a scalar, then the *ArrayDimensions* result will be missing and is ignored.

ArrayDimensions

ArrayDimensions is an optional attribute and may not be present. If it is not present it is considered to be *null*. It specifies the maximum supported length of each dimension. If the maximum is unknown the value for that dimension shall be 0. The number of dimensions must match the value of the *ValueRank* attribute if the *ValueRank* > 0.

The value of the *ArrayDimensions* shall be *null* (missing) if the value of *ValueRank* ≤ 0.

If the node is a scalar (*ValueRank* is -1) this attribute is *null*. If the node can change at runtime between a scalar and an array it is *null*. If the size of the array can change at runtime it may be *null* or set to the current value – the server chooses what to do.

Beware – even if you know the number of dimensions when the session begins, either by reading the attribute *ArrayDimensions* or by reading the *Value* and getting its size (as described below), it can change during the session. Both the number of dimensions and the length of an array's dimensions can change during a session. The server is not required to update either the *ValueRank* or *ArrayDimensions* attributes when it changes.

Discovering the number of dimensions in an array can be easy or hard, depending on the UA server's implementation. The easiest way is to directly read the attribute *ArrayDimensions* – if it exists and has a non-zero value then you can use the value directly. But remember that even if this value is present the actual array may change at any time – always use the data itself to determine dimensions and size.

If that fails and your client must know this value, then the only other way to determine the size is to read the *Value* attribute and get the entire contents of the array. When a *ReadValueId* is issued to the Read Service with a *null* (missing) *IndexRange* the server will return the entire contents of the array.

The length of the serialized data (number of elements) is always in the reply data, and if it is a multi-dimensional array, the dimensions are also in the reply data. This holds true whether an *IndexRange* was used or not – the reply data is always self-describing. The client must interpret that data based on how the command was sent. If an *IndexRange* was included the client must map the reply data into the correct elements by using offsets adjusted with a relative offset. If an *IndexRange* was not included, then the reply data maps directly to the array element and the size contains the actual dimensions of the array. This may be the only way to determine the actual size of the array.

The OPC UA specification defines the maximum number of elements that may be transferred on the wire as 2,147,483,647 (max Int32).

Locating Array Elements in Buffers – Serializing Data

OPC UA is meant to be used in distributed systems where the UA client and server are separated by networks. All data must support transfer over a network and must be serialized into a stream of bytes to get it on the wire and deserialized from a stream of bytes to get it into a component. Both sides must understand how this is done so that the values get mapped to the correct array location.

For one-dimensional arrays when indexing is not used this is quite simple – simply copy the bytes starting at element 0.

For multi-dimensional arrays it is not simple and getting it wrong can have catastrophic results. How the data is encoded is defined in *OPC UA Part 6 – Mappings*, **section 5.2.2.16 Variant**, *Table 15 – Variant Binary DataEncoding*.

In the table column titled *ArrayLength*, it says:

“Multi-dimensional arrays are encoded as a one-dimensional array and this field specifies the total number of elements. The original array can be reconstructed from the dimensions that are encoded after the value field. Higher rank dimensions are serialized first. For example, an array with dimensions [2,2,2] is written in this order:

[0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], [1,1,1]”

“Higher rank dimensions are serialized first” means the dimension with the highest value for its rank.

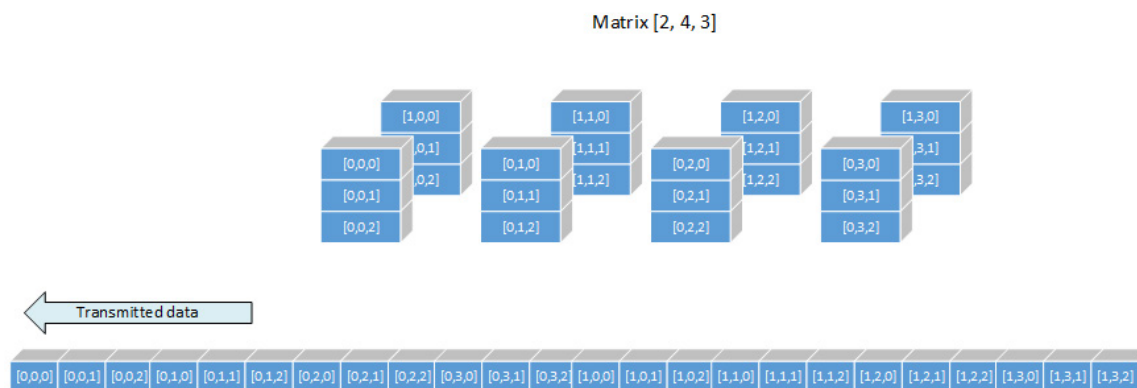
Think of *higher rank* as the innermost dimension (furthest to the right) and the lowest rank dimension (furthest to the left) as the outermost. Computer scientists call this *depth-first* ordering.

Figure 4 shows a matrix [2, 4, 3] and the order in which its elements are serialized into a flat buffer.

- The highest rank (2 or innermost) dimension has length 3 and is shown as moving from top-to-bottom (y axis). Think of these as buckets of data.
- The middle rank (1) dimension has length 4 and is shown as moving from left-to-right (x axis).
- The lowest rank (0, or outermost) dimension has length 2 and is shown moving from front-to-back (z axis).

The transmitted data is shown serialized into a stream.

Figure 4: Serialization of a matrix



The order in which elements are serialized/deserialized must be consistent for all clients and servers regardless of platform and operating system, otherwise data will get corrupted.

Algorithms

In the case where the entire contents of the matrix are used, locating an element within a serialized buffer can be done using these algorithms (represented below using pseudo-C):

LocationToFlatOffset:

```
offset = 0
sizeofInnerDimensions = 1
for ( element = dimensions - 1; element < dimensions && element >= 0;
    element-- )
    offset = offset + location[element] * sizeofInnerDimensions
    sizeofInnerDimensions = sizeofInnerDimensions *
dimensions[element]
return offset
```

Using the example from the specification, in matrix [2,2,2] for element [1, 1, 0] there are 3 iterations of the *for* loop

1. $[0 + (0*1)] = 0$
2. $[0 + (1*2)] = 2$
3. $[2 + (1*4)] = 6$

Buffer[6] contains the data for array element [1, 1, 0] – this is the 7th element in the buffer.

Going the opposite way is a little more complex. Mapping from a flat buffer to a specific array element requires knowing the actual size of the array and the offset into the flat buffer.

FlatOffsetToArrayElement(offset, ArrayDimensions)

```
for ( dimension = 0; dimension < ArrayDimensions; dimensions++ )
    sizeofInnerDimensions =
CalculateSizeOfInnerDimensions(ArrayDimensions, dimension)
    location = offset/ sizeofInnerDimensions
    arrayLocation.Add( location )
    offset = offset - (location * sizeofInnerDimensions)
return arrayLocation
```

When dividing integer values the result is a whole integer and the remainder is dropped. For example, $5/4 = 1$. This applies to line *location = offset/size*

Using the same example as above there are 3 iterations of the *for* loop: (offset = 6)

1. sizeofInnerDimensions = 4; location = $6/4 \rightarrow 1$; offset = $6-4 \rightarrow 2$
2. sizeofInnerDimensions = 2; location = $2/2 \rightarrow 1$; offset = $2-2 \rightarrow 0$
3. sizeofInnerDimensions = 1; location = $0/1 \rightarrow 0$; offset = $0-0 \rightarrow 0$

This produces arrayLocation [1, 1, 0].

This algorithm starts at the lowest rank dimension (outermost/leftmost) and moves inwards towards the higher ranks. The element location for the current dimension is calculated by dividing the offset by the size of all the inner dimensions – these are the number of elements that must get “used up” for the outermost dimension to increment by 1.

CalculateSizeOfInnerDimensions is left as an exercise for the reader ☺ (hint: multiply together all

dimensions whose rank is larger than the current one).

For example, an array of size [100, 10] contains 1000 total elements. There are 10 elements between [0,0] and [1,0]. Before the leftmost dimension can be incremented by 1, the elements [0, 0], [0, 1], [0, 2], etc. must first be traversed.

These algorithms are provided here because unless someone provides a library that contain these conversion functions, you will need to write your own. Additional algorithms are needed to calculate offsets when using Indexes – those will be discussed in the next section.

Accessing Array Elements

There are two different ways of accessing the data in an array. One is by reading or writing the entire contents of the array. This is suitable for small size arrays and for arrays which change value very slowly. This mode is referred to as *Read-Modify-Write*, or *RMW* because this is the manner used for writing data to the array.

The other way is by specifying a range of elements using a *NumericRange*. This is suitable for large arrays and for arrays that are changing value quickly. This mode has pitfalls and will be discussed in detail. This mode is referred to as *Indexed* and is optional – servers are not required to support it.

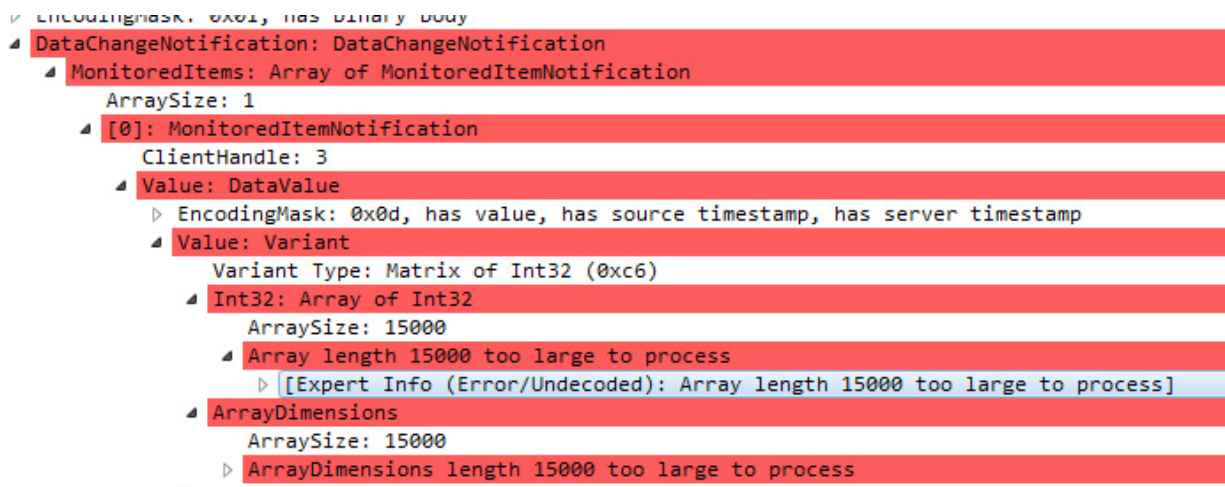
Both approaches have advantages and disadvantages.

Read-Modify-Write (RMW)

It is always “safe” to read the array in its entirety; however, if the array is large it may have problems being transmitted over a network and the OPC UA stack may generate encoding errors. “Safe” in this context refers to data integrity –the data read or written has not been corrupted, nor has data in the OPC UA server been incorrectly changed.

TCP can handle the fragmentation and reassembly but tools like *Wireshark* may not be able to decode large packets. For example, an array with 15,000 elements causes *Wireshark* to display a packet like this:

Figure 5: Wireshark cannot display packets that are "too large"



While it is always “safe” to read the entire array, writing to all the elements of an array using *RMW* may

not be safe.

If the client is intending to write to all the elements in the array then, assuming there are not any network errors, the integrity of the data is not at risk – the client is intentionally setting all the values and overwriting whatever is there.

However, if the goal is to write to only a subset of elements then it may or may not be “safe” – it depends on the timing of the write operation and the current operations of the server. *RMW* mode means the client first *reads* all the elements in the array, and then as fast as it can *modifies* a subset of values, leaves the others alone, and *writes* them all back to the server. If the data in the server has not changed between the moment when the array was read and the time the modified array arrives back at the UA server then the data is uncorrupted.

If the UA server changed the value in any of the unmodified elements before the modified array is received, those changes will get overwritten with old values when the data arrives. This will result in the UA server containing old values intermixed with new values from the client. In a production environment the consequences can vary from having no impact to a system failure.

Some servers allow the client to change the size of an array. Another potential problem with RMW is in accidentally changing the size of an array. When the client writes an array without using indexing the server may simply replace the current array with the new one, regardless of the new size.

A production server should apply suitable constraints to data written to it to prevent accidental changes like these, but some may omit this check. If these risks are unacceptable in your environment, then your other option is to use *Index Ranges*. However, the server may not support indexed write operations.

Indexing Mode (NumericRange)

All servers are required to support indexed reads but support for indexed writes is optional. However, even though the OPC UA specification requires support of indexed reads there are servers that do not support it, and the specification does not define a way to determine if a server supports indexing mode.

An *IndexRange* is the field name in the data that is transmitted over a network. This field contains a *NumericRange*, which is a data type defined in OPC UA Specification. You can find the formal definition of a *NumericRange* in *OPC UA Part 4 – Services*, section **7.22 NumericRange** and **Table 164**. If an *IndexRange* is not used, then the field in the transmitted data is **null** and the operation is performed on the entire array (*RMW*).

Index Range

A subset of contiguous elements is specified by defining one or more *NumericRanges* (*NR*), one for each dimension of the array. *Contiguous* means in its serialized form, not just within a single dimension.

An *NR* defines a range of elements within a dimension by specifying a starting and ending offset – these are referred to as the lower and upper bounds.

There must be one *NR* for each dimension of the array. For a 1D array there is a single *NR*, a 2D requires 2 *NRs*, and so on.

When constructing an *NR*:

- All dimensions are indexed starting with 0.
- $\text{Start} \leq \text{End}$ - the starting element must always be less than or equal to the ending element.
- A range defines a continuous set of elements with no gaps, so if you need to skip some elements you must create a new *NR*.
- Each *NR* must be specified in the same order as the array dimensions.
 - The first *NR* is used for the dimension with the lowest rank value (the outermost or leftmost dimension).

String Representation of a NumericRange

There are two types of constructs, an individual element or a range of elements.

When a single element is specified, such as, element 34, the *NR* contains *NR.Start* = 34 and *NR.End* = 34. The string representation is “34”. When a range of elements is specified, such as, elements 42 through 56, the *NR* contains *NR.Start* = 42 and *NR.End* = 56. The string representation is “42:56”.

For multi-dimensional arrays each *NR* is separated by a comma (,). Dimensions are specified in the same order they appear in the *ArrayDimensions* attribute. A complete *NumericRange* for a three-dimensional array would be represented: “1:2,3,4:10”.

White spaces are not allowed in the string.

All indexes start with 0. The maximum value for any index is one less than the length of the dimension.

Equivalent Arrays from NumericRange

One *NR* defines a 1D array and a list of *NR* defines an array with multiple dimensions. The size of each

dimension is calculated by $size = end - start + 1$ and the size of the overall array is the size of each dimension multiplied together.

This can be used to validate the constructed *NR* to ensure it does not read or write more elements than what you intended. For a simple validation test, compare the number of elements you intend to write to the calculated size of the array that the *NR* defines – if the comparison doesn't match then an error has occurred, the operation should be aborted and the *NR* should not be used.

Server Failure Codes Related to Arrays

When reading a value, if the lower bound is out of range the server should return error code *Bad_IndexRangeNoData*.

When reading a value, if the upper bound is out of range the server can return *partial results* and the overall status code is *Bad_IndexRangeNoData*. *Partial results* mean that *the data returned will be less than what was expected*. The elements that are in range are returned and the elements that are out of range are missing. Clients must be prepared to examine both the overall status code for the operation and the length of the returned array – when extracting data for an element do not exceed the length of the reply buffer.

If the syntax of *NumericRange* is invalid the server should return *Bad_IndexRangeInvalid*.

Other error codes may be returned – it depends on the operation requested and the server and should match the error codes specified by the operation. The client should be prepared to deal with both expected and unexpected error codes.

NumericRange Construction for 1 Dimension Arrays

Constructing a *NR* for a one-dimension array is straightforward.

For any given subset of elements construct an *NR* using the starting and ending offsets. Each *NR* gets bound to a *ReadValueId* or *WriteValue* for specific reads or writes, or to a *MonitoredItem* for subscribing for notifications to a subset of elements. Just remember you need a separate *NR* for each consecutive subset of elements.

For example, in *M[1000]* if you want to read elements 3 through 10 and 12 through 15 you must use two *ReadValueId*s, one with *NR* = "3:10" and the other with *NR*="12:15". You can send both *ReadValueId*s in the same read command – the Read Service takes an array of *ReadValueId* structures and each is a separate operation on the server.

NumericRange Construction for *n* Dimension Arrays

This construction has subtle pitfalls and the risks are high when writing data or acting on data read. The prime directive for reads is that the value returned from a read must be from the correct source with correct status. When the client wants the value from element 4 it must be that value and not from element 3, 5, or anywhere else.

In practice this means that the UA client can request more data than what is needed – the elements can be picked out of the reply buffer and the rest ignored.

The prime directive for writes is more stringent – only the intended targets are to be written, no other elements should be changed. There is no way of telling the server to ignore elements so the data written must contain only those elements that are to be written, and the Indexes used must specify only those elements.

Why is this hard you ask? Because it is a lot harder to correctly specify a subset of a matrix than it appears - it might be time to take a refresher class on matrix math.

An *NR* does more than specify a subset of elements, it also defines a new array because a subset of elements is an array which contains only those elements.

For example, a matrix $M(10,10,10)$ contains 1,000 elements (multiply all the dimensions together). An index range of “3:4,2:4, 3:5” defines a matrix $R(2,3,3)$ and contains 18 elements. It will read or write elements (3,2,3), (3,2,4), (3,2,5), (3,3,3), (3,3,4), (3,3,5), (3,4,3), (3,4,4), (3,4,5), (4,2,3), (4,2,4), (4,2,5), (4,3,3), (4,3,4), (3,3,5), (3,4,3), (3,4,4), (3,4,5). Easy, right? It's not.

Indexed Reading – *Imprecise Mode*

Reading can be imprecise and still do little harm other than waste some network bandwidth. Writing can never be imprecise, and the technique described here should not be used for writes. “Imprecise” in this case means that even if there are gaps between the elements you want, you can read all the values with a single *NumericRange* that starts at the first element and ends at the last element.

For example, if you want data from $M(10,10)$ for elements (2,4), (3,5) and (6,2), you can use $NR = “2:6, 2:5”$. This *NR* defines a matrix of size (5,4) which contains 20 elements. The server will return an array that contains 20 values, and the client must extract from it the 3 values it wants.

To extract the values the client must calculate the offset into the buffer for each element using some pre-processing of the array location to determine the locations. For each element subtract the *NR* starting offset for each dimension from the element's location to create a relative location within the buffer.

This pseudo-C code illustrates the technique:

ConstructRelativeLocation:

```
for ( dimension = 0; dimension < ArrayDimensions; dimensions++ )
    relativeLocation = elementLocation[ dimension ] - NR[ dimension
].start
    relativeArray.Add(relativeLocation)
return relativeArray
```

Pass *relativeArray* to **LocationToFlatOffset** to get to the actual offset within the reply buffer.

This technique can be used to reduce the number of individual read operations that the server must perform at the expense of wasted bandwidth and larger buffers. How large is too large? There's no answer that is always correct - it depends on too many factors. At some point it should be broken up into smaller subsets but where to draw that line depends on many factors: your product, the system it is in, products it interacts with, and so on.

If this mode is unsuitable for your system and exact elements must be extracted, then *Precise* mode should be used.

Indexed Reading and Writing– *Precise Mode*

Precise means exactly that – the *NR* specifies only those elements it wants, and no other elements are included.

The prime directive here is that the ranges must be absolutely correct - this is more difficult than it appears. To be contiguous all elements must be adjacent, either within all inner dimensions or within a single dimension. Adjacency is required because *NRs* do not support all forms of *dimension overflow* (my term). In fact, it only supports a single type of *dimension overflow*.

This author defines Dimension Overflow as occurring when a dimension's offset is incremented such that the next outer (lower rank) dimension's offset increments and the inner (higher rank) dimension's offset

resets to 0.

For example, in $M[4,6,5]$ this occurs when traversing from element (2, 2, 4) to (2, 3, 0). The innermost dimension has 5 elements and the last offset within that dimension is 4.

If you wanted to write to only elements (2, 2, 4) and (2, 3, 0) it would be incorrect to define the *NR* as “2, 2:3, 4:0”.

This definition is incorrect because the third part, “4:0” specifies a starting element that is larger than the last offset. This violates the OPC UA Specification so either the toolkit or the server will return an error.

Reversing the definition will not work either. *NR* = “2,2:3,0:4” is syntactically correct but semantically incorrect – it specifies an array of size $(1 * 2 * 5) = 10$ elements, not the 2 elements you want. This *NR* specifies elements (2,2,0), (2,2,1), (2,2,2), (2,2,3), (2,2,4), (2,3,0), (2,3,1), (2,3,2), (2,3,3), (2,3,4) and affects elements other than the ones intended. Even worse, depending on the technology, language and libraries you use, it may get data from random places in the client’s memory and write that to unintended places in the server – alarms will not sound and you will not be warned, but it will not work correctly. Hopefully the toolkit or server you are using will reject this definition or the effect will be harmless, but you may have just sent a forklift through a wall.

***TIP:** One way to validate a constructed NR is to calculate the size of the matrix it defines and compare it to the number of elements you want to access – if they are not exactly equal then the NR definition is not correctly constructed.*

One way to avoid mistakes is to construct a separate *NR* for each element, but unless you are reading/writing to a single element this is also the least efficient – each *NR* is acted on as a separate operation by the server, including validation and verification of the *NR*, moving data around, replying to the request, and so on. If you did this for a large number of elements it might take an unreasonably long time, cause operation timeouts, and have other undesirable side-effects.

The best approach is to optimize the construction of *NRs* so that it uses the fewest number of *NRs* and operations that will still be precise. After analyzing *NumericRanges* and how they are applied I developed these rules for constructing optimized *NRs* for an arbitrary subset of elements.

- **All elements in an NR must be adjacent (with one exception – see *Adjacent across a single dimension*)**

When incrementing offsets within a dimension, elements are numerically adjacent only in the innermost (highest rank) dimension. For example, in $M[4,6,5]$, elements (2,3,1) and (2,3,2) are adjacent, but elements (2,3,1) and (2,4,1) are not adjacent. It is possible to optimize across other dimensions - see rule **Elements must be adjacent across a single dimension**.

- ***Dimension Overflow* is adjacent if and only if all the elements in the inner dimensions are included**

This means that when wrapping dimension boundaries all the inner dimensions (to the right of the wrapped dimension) must be intended to be included in the *NR*. If this is not suitable then you need another *NR*. For example, in $M[4,6,5]$, to wrap from element (2,3,x) to (2,4,x) requires that all the elements in dimension 2 are included in the *NR*. This includes elements (2,3,0) through (2,3,4) and (2,4,0) through (2,4,4). This defines an *NR* = “2,3:4,0:4”, which is matrix [1,2,5] with 10 elements. Using the same source matrix, to wrap from elements (1,5,x) to (2,0,x) requires that all the elements in both dimensions 1 and 2 are included in the *NR*. This specifies an *NR* = “1:2,0:5,0:4”, which defines a matrix [2,6,5] which contains 60 elements.

- **Elements must be adjacent within a single dimension**

Elements in dimensions other than the highest rank (innermost) are adjacent within a dimension if and only if all the other dimension's offsets are exactly equal. For example, elements (1,4,2) and (1,5,2) can be specified in a single *NR* = "1,4:5,2". This defines a matrix of size (1,2,1) which contains 2 elements. This is valid and will work correctly. Elements (1,4,2) and (1,5,3) cannot be specified in a single *NR* because there are two dimensions that are changing. If a single *NR* were used it would be *NR*="1, 4:5, 2:3", which is a matrix of size [1,2,2] and contains 4 elements, not the 2 elements expected.

Monitoring Subsets of Elements in an Array

When creating a subscription and monitoring nodes a client can optionally include an *NR* with the request.

If the *MonitoredItem* does not include an *NR*, then when any element in the array changes the server sends a notification with the entire array as the data. This works great for relatively small arrays, but with large arrays it may degrade performance and might create other problems (timeouts, etc.).

If the server supports Indexing, then a client can specify an *NR* when establishing the *MonitoredItem*. The server monitors only the range of elements specified and send a notification with the data just from the specified elements.

Some differences between using and not using an *NR* with *MonitoredItems* are:

- When a data notification is sent for a *MonitoredItem* that was created without an *NR* then the entire array is sent with each notification. That makes it possible for the client to determine the actual size of the array with each notification.
 - The buffer contents start with the first element and ends with the last element.
- When a data notification is sent for a *MonitoredItem* that was created with an *NR* then a subset of the array is sent, and it is not possible for the client to determine the actual size of the entire array.
 - The buffer starts with the first specified element, ends with the last, and only includes elements that were specified.
- Some toolkits may define a method that allows a client to "read" a *MonitoredItem* that was created with an *NR*. This is a convenience provided by the toolkit as there is no service that "reads" a monitored item. The toolkit is actually using the Read Service using the *NR* that is part of the *MonitoredItem*. It will produce the same results as directly invoking the Read Service with the same *NR*.

Possible Issues when Using Index Range

The technology and toolkits you use will influence the degree to which you need to use caution. Higher level languages, such as C# and Java, offer built-in type safety and range validation, and probably more error checking, so that it is less likely to encounter problems due to a miscalculated index range. C++ and C work directly with low-level pointers and so are subject to buffer overruns – a miscalculation can go undetected and data can be read/written from or to unintended locations.

For example, some APIs for writing matrices take a pointer to a buffer and the *NR*. If there is a miscalculation due to a Dimension Overflow it may index past the end of the buffer into random data, either on the heap or the stack, and result in writing incorrect data to the PLC.

Thoroughly test all operations during development.

Developers creating UA servers should test it using clients written by others, otherwise it may validate

your misconceptions about how a client will interact with it. Developers creating UA clients should test with as many servers as they can. OPC UA Foundation interop workshops are an ideal venue for this. If your server will be used to contain large arrays then indexing mode is the only practical way to interact with it, so it should be supported.

Runtime Considerations

An adaptive client may try to recover from errors or missing features in a server by attempting the same operation using different or less powerful techniques. For clients that are working with OPC UA array data if the server does not support indexing then the only other way for the client to read/write array data is *Read-Modify-Write (RMW)*.

For reads *RMW* is always “safe” but may not be practical due to array size, and *RMW* is not “safe” for writes because of the risk of data loss. Due to these constraints, implementing automatic client error recovery by switching the data access mode from indexed to *RMW* is not recommended for general purpose clients.

Testing Observations on the Consistency and Quality of Servers

During testing of our OPC UA client some servers supported indexing mode, and some did not. One embedded server was especially problematic in that it supported indexing mode incorrectly – it reversed the inner and outer dimensions, so all the reads and writes were in the wrong elements.

Perhaps one reason why some servers do not support indexing at all, and others may get it wrong is the lack of clients that use this technique – this makes testing the server difficult. Demo clients used during testing all used *RMW* mode.

Another interesting fact found during testing was that some servers allow the client to change the size of the array. The two demo servers used (Softing and UnifiedAutomation) supported this and the UnifiedAutomation demo browser, *UAExpert*, supported this. In our tests this only worked when the *NR* was not specified in the write.

Some servers sit on top of the actual data source, such as a controller. If the data source exposed an actual array of values some of the desktop servers would expose them as OPC arrays and others converted each element into a separate node (not an array), and some embedded servers did both.

The downside of treating each element as a separate node is scalability, both in memory consumption and performance. More nodes consume more memory in both the client and the server. More nodes require more operations to read the same data – a client can read the entire array in one operation but must read each node as a separate operation.

Some demo servers change the size of the array dynamically - each time the client reads the data the size of the array is different. This was great for testing and validating the client but not as effective for getting a steady stream of changing values.

Development and Testing Practices

Due to the lack of sample code and UA clients that read and write arrays using indexing it was difficult to determine if the client being written was correct – there was nothing to compare it to. Since writing to a

PLC is potentially dangerous, we used extensive internal tests and where necessary wrote custom test tools to validate the operations.

Internal Tests

During development the logic that creates the *NumericRange* indexes was separated from external interaction logic so that it could be thoroughly unit tested. Unit tests are code modules that directly call code in the production system. These are great for basic validation and for testing edge and failure cases that are otherwise difficult to test.

Tools

WireShark is a tool that captures packets on a network and can dissect the OPC UA protocol. It is currently the best source of information for how the client is interacting with the server, and it's free! This tool is invaluable in understanding what is really going on between the client and the server.

One of the custom tools I wrote cycles through multiple sets of elements, each time using a different set of elements that requires different *NRs*. In each pass it sets all the elements in the array to a known value (e.g. 0), writes new values to the selected elements, reads back the entire array, and then verifies that all the elements are set to the correct value, including those that should have been left alone. It stops when it detects an anomaly. This can be used for both *RMW* and Indexed mode testing.

Another tool I wrote is a generic client that connects to any UA server and provides independent evaluation of array support in any UA server. It is written in *.NET* (requires Windows) and runs completely independently of any products. It can set up a *Subscription* and *MonitoredItems* for the entire array (*RMW*) and for any subset of the array using Indexed operations. Multiple *MonitoredItems* can be created for the same node so you can see how the system behaves. This tool includes an element picker to select the subsets to observe.

The use of these tools does not eliminate the need for thorough functional testing.

Conclusion and Final Thoughts





A control system's primary job is to control a process with hard real-time determinism. This requires knowing how long it will take to execute operations, so the control system needs advance knowledge of types and sizes.

Clients that map into these systems have a similar need – when it connects a control system variable to an OPC UA node's data the client must know that the operation is compatible and safe. I hope that the techniques discussed here help you achieve these goals.

PLEASE CONTACT THE FOLLOWING PEOPLE FOR MORE INFORMATION.

Rockwell Automation
David A. Levine
Senior Software Engineer
dalevine@ra.rockwell.com

Rockwell Automation
Rosanna (Dyer) Navarro
Engineering Team Lead
rnavarro@ra.rockwell.com

Connect with us.    

rockwellautomation.com — expanding **human possibility**™

AMERICAS: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444

EUROPE/MIDDLE EAST/AFRICA: Rockwell Automation NV, Pegasus Park, De Kleetlaan 12a, 1831 Diegem, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640

ASIA PACIFIC: Rockwell Automation, Level 14, Core F, Cyberport 3, 100 Cyberport Road, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Rockwell Automation is a trademark of Rockwell Automation, Inc. Trademarks not belonging to Rockwell Automation are property of their respective companies.

Publication OPCUA-WP001A-EN-P — December 2019
Copyright © 2019 Rockwell Automation, Inc. All rights reserved. Printed in USA.