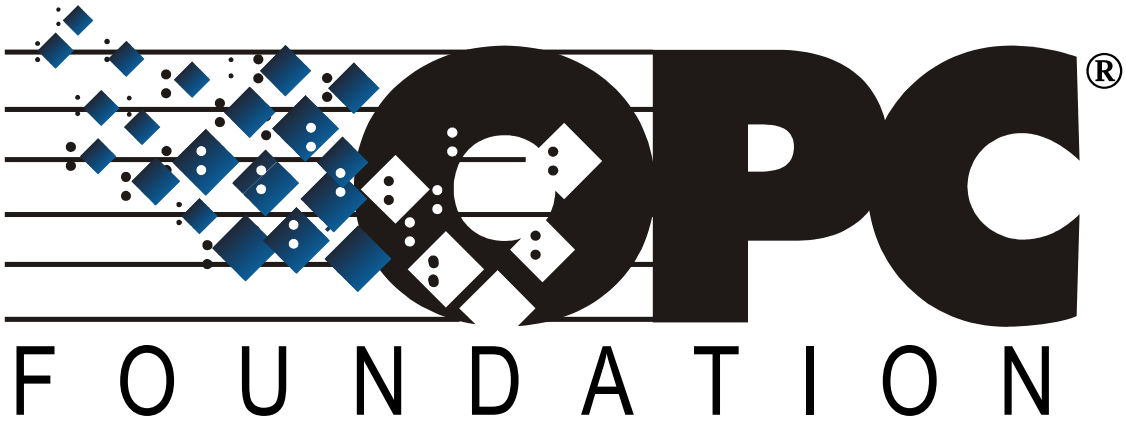

OPC Unified Architecture

Specification

Part 11: Historical Access

Release 1.03

December 11, 2015

Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture Historical Access	Date:	December 11, 2015
Version:	Release 1.03	Software Source:	MS-Word OPC UA Part 11 - Historical Access 1.03 Specification.docx
Author:	OPC Foundation	Status:	Release

CONTENTS

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

This specification is for developers of OPC UA clients and servers. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

AGREEMENT OF USE

ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: https://opcfoundation.org/developer-tools/specifications-unified-architecture/errata/.

Revision 1.03 Highlights

The following table includes the Mantis issues resolved with this revision.

Mantis ID	Summary	Resolution
2373	IEC comment: Is A.1 normative? Or informative?	Added informative identifier to Annex A.
2425	HistoricalDataConfigurationType: AggregateFunctions is Optional, should be Mandatory?	The text has been changed to remove the requirement for an empty folder when aggregates do not exist.
2431	In Subclause 6.4.3.2 what is "format"?	“Format” has been replaced with the correct term “TimestampToReturn”.
2444	UpdateEventDetails only allows one event message	UpdateEventDetails has been changed to allow more than one event message.
2445	Timestamps: Client requests TimestampsToReturn.Both, what if server supports Source only?	Explicitly stated that the server shall return an error when the TimestampsToReturn Both selection is used and the server only supports the source timestamp.
2446	6.4.3.3 Read Modified Functionality: Para 1 is confusing; should original value be returned?	Added that only modified values are returned and added a link to where the upateTypes are defined.
2469	ServerTimestamp support, possibly make available as a property for clients to discover?	Changed HistoryServerCapabilitiesType to have the optional global indication. Changed historicalDataConfigurationType to have the optional individual indication. Added descriptions for each.
2471	Validation on inserting records exceeding the bounds of the database	Text was added stating what result codes are returned when the vendor defined limits are exceeded.
2477	ReadAtTime when SimpleBounds=TRUE/FALSE	Text was added to clarify the useSimpleBounds parameter.
2517	"Subtype of" for "Inherit the properties"	Text changed to "Subtype of".
2531	Uniqueness of Annotation records - do not include message for uniqueness	Removed message from the formula for uniqueness of the structured data key.
2584	6.8.7.2 DeleteAtTimeDetails "all entries" clarification	The text “all entries” was replaced with “all raw values, modified values, and annotations”
2667	Description of Continuation Point (6.3) does not match general concept	Changed the continuation point text to be correct with how continuation points are used in the rest of the UA specification.
2784	What does 'ProcessingInterval=0' mean	Added clarification text subclause 6.4.4.2 describing the meaning of the value 0.
2887	Continuation point handling for large HistoryRead requests	Continuation points have been added as valid replies for all read requests when the time needed to reply is longer than the timeout hint.
2985	Typo in 5.6.4 AuditHistoryDeleteEventType	Incorrect “NodeID” was changed to “UpdateNode” to match the property name.
3091	OperationLimitsType - MaxNodesPerHistoryUpdateXxx issues	The text was clarified when both Events and Data are included in the same call when using the UpdateHistory Service.
3236	Aggregate Configuration type missing from Historical capabilities	Added an instance of AggreagteConfigurationType to the HistoricalServerCapabilities.

OPC Unified Architecture

Part 11: Historical Access

1 Scope

This specification is part of the overall OPC Unified Architecture standard series and defines the *information model* associated with Historical Access (HA). It particularly includes additional and complementary descriptions of the *NodeClass*es and *Attributes* needed for Historical Access, additional standard *Properties*, and other information and behaviour.

The complete *AddressSpace* Model including all *NodeClass*es and *Attributes* is specified in Part 3. The predefined *Information Model* is defined in Part 5. The *Services* to detect and access historical data and events, and description of the *ExtensibleParameter* types are specified in Part 4.

This part includes functionality to compute and return *Aggregates* like minimum, maximum, average etc. The *Information Model* and the concrete working of *Aggregates* are defined in Part 13.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Part 1: OPC UA Specification: Part 1 – Overview and Concepts
http://www.opcfoundation.org/UA/Part1/

Part 3: OPC UA Specification: Part 3 – Address Space Model
http://www.opcfoundation.org/UA/Part3/

Part 4: OPC UA Specification: Part 4 – Services
http://www.opcfoundation.org/UA/Part4/

Part 5: OPC UA Specification: Part 5 – Information Model
http://www.opcfoundation.org/UA/Part5/

Part 6: OPC UA Specification: Part 6 – Mapping
http://www.opcfoundation.org/UA/Part6/

Part 7: OPC UA Specification: Part 7 – Profiles
http://www.opcfoundation.org/UA/Part7/

Part 8: OPC UA Specification: Part 8 – Data Access
http://www.opcfoundation.org/UA/Part8/

Part 13:OPC UA Specification: Part 13 – Aggregates
http://www.opcfoundation.org/UA/Part13/

3 Terms, definitions, and abbreviations

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in Part 1, Part 3, Part 4, and Part 13 as well as the following apply.

3.1.1
Annotation
metadata associated with an item at a given instance in time

Note 1 to entry: An *Annotation* is metadata that is associated with an item at a given instance in time. There does not have to be a value stored at that time.

3.1.2
BoundingValues
values associated with the starting and ending time

Note 1 to entry: *BoundingValues* are the values that are associated with the starting and ending time of a *ProcessingInterval* specified when reading from the historian. *BoundingValues* may be required by *Clients* to determine the starting and ending values when requesting *raw data* over a time range. If a *raw data* value exists at the start or end point, it is considered the bounding value even though it is part of the data request. If no *raw data* value exists at the start or end point, then the *Server* will determine the boundary value, which may require data from a data point outside of the requested range. See 4.4 for details on using *BoundingValues*.

3.1.3
HistoricalNode
Object, *Variable*, *Property* or *View* in the *AddressSpace* where a *Client* can access historical data or *Events*

Note 1 to entry: A *HistoricalNode* is a term used in this document to represent any *Object*, *Variable, Property* or *View* in the *AddressSpace* for which a *Client* may read and/or update historical data or *Events*. The terms "*HistoricalNode's* history" or "history of a *HistoricalNode*" will refer to the time series data or *Events* stored for this *HistoricalNode*. The term *HistoricalNode* refers to both *HistoricalDataNodes* and *HistoricalEventNodes*.

3.1.4
HistoricalDataNode
Variable or *Property* in the *AddressSpace* where a *Client* can access historical data

Note 1 to entry: A *HistoricalDataNode* represents any Variable or Property in the *AddressSpace* for which a *Client* may read and/or update historical data. "*HistoricalDataNode*'s history" or "history of a *HistoricalDataNode*" refers to the time series data stored for this *HistoricalNode*. Examples of such data are:

- device data (like temperature sensors)
- calculated data
- status information (open/closed, moving)
- dynamically changing system data (like stock quotes)
- diagnostic data

The term *HistoricalDataNodes* is used when referencing aspects of the standard that apply to accessing historical data only.

3.1.5
HistoricalEventNode
Object or *View* in the *AddressSpace* for which a *Client* can access historical *Events*

Note 1 to entry: "*HistoricalEventNode*'s history" or "history of a *HistoricalEventNode*" refers to the time series *Events* stored in some historical system. Examples of such data are:

- *Notifications*
- system *Alarms*
- operator action *Events*
- system triggers (such as new orders to be processed)

The term *HistoricalEventNode* is used when referencing aspects of the standard that apply to accessing historical *Events* only.

3.1.6
modified values
HistoricalDataNode's value that has been changed (or manually inserted or deleted) after it was stored in the historian

Note 1 to entry: For some *Servers*, a lab data entry value is not a *modified value*, but if a user corrects a lab value, the original value would be considered a *modified value*, and would be returned during a request for *modified values*. Also manually inserting a value that was missed by a standard collection system may be considered a *modified value*. Unless specified otherwise, all historical *Services* operate on the current, or most recent, value for the specified *HistoricalDataNode* at the specified timestamp. Requests for *modified values* are used to access values that have been superseded, deleted or inserted. It is up to a system to determine what is

considered a *modified value*. Whenever a *Server* has modified data available for an entry in the historical collection it shall set the *ExtraData* bit in the *StatusCode.*

3.1.7
raw data
data that is stored within the historian for a *HistoricalDataNode*

Note 1 to entry: The data may be all data collected for the *DataValue* or it may be some subset of the data depending on the historian and the storage rules invoked when the item's values were saved.

3.1.8
StartTime/EndTime
bounds of a history request which define the time domain

Note 1 to entry: For all requests, a value falling at the end time of the time domain is not included in the domain, so that requests made for successive, contiguous time domains will include every value in the historical collection exactly once.

3.1.9
TimeDomain
interval of time covered by a particular request, or response

Note 1 to entry: In general, if the start time is earlier than or the same as the end time, the time domain is considered to begin at the start time and end just before the end time; if the end time is earlier than the start time, the time domain still begins at the start time and ends just before the end time, with time "running backward" for the particular request and response. In both cases, any value which falls exactly at the end time of the *TimeDomain* is not included in the *TimeDomain*. See the examples in 4.4. *BoundingValues* effect the time domain as described in 4.4.

All timestamps which can legally be represented in a *UtcTime DataType* are valid timestamps, and the *Server* may not return an invalid argument result code due to the timestamp being outside of the range for which the *Server* has data. See Part 3 for a description of the range and granularity of this *DataType*. *Servers* are expected to handle out-of-bounds timestamps gracefully, and return the proper *StatusCodes* to the *Client.*

3.1.10
Structured History Data
structured data stored in a history collection where parts of the structure are used to uniquely identify the data within the data collection

Note 1 to entry: Most historical data applications assume only one current value per timestamp. Therefore the timestamp of the data is considered the unique identifier for that value. Some data or meta data such as *Annotations* may permit multiple values to exist at a single timestamp. In such cases the *Server* would use one or more parameters of the *Structured History Data* entry to uniquely identifiy each element within the history collection. *Annotations* are examples of *Structured History Data*.

3.2 Abbreviations

DA Data Access
HA Historical Access
HDA Historical Data Access
UA Unified Architecture

4 Concepts

4.1 General

This part defines the handling of historical time series data and historical *Event* data in the OPC Unified Architecture. Included is the specification of the representation of historical data and *Events* in the *AddressSpace.*

4.2 Data architecture

A *Server* supporting Historical Access provides *Clients* with transparent access to different historical data and/or historical *Event* sources (e.g. process historians, event historians, etc.).

The historical data or *Events* may be located in a proprietary data collection, database or a short term buffer within the memory. A *Server* supporting Historical Access will provide historical data and *Events* for all or a subset of the available *Variables*, *Objects, Properties* or *Views* within the *Server AddressSpace*.

Figure 1 illustrates how the *AddressSpace* of a UA *Server* might consist of a broad range of different historical data and/or historical *Event* sources.

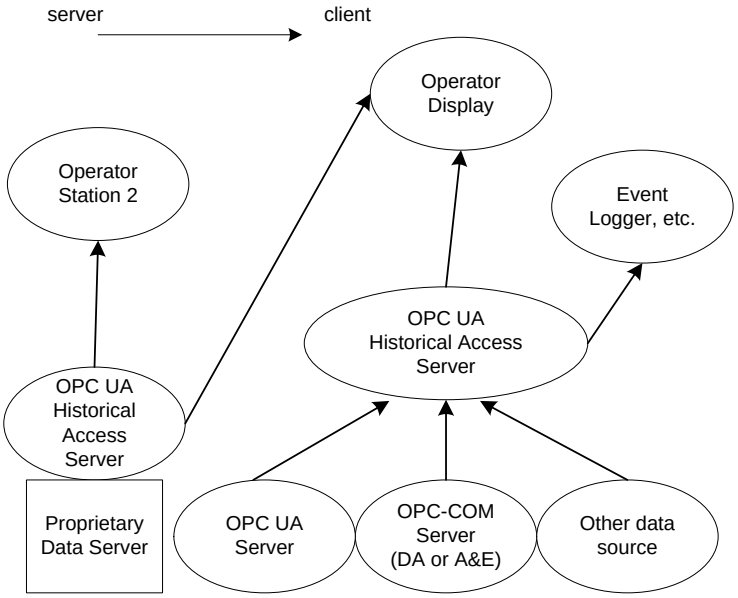

Figure 1 – Possible OPC UA Server supporting Historical Access

The *Server* may be implemented as a standalone OPC UA *Server* that collects data from another OPC UA *Server* or another data source. The *Client* that references the OPC UA *Server* supporting Historical Access for historical data may be simple trending packages that just desire values over a given time frame or they may be complex reports that require data in multiple formats.

4.3 Timestamps

The nature of OPC UA Historical Access requires that a single timestamp reference be used to relate the multiple data points, and the *Client* may request which timestamp will be used as the reference. See Part 4 for details on the *TimestampsToReturn* enumeration. An OPC UA *Server* supporting Historical Access will treat the various timestamp settings as described below. A HistoryRead with invalid settings will be rejected with Bad_TimestampsToReturnInvalid (see Part 4).

For *HistoricalDataNodes*, the SourceTimestamp is used to determine which historical data values are to be returned.

The request is in terms of *SourceTimestamp* but the reply could be in *SourceTimestamp*, *ServerTimestamp* or both timestamps. If the reply has the *Server* timestamp the timestamps could fall outside of the range of the requested time.

SOURCE_0 Return the *SourceTimestamp*.

SERVER_1 Return the *ServerTimestamp*.

BOTH_2 Return both the *SourceTimestamp* and *ServerTimestamp*.

NEITHER_3 This is not a valid setting for any HistoryRead accessing *HistoricalDataNodes.*

Any reference to timestamps in this context throughout this part will represent either *ServerTimestamp* or *SourceTimestamp* as dictated by the type requested in the HistoryRead *Service*. Some *Servers* may not support historizing both *SourceTimestamp* and *ServerTimestamp*, but it is expected that all *Servers* will support historizing *SourceTimestamp* (see Part 7 for details on *Server Profiles*).

If a request is made requesting both *ServerTimestamp* and *SourceTimestamp* and the *Server* is only collecting the *SourceTimestamp* the *Server* shall return Bad_TimestampsToReturnInvalid.

For *HistoricalEventNodes* this parameter does not apply. This parameter is ignored since the entries returned are dictated by the *Event* Filter. See Part 4 for details.

4.4 Bounding Values and time domain

When accessing *HistoricalDataNodes via* the *HistoryRead Service*, requests can set a flag, returnBounds, indicating that *BoundingValues* are requested. For a complete description of the *Extensible Parameter HistoryReadDetails* that include *StartTime*, *EndTime* and NumValuesPerNode, see 6.4. The concept of Bounding Values and how they affect the time domain that is requested as part of the *HistoryRead* request is further explained in 4.4. 4.4 also provides examples of *TimeDomains* to further illustrate the expected behaviour.

When making a request for historical data using the *HistoryRead Service*, the required parameters include at least 2 of these three parameters: *startTime*, *endTime* and numValuesPerNode. What is returned when Bounding Values are requested varies according to which of these parameters are provided. For a historian that has values stored at 5:00, 5:02, 5:03, 5:05 and 5:06, the data returned when using the Read *Raw* functionality is given by Table 1. In the table, FIRST stands for a tuple with a value of null, a timestamp of the specified *StartTime*, and a *StatusCode* of *Bad_BoundNotFound*. LAST stands for a tuple with a value of null, a timestamp of the specified *EndTime*, and a *StatusCode* of *Bad_BoundNotFound.*

In some cases, attempting to locate bounds, particularly FIRST or LAST points, may be resource intensive for *Servers*. Therefore how far back or forward to look in history for Bounding Values is *Server* dependent, and the *Server* search limits may be reached before a bounding value can be found. There are also cases, such as reading *Annotations* or *Attribute* data where Bounding Values may not be appropriate. For such use cases it is permissible for the *Server* to return a *StatusCode* of *Bad_BoundNotSupported.*

Table 1 – Bounding Value examples

Start Time	End Time	numValuesPerNode	Bounds	Data Returned
5:00	5:05	0	Yes	5:00, 5:02, 5:03, 5:05
5:00	5:05	0	No	5:00, 5:02, 5:03
5:01	5:04	0	Yes	5:00, 5:02, 5:03, 5:05
5:01	5:04	0	No	5:02, 5:03
5:05	5:00	0	Yes	5:05, 5:03, 5:02, 5:00
5:05	5:00	0	No	5:05, 5:03, 5:02
5:04	5:01	0	Yes	5:05, 5:03, 5:02, 5:00
5:04	5:01	0	No	5:03, 5:02
4:59	5:05	0	Yes	FIRST, 5:00, 5:02, 5:03, 5:05
4:59	5:05	0	No	5:00, 5:02, 5:03
5:01	5:07	0	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST
5:01	5:07	0	No	5:02, 5:03, 5:05, 5:06
5:00	5:05	3	Yes	5:00, 5:02, 5:03
5:00	5:05	3	No	5:00, 5:02, 5:03
5:01	5:04	3	Yes	5:00, 5:02, 5:03
5:01	5:04	3	No	5:02, 5:03
5:05	5:00	3	Yes	5:05, 5:03, 5:02
5:05	5:00	3	No	5:05, 5:03, 5:02
5:04	5:01	3	Yes	5:05, 5:03, 5:02
5:04	5:01	3	No	5:03, 5:02
4:59	5:05	3	Yes	FIRST, 5:00, 5:02
4:59	5:05	3	No	5:00, 5:02, 5:03
5:01	5:07	3	Yes	5:00, 5:02, 5:03
5:01	5:07	3	No	5:02, 5:03, 5:05
5:00	UNSPECIFIED	3	Yes	5:00, 5:02, 5:03

Start Time	End Time	numValuesPerNode	Bounds	Data Returned
5:00	UNSPECIFIED	3	No	5:00, 5:02, 5:03
5:00	UNSPECIFIED	6	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST[a]
5:00	UNSPECIFIED	6	No	5:00, 5:02, 5:03, 5:05, 5:06
5:07	UNSPECIFIED	6	Yes	5:06, LAST
5:07	UNSPECIFIED	6	No	NODATA
UNSPECIFIED	5:06	3	Yes	5:06,5:05,5:03
UNSPECIFIED	5:06	3	No	5:06,5:05,5:03
UNSPECIFIED	5:06	6	Yes	5:06,5:05,5:03,5:02,5:00,FIRST[b]
UNSPECIFIED	5:06	6	No	5:06, 5:05, 5:03, 5:02, 5:00
UNSPECIFIED	4:48	6	Yes	5:00, FIRST
UNSPECIFIED	4:48	6	No	NODATA
4:48	4:48	0	Yes	FIRST,5:00
4:48	4:48	0	No	NODATA
4:48	4:48	1	Yes	FIRST
4:48	4:48	1	No	NODATA
4:48	4:48	2	Yes	FIRST,5:00
5:00	5:00	0	Yes	5:00,5:02[c]
5:00	5:00	0	No	5:00
5:00	5:00	1	Yes	5:00
5:00	5:00	1	No	5:00
5:01	5:01	0	Yes	5:00, 5:02
5:01	5:01	0	No	NODATA
5:01	5:01	1	Yes	5:00
5:01	5:01	1	No	NODATA

a The timestamp of LAST cannot be the specified End Time because there is no specified End Time. In this situation the timestamp for LAST will be equal to the previous timestamp returned plus one second.

b The timestamp of FIRST cannot be the specified End Time because there is no specified Start Time. In this situation the timestamp for FIRST will be equal to the previous timestamp returned minus one second.

c When the Start Time = End Time (there is data at that time), and Bounds is set to True, the start bounds will equal the Start Time and the next data point will be used for the end bounds.

4.5 Changes in AddressSpace over time

Clients use the browse *Services* of the *View Service Set* to navigate through the *AddressSpace* to discover the *HistoricalNodes* and their characteristics. These *Services* provide the most current information about the *AddressSpace*. It is possible and probable that the *AddressSpace* of a *Server* will change over time (i.e. *TypeDefinitions* may change; *NodeIds* may be modified, added or deleted).

Server developers and administrators need to be aware that modifying the *AddressSpace* may impact a *Client's* ability to access historical information. If the history for a *HistoricalNode* is still required, but the *HistoricalNode* is no longer historized, then the *Object* should be maintained in the AddressSpace, with the appropriate *AccessLevel Attribute* and *Historizing Attribute* settings (see Part 3 for details on access levels).

5 Historical Information Model

5.1 HistoricalNodes

5.1.1 General

The Historical Access model defines additional *Properties* that are applicable for both *HistoricalDataNodes* and *HistoricalEventNodes.*

5.1.2 Annotations Property

The *DataVariable* or *Object* that has *Annotation* data will add the *Annotations Property* as shown in Table 2.

Table 2 – Annotations Property

Name	Use	Data Type	Description
Standard Properties			
Annotations	O	Annotation	The *Annotations Property* is used to indicate that *Annotation* data exists for the history collection exposed by a *HistoricalDataNode*. *Annotation DataType* is defined in5.5.

Since it is not allowed for *Properties* to have *Properties*, the *Annotation Property* is only available for *DataVariables* or *Objects*.

Not every *HistoricalDataNode* in the *AddressSpace* might contain *Annotation* data. The *Annotations Property* indicates whether or not a *HistoricalDataNode* supports *Annotations*. *Annotation* data is accessed using the standard *HistoryRead* functions. *Annotations* are modified, inserted or deleted using the standard *HistoryUpdate* functions.

As with all *HistoricalNodes*, modifications, deletions or additions of *Annotations* will raise the appropriate Historical Audit Event with the corresponding *NodeId*.

5.2 HistoricalDataNodes

5.2.1 General

The Historical Data model defines additional *ObjectTypes and Objects*. These descriptions also include required use cases for *HistoricalDataNodes.*

5.2.2 HistoricalDataConfigurationType

The Historical Access Data model extends the standard type model by defining the *HistoricalDataConfigurationType.* This *Object* defines the general characteristics of a *Node* that defines the historical configuration of any *HistoricalDataNode* that is defined to contain history. It is formally defined in Table 3.

All *Instances* of the *HistoricalDataConfigurationType* use the standard *BrowseName* as defined in Table 6.

Table 3 – HistoricalDataConfigurationType definition

Attribute	Value				
BrowseName	HistoricalDataConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	AggregateConfiguration	--	AggregateConfigurationType	Mandatory
HasComponent	Object	AggregateFunctions	--	FolderType	Optional
HasProperty	Variable	Stepped	Boolean	PropertyType	Mandatory
HasProperty	Variable	Definition	String	PropertyType	Optional
HasProperty	Variable	MaxTimeInterval	Duration	PropertyType	Optional
HasProperty	Variable	MinTimeInterval	Duration	PropertyType	Optional
HasProperty	Variable	ExceptionDeviation	Double	PropertyType	Optional
HasProperty	Variable	ExceptionDeviationFormat	Enum	PropertyType	Optional
HasProperty	Variable	StartOfArchive	UtcTime	PropertyType	Optional
HasProperty	Variable	StartOfOnlineArchive	UtcTime	PropertyType	Optional

HasProperty	Variable	ServerTimestampSupported	Boolean	PropertyType	Optional

AggregateConfiguration Object represents the browse entry point for information on how the *Server* treats *Aggregate* specific functionality such as handling *Uncertain data*. This *Object* is required to be present even if it contains no *Aggregate* configuration *Objects*. *Aggregates* are defined in Part 13.

AggregateFunctions is an entry point to browse to all *Aggregate* capabilities supported by the *Server* for Historical Access. All *HistoryAggregates* supported by the *Server* should be able to be browsed starting from this *Object*. *Aggregates* are defined in Part 13.

The *Stepped Variable* specifies whether the historical data was collected in such a manner that it should be displayed as *SlopedInterpolation* (sloped line between points) or as *SteppedInterpolation* (vertically-connected horizontal lines between points) when *raw data* is examined. This *Property* also effects how some *Aggregates* are calculated. A value of True indicates the stepped interpolation mode. A value of False indicates *SlopedInterpolation* mode. The default value is False.

The *Definition Variable* is a vendor-specific, human readable string that specifies how the value of this *HistoricalDataNode* is calculated. Definition is non-localized and will often contain an equation that can be parsed by certain *Clients*.

Example: *Definition*::= “(TempA – 25) + TempB”

The *MaxTimeInterval Variable* specifies the maximum interval between data points in the history repository regardless of their value change (see Part 3 for definition of *Duration*).

The *MinTimeInterval Variable* specifies the minimum interval between data points in the history repository regardless of their value change (see Part 3 for definition of *Duration*).

The *ExceptionDeviation Variable* specifies the minimum amount that the data for the *HistoricalDataNode* shall change in order for the change to be reported to the history database.

The *ExceptionDeviationFormat Variable* specifies how the ExceptionDeviation is determined. Its values are defined in Table 4.

The *StartOfArchive Variable* specifies the date before which there is no data in the archive either online or offline.

The *StartOfOnlineArchive Variable* specifies the date of the earliest data in the online archive.

The *ServerTimestampSupported Variable* indicates support for the *ServerTimestamp* capability. A value of True indicates the *Server* supports *ServerTimestamps* in addition to *SourceTimestamp*. The default is False.

Table 4 – ExceptionDeviationFormat Values

Value	Description
ABSOLUTE_VALUE_0	ExceptionDeviation is an absolute Value.
PERCENT_OF_VALUE_1	ExceptionDeviation is a percentage of Value.
PERCENT_OF_RANGE_2	ExceptionDeviation is a percentage of InstrumentRange (see Part 8).
PERCENT_OF_EU_RANGE_3	ExceptionDeviation is a percentage of EURange (see Part 8).
UNKNOWN_4	ExceptionDeviation type is Unknown or not specified.

5.2.3 HasHistoricalConfiguration ReferenceType

This *ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType* and will be used to refer from a Historical *Node* to one or more *HistoricalDataConfigurationType Objects*.

The semantic indicates that the target *Node* is "used" by the source *Node* of the *Reference*. Figure 2 informally describes the location of this *ReferenceType* in the OPC UA hierarchy. Its representation in the *AddressSpace* is specified in Table 5.

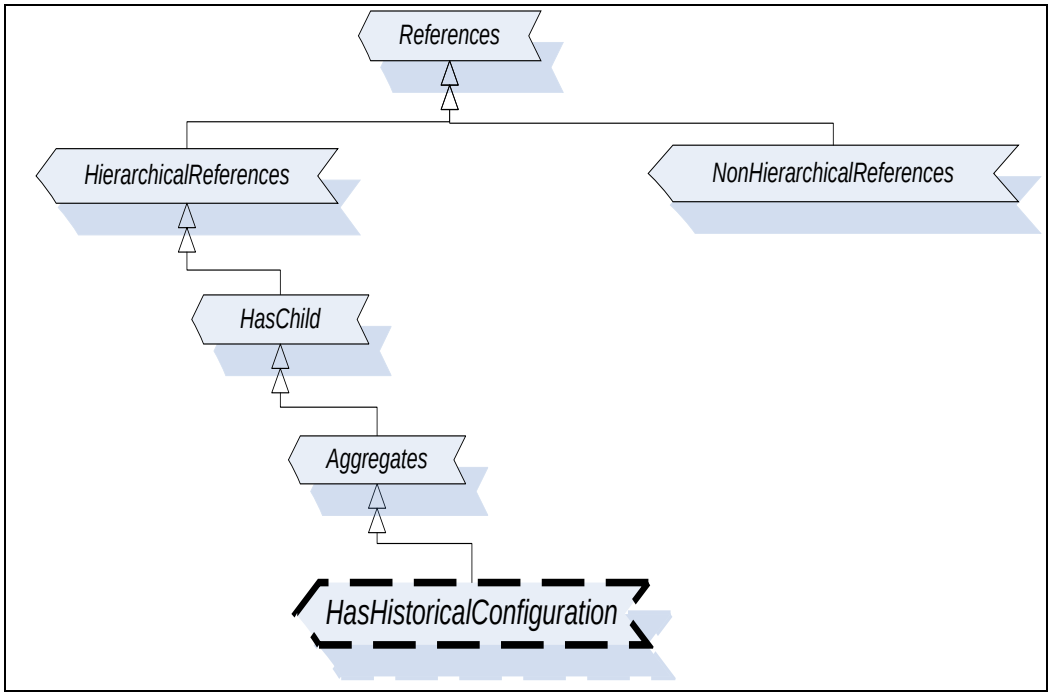

Figure 2 – ReferenceType hierarchy

Table 5 – HasHistoricalConfiguration ReferenceType

Attributes	**Value**		
BrowseName	HasHistoricalConfiguration		
InverseName	HistoricalConfigurationOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
The subtype of Aggregates *ReferenceType* is defined in Part 5.			

5.2.4 Historical Data Configuration Object

This *Object* is used as the browse entry point for information about *HistoricalDataNode* configuration. The content of this *Object* is already defined by its type definition in Table 3. It is formally defined in Table 6. If a *HistoricalDataNode* has configuration defined then one instance shall have a *BrowseName* of 'HA Configuration'. Additional configurations may be defined with different *BrowseNames*. All Historical Configuration *Objects* shall be referenced using the *HasHistoricalConfiguration ReferenceType*. It is also highly recommended that display names are chosen that clearly describe the historical configuration e.g. "1 Second Collection", "Long Term Configuration" etc.

Table 6 – Historical Access configuration definition

Attribute	**Value**				
BrowseName	HA Configuration				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	Object Type	HistoricalDataConfigurationType	Defined in Table 3		

5.2.5 HistoricalDataNodes Address Space Model

HistoricalDataNodes are always a part of other *Nodes* in the *AddressSpace.* They are never defined by themselves. A simple example of a container for *HistoricalDataNodes* would be a "Folder Object".

Figure 3 illustrates the basic *AddressSpace* Model of a *DataVariable* that includes History.

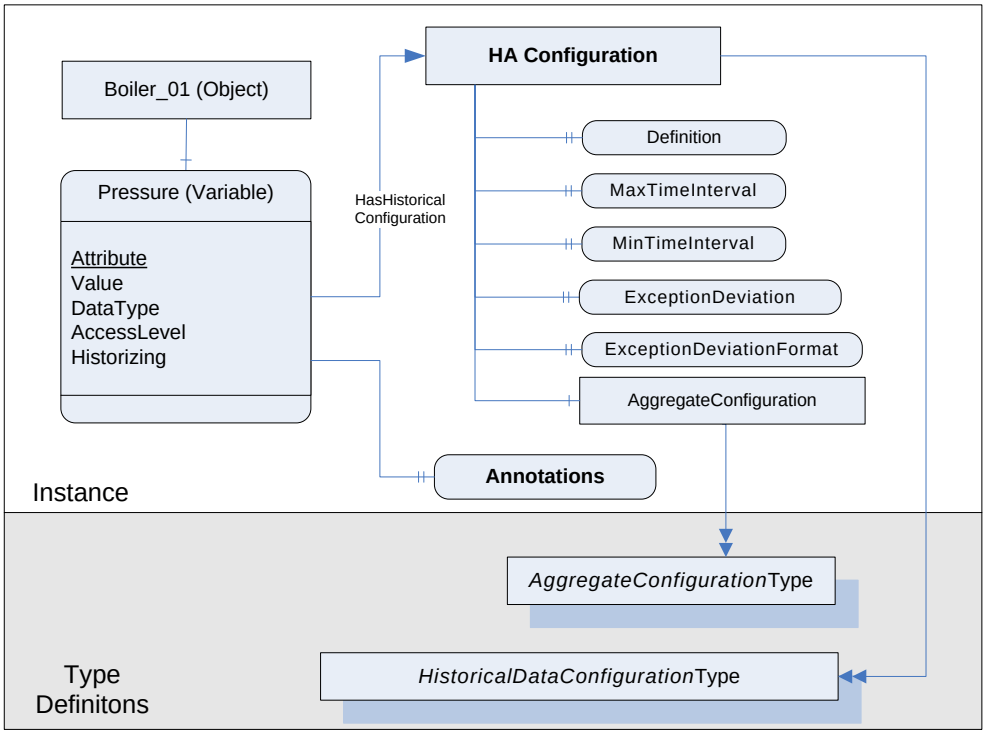

Figure 3 – Historical Variable with Historical Data Configuration and Annotations

Each *HistoricalDataNode* with history shall have the *Historizing Attribute* (see Part 3) defined and may reference a *HistoricalAccessConfiguration Object*. In the case where the HistoricalDataNode is itself a *Property* then the *HistoricalDataNode* inherits the values from the Parent of the *Property*.

Not every *Variable* in the *AddressSpace* might contain history data. To see if history data is available, a *Client* will look for the HistoryRead/Write states in the *AccessLevel Attribute* (see Part 3 for details on use of this *Attribute*).

Figure 3 only shows a subset of *Attributes* and *Properties*. Other *Attributes* that are defined for *Variables* in Part 3, may also be available.

5.2.6 Attributes

This part lists the *Attributes* of *Variables* that have particular importance for historical data. They are specified in detail in Part 3.

- AccessLevel
- Historizing

5.3 HistoricalEventNodes

5.3.1 General

The Historical *Event* model defines additional *Properties*. These descriptions also include required use cases for *HistoricalEventNodes.*

Historical Access of *Events* uses an *EventFilter*. It is important to understand the differences between applying an *EventFilter* to current *Event Notifications*, and historical *Event* retrieval.

In real time monitoring *Events* are received via *Notifications* when subscribing to an *EventNotifier*. The *EventFilter* provides the filtering and content selection of *Event Subscriptions*. If an *Event Notification* conforms to the filter defined by the *where* parameter of the *EventFilter*, then the *Notification* is sent to the *Client*.

In historical *Event* retrieval the *EventFilter* represents the filtering and content selection used to describe what parameters of *Events* are available in history. These may or may not include all of the parameters of the real-time *Event*, i.e. not all fields available when the *Event* was generated may have been stored in history.

The *HistoricalEventFilter* may change over time so a *Client* may specify any field for any *EventType* in the *EventFilter*. If a field is not stored in the historical collection then the field is set to null when it is referenced in the *selectClause* or the *whereClause*.

5.3.2 HistoricalEventFilter Property

A *HistoricalEventNode* that has *Event* history available will provide the *Property*. This *Property* is formally defined in Table 7.

Table 7 – Historical Events Properties

Name	Use	Data Type	Description
Standard Properties			
HistoricalEventFilter	M	EventFilter	A filter used by the *Server* to determine which *HistoricalEventNode* fields are available in history. It may also include a where clause that indicates the types of *Events* or restrictions on the *Events* that are available via the *HistoricalEventNode.* The *HistoricalEventFilter Property* can be used as a guideline for what *Event* fields the Historian is currently storing. But this field may have no bearing on what *Event* fields the Historian is capable of storing.

5.3.3 HistoricalEventNodes Address Space Model

HistoricalEventNodes are *Objects* or *Views* in the *AddressSpace* that expose historical *Events*. These *Nodes* are identified via the *EventNotifier Attribute*, and provide some historical subset of the *Events* generated by the *Server*.

Each *HistoricalEventNode* is represented by an *Object* or View with a specific set of Attributes. The *HistoricalEventFilter Property* specifies the fields available in the history.

Not every *Object* or *View* in the *AddressSpace* may be a *HistoricalEventNode*. To qualify as *HistoricalEventNodes*, a *Node* has to contain historical *Events*. To see if historical *Events* are available, a *Client* will look for the HistoryRead/Write states in the *EventNotifier Attribute*. See Part 3 for details on the use of this *Attribute.*

Figure 4 illustrates the basic *AddressSpace* Model of an *Event* that includes History.

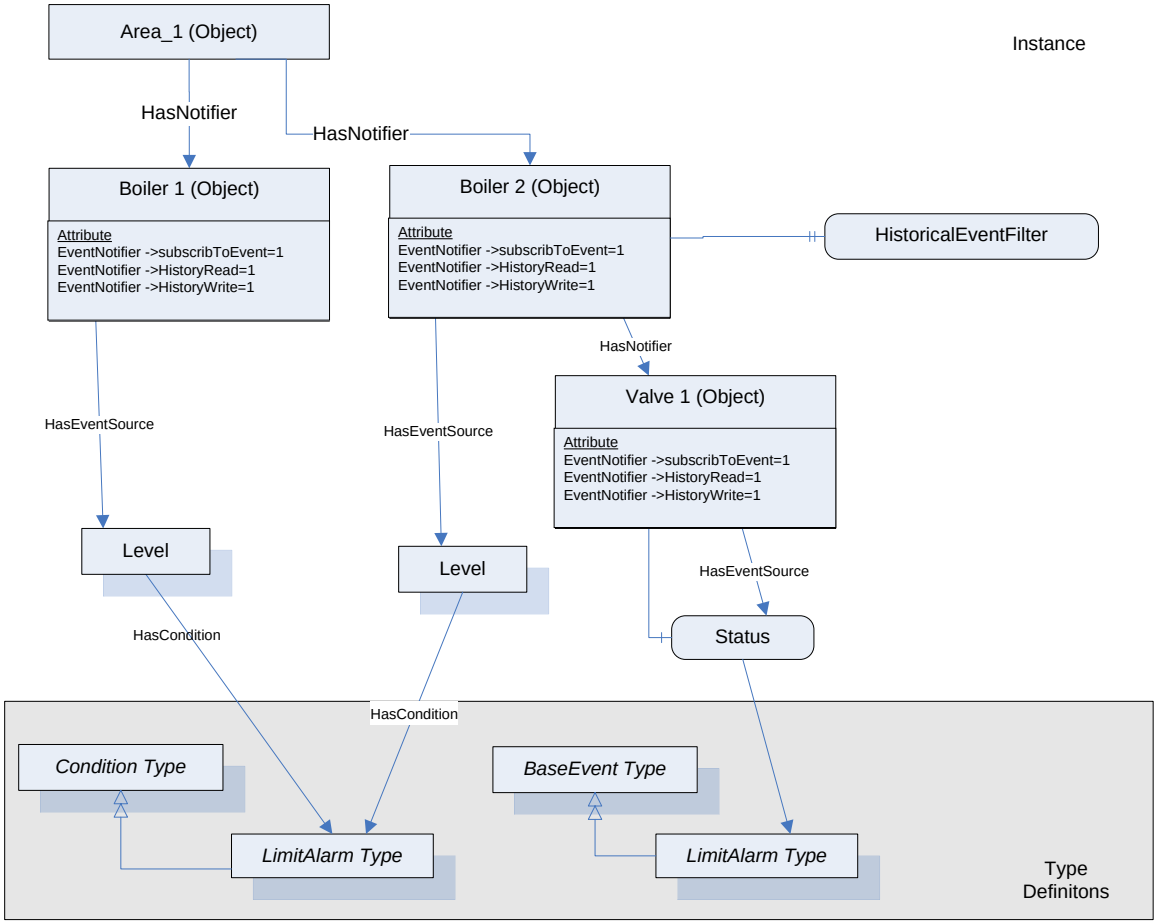

Figure 4 – Representation of an Event with History in the AddressSpace

5.3.4 HistoricalEventNodes Attributes

This part lists the *Attributes* of *Objects* or *Views* that have particular importance for historical *Events*. They are specified in detail in Part 3. The following *Attributes* are particularly important for *HistoricalEventNodes*.

- EventNotifier

The *EventNotifier Attribute* is used to indicate if the *Node* can be used to read and/or update historical *Events*.

5.4 Exposing supported functions and capabilities

5.4.1 General

OPC UA *Servers* can support several different functionalities and capabilities. The following standard *Objects* are used to expose these capabilities in a common fashion, and there are several standard defined concepts that can be extended by vendors. The *Objects* are outlined in Part 1.

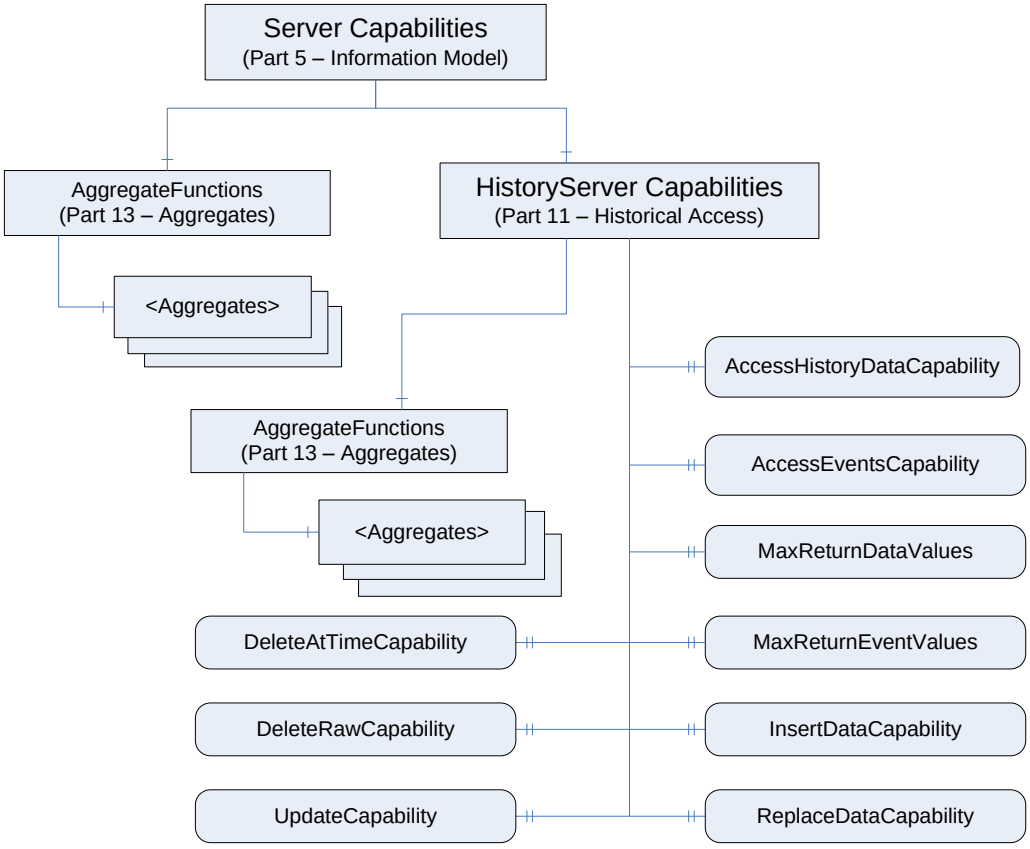

Figure 5 – Server and HistoryServer Capabilities

5.4.2 HistoryServerCapabilitiesType

The *ServerCapabilitiesType Objects* for any OPC UA *Server* supporting Historical Access shall contain a *Reference* to a *HistoryServerCapabilitiesType Object*.

The content of this *BaseObjectType* is already defined by its type definition in Part 5. The *Object* extensions are formally defined in Table 8.

These properties are intended to inform a *Client* of the general capabilities of the *Server*. They do not guarantee that all capabilities will be available for all *Nodes*. For example not all *Nodes* will support *Events*, or in the case of an aggregating *Server* where underlying *Servers* may not support *Insert* or a particular *Aggregate*. In such cases the *HistoryServerCapabilities Property* would indicate the capability is supported, and the *Server* would return appropriate *StatusCodes* for situations where the capability does not apply.

Table 8 – HistoryServerCapabilitiesType Definition

Attribute	Value				
BrowseName	HistoryServerCapabilitiesType				
IsAbstract	False				
References	**NodeClass**	**Browse Name**	**Data Type**	**Type Definition**	**ModelingRule**
HasProperty	Variable	AccessHistoryDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	AccessHistoryEventsCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	MaxReturnDataValues	UInt32	PropertyType	Mandatory
HasProperty	Variable	MaxReturnEventValues	UInt32	PropertyType	Mandatory
HasProperty	Variable	InsertDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	ReplaceDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	UpdateDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteRawCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteAtTimeCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	InsertEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	ReplaceEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	UpdateEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	InsertAnnotationsCapability	Boolean	PropertyType	Mandatory
HasComponent	Object	AggregateFunctions	--	FolderType	Mandatory
HasComponent	Object	AggregateConfiguration	--	AggregateConfigurationType	Optional
HasComponent	Variable	ServerTimestampSupported	Boolean	PropertyType	Optional

All UA *Servers* that support Historical Access shall include the *HistoryServerCapabilities* as part of its *ServerCapabilities*.

The *AccessHistoryDataCapability Variable* defines if the *Server* supports access to historical data values. A value of True indicates the *Server* supports access to the history for *HistoricalNodes*, a value of False indicates the *Server* does not support access to the history for *HistoricalNodes*. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessHistoryEventsCapability* shall have a value of True for the *Server* to be a valid OPC UA *Server* supporting Historical Access.

The *AccessHistoryEventCapability Variable* defines if the server supports access to historical *Events*. A value of True indicates the server supports access to the history of *Events*, a value of False indicates the *Server* does not support access to the history of *Events*. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessHistoryEventsCapability* shall have a value of True for the *Server* to be a valid OPC UA *Server* supporting Historical Access.

The *MaxReturnDataValues Variable* defines the maximum number of values that can be returned by the *Server* for each *HistoricalNode* accessed during a request. A value of 0 indicates that the *Server* forces no limit on the number of values it can return. It is valid for a *Server* to limit the number of returned values and return a continuation point even if *MaxReturnValues* = 0. For example, it is possible that although the *Server* does not impose any restrictions, the underlying system may impose a limit that the *Server* is not aware of. The default value is 0.

Similarily, the *MaxReturnEventValues* specifies the maximum number of *Events* that a *Server* can return for a *HistoricalEventNode*.

The *InsertDataCapability Variable* indicates support for the Insert capability. A value of True indicates the *Server* supports the capability to insert new data values in history, but not overwrite existing values. The default value is False.

The *ReplaceDataCapability Variable* indicates support for the Replace capability. A value of True indicates the *Server* supports the capability to replace existing data values in history, but will not insert new values. The default value is False.

The *UpdateDataCapability Variable* indicates support for the Update capability. A value of True indicates the *Server* supports the capability to insert new data values into history if none exists, and replace values that currently exist. The default value is False.

The *DeleteRawCapability Variable* indicates support for the delete raw values capability. A value of True indicates the *Server* supports the capability to delete *raw data* values in history. The default value is False.

The *DeleteAtTimeCapability Variable* indicates support for the delete at time capability. A value of True indicates the *Server* supports the capability to delete a data value at a specified time. The default value is False.

The *InsertEventCapability Variable* indicates support for the Insert capability. A value of True indicates the *Server* supports the capability to insert new *Events* in history. An insert is not a replace. The default value is False.

The *ReplaceEventCapability Variable* indicates support for the Replace capability. A value of True indicates the *Server* supports the capability to replace existing *Events* in history. A replace is not an insert. The default value is False.

The *UpdateEventCapability Variable* indicates support for the Update capability. A value of True indicates the *Server* supports the capability to insert new *Events* into history if none exists, and replace values that currently exist. The default value is False.

The *DeleteEventCapability Variable* indicates support for the deletion of *Events* capability. A value of True indicates the *Server* supports the capability to delete *Events* in history. The default value is False.

The *InsertAnnotationCapability Variable* indicates support for *Annotations*. A value of True indicates the *Server* supports the capability to insert *Annotations*. Some *Servers* that support Inserting of *Annotations* will also support editing and deleting of *Annotations*. The default value is False.

AggregateFunctions is an entry point to browse to all *Aggregate* capabilities supported by the *Server* for Historical Access. All *HistoryAggregates* supported by the *Server* should be able to be browsed starting from this *Object*. *Aggregates* are defined in Part 13. If the *Server* does not support *Aggregates* the *Folder* is left empty.

AggregateConfiguration Object represents the browse entry point for information on how the *Server* treats *Aggregate* specific functionality such as handling *Uncertain data*. This *Object* is listed as optional for backward compatability, but it is required to be present if Aggregates are supported (via *Profiles*). *Aggregates* are defined in Part 13.

The *ServerTimestampSupported Variable* indicates support for the *ServerTimestamp* capability. A value of True indicates the *Server* supports *ServerTimestamps* in addition to *SourceTimestamp*. The default is False. This property is optional but it is expected all new *Servers* include this property.

5.5 Annotation DataType

This *DataType* describes *Annotation* information for the history data items. Its elements are defined in Table 9.

Table 9 – Annotation Structure

Name	Type	Description
Annotation	Structure	
message	String	*Annotation* message or text.
username	String	The user that added the *Annotation*, as supplied by the underlying system.
annotationTime	UtcTime	The time the *Annotation* was added. This will probably be different than the *SourceTimestamp*.

5.6 Historical Audit Events

5.6.1 General

AuditEvents are generated as a result of an action taken on the *Server* by a *Client* of the *Server*. For example, in response to a *Client* issuing a write to a *Variable*, the *Server* would generate an *AuditEvent* describing the *Variable* as the source and the user and *Client Session* as the initiators of the *Event*. Not all *Servers* support auditing, but if a *Server* supports auditing then it shall support audit *Events* as described in 5.6. *Profiles* (see Part 7) can be used to determine if a *Server* supports auditing. *Servers* shall generate *Events* of the *AuditHistoryUpdateEventType* or a sub-type of this type for all invocations of the *HistoryUpdate Service* on any *HistoricalNode*. See Part 3 and Part 5 for details on the *AuditHistoryUpdateEventType* model. In the case where the *HistoryUpdate Service* is invoked to insert Historical *Events*, the *AuditHistoryEventUpdateEventType Event* shall include the EventId of the inserted *Event* and a description that indicates that the *Event* was inserted. In the case where the *HistoryUpdate Service* is invoked to delete records, the *AuditHistoryDeleteEventType* or one of its sub-types shall be generated. See 6.7 for details on updating historical data or *Events*.

In particular using the Delete raw or Delete modified functionality shall generate an *AuditHistoryRawModifyDeleteEventType Event* or a sub-type of it. Using the Delete at time functionality shall generate an *AuditHistoryAtTimeDeleteEventType Event* or a sub-type of it. Using the Delete *Event* functionality shall generate an *AuditHistoryEventDeleteEventType Event* or a sub-type of it. All other updates shall follow the guidelines provided in the *AuditHistoryUpdateEventType* model.

5.6.2 AuditHistoryEventUpdateEventType

This is a subtype of AuditHistoryUpdateEventType and is used for categorization of History *Event* update related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 10.

Table 10 – AuditHistoryEventUpdateEventType definition

Attribute	Value				
BrowseName	AuditHistoryEventUpdateEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditHistoryUpdateEventType* defined in Part 3, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	PerformInsertReplace	PerformUpdateType	PropertyType	Mandatory
HasProperty	Variable	Filter	EventFilter	PropertyType	Mandatory
HasProperty	Variable	NewValues	HistoryEventFieldList [ ]	PropertyType	Mandatory
HasProperty	Variable	OldValues	HistoryEventFieldList [ ]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryUpdateEventType*. Their semantic is defined in Part 5.

The *UpdateNode* identifies the *Attribute* that was written on the *SourceNode*.

The *PerformInsertReplace* enumeration reflects the parameter on the *Service* call.

The *Filter* reflects the *Event* filter passed on the call to select the *Events* that are to be updated.

The *NewValues* identify the value that was written to the *Event*.

The *OldValues* identify the value that the *Events* contained before the update. It is acceptable for a *Server* that does not have this information to report a null value. In the case of an insert it is expected to be a null value.

Both the *NewValues* and the *OldValues* will contain *Events* with the appropriate fields, each with appropriately encoded values.

5.6.3 AuditHistoryValueUpdateEventType

This is a subtype of *AuditHistoryUpdateEventType* and is used for categorization of history value update related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 11.

Table 11 – AuditHistoryValueUpdateEventType definition

Attribute	**Value**				
BrowseName	AuditHistoryValueUpdateEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditHistoryUpdateEventType* defined in Part 3, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	PerformInsertReplace	PerformUpdateType	PropertyType	Mandatory
HasProperty	Variable	NewValues	DataValue[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValue[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryUpdateEventType*. Their semantic is defined in Part 5.

The *UpdatedNode* identifies the *Attribute* that was written on the *SourceNode*.

The *PerformInsertReplace* enumeration reflects the parameter on the *Service* call.

The *NewValues* identify the value that was written to the *Event*.

The *OldValues* identify the value that the *Event* contained before the write. It is acceptable for a *Server* that does not have this information to report a null value. In the case of an insert it is expected to be a null value.

Both the *NewValues* and the *OldValues* will contain a value in the *DataType* and encoding used for writing the value.

5.6.4 AuditHistoryDeleteEventType

This is a subtype of *AuditHistoryUpdateEventType* and is used for categorization of history delete related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 12.

Table 12 – AuditHistoryDeleteEventType definition

Attribute	**Value**
BrowseName	AuditHistoryDeleteEventType
IsAbstract	False

References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the *AuditHistoryUpdateEventType* defined in Part 3, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasSubtype	ObjectType	AuditHistoryRawModifyDeleteEventType			
HasSubtype	ObjectType	AuditHistoryAtTimeDeleteEventType			
HasSubtype	ObjectType	AuditHistoryEventDeleteEventType			

This *EventType* inherits all *Properties* of the *AuditUpdateEventType*. Their semantic is defined in Part 5.

The *UpdatedNode property* identifies the *NodeId* that was used for the delete operation.

5.6.5 AuditHistoryRawModifyDeleteEventType

This is a subtype of *AuditHistoryDeleteEventType* and is used for categorization of history delete related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 13.

Table 13 – AuditHistoryRawModifyDeleteEventType definition

Attribute	Value				
BrowseName	AuditHistoryRawModifyDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditHistoryDeleteEventType* defined in Table 12, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	IsDeleteModified	Boolean	PropertyType	Mandatory
HasProperty	Variable	StartTime	UtcTime	PropertyType	Mandatory
HasProperty	Variable	EndTime	UtcTime	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValue[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in 5.6.4.

The *isDeleteModified* reflects the isDeleteModified parameter of the call.

The *StartTime* reflects the starting time parameter of the call.

The *EndTime* reflects the ending time parameter of the call.

The *OldValues* identify the value that history contained before the delete. A *Server* should report all deleted values. It is acceptable for a *Server* that does not have this information to report a null value. The *OldValues* will contain a value in the *DataType* and encoding used for writing the value.

5.6.6 AuditHistoryAtTimeDeleteEventType

This is a subtype of *AuditHistoryDeleteEventType* and is used for categorization of history delete related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 14.

Table 14 – AuditHistoryAtTimeDeleteEventType definition

Attribute	Value
BrowseName	AuditHistoryAtTimeDeleteEventType
IsAbstract	False

References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the *AuditHistoryDeleteEventType* defined in Table 12, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	ReqTimes	UtcTime[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValues[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in 5.6.7.

The *ReqTimes* reflect the request time parameter of the call.

The *OldValues* identifies the value that history contained before the delete. A *Server* should report all deleted values. It is acceptable for a *Server* that does not have this information to report a null value. The *OldValues* will contain a value in the *DataType* and encoding used for writing the value.

5.6.7 AuditHistoryEventDeleteEventType

This is a subtype of *AuditHistoryDeleteEventType* and is used for categorization of history delete related *Events*. This type follows all the behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 15.

Table 15 – AuditHistoryEventDeleteEventType definition

Attribute	Value				
BrowseName	AuditHistoryEventDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditHistoryDeleteEventType* defined in Table 12, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	EventIds	ByteString[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	HistoryEventFieldList	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in 5.6.4.

The *EventIds* reflect the EventIds parameter of the call.

The *OldValues* identify the value that history contained before the delete. A *Server* should report all deleted values. It is acceptable for a *Server* that does not have this information to report a null value. The *OldValues* will contain an *Event* with the appropriate fields, each with appropriately encoded values.

6 Historical Access specific usage of Services

6.1 General

Part 4 specifies all *Services* needed for OPC UA Historical Access. In particular:

- The Browse Service Set or Query Service Set to detect *HistoricalNodes* and their configuration.
- The *HistoryRead* and *HistoryUpdate* Services of the Attribute Service Set to read and update history of *HistoricalNodes.*

6.2 Historical Nodes StatusCodes

6.2.1 Overview

6.2 defines additional codes and rules that apply to the *StatusCode* when used for *HistoricalNodes*.

The general structure of the *StatusCode* is specified in Part 4. It includes a set of common operational result codes which also apply to historical data and/or *Events*.

6.2.2 Operation level result codes

In OPC UA Historical Access the *StatusCode* is used to indicate the conditions under which a *Value* or *Event* was stored, and thereby can be used as an indicator of its usability. Due to the nature of historical data and/or *Events*, additional information beyond the basic quality and call result code needs to be conveyed to the *Client*, for example, whether the value is actually stored in the data repository, whether the result was *Interpolated*, whether all data inputs to a calculation were of good quality, etc.

In the following, Table 16 contains codes with *Bad* severity indicating a failure; Table 17 contains *Good* (success) codes.

It is important to note that these are the codes that are specific for OPC UA Historical Access and supplement the codes that apply to all types of data and are therefore defined in Part 4 , Part 8 and Part 13.

Table 16 – Bad operation level result codes

Symbolic Id	Description
Bad_NoData	No data exists for the requested time range or *Event* filter.
Bad_BoundNotFound	No data found to provide upper or lower bound value.
Bad_BoundNotSupported	Bounding Values are not applicable or the *Server* has reached its search limit and will not return a bound.
Bad_DataLost	Data is missing due to collection started/stopped/lost.
Bad_DataUnavailable	Expected data is unavailable for the requested time range due to an un-mounted volume, an off-line historical collection, or similar reason for temporary unavailability.
Bad_EntryExists	The data or *Event* was not successfully inserted because a matching entry exists.
Bad_NoEntryExists	The data or *Event* was not successfully updated because no matching entry exists.
Bad_TimestampNotSupported	The *Client* requested history using a TimestampsToReturn the *Server* does not support (i.e. requested *Server* Timestamp when *Server* only supports SourceTimestamp).
Bad_InvalidArgument	One or more arguments are invalid or missing.
Bad_AggregateListMismatch	The list of Aggregates does not have the same length as the list of operations.
Bad_AggregateConfigurationRejected	The *Server* does not support the specified AggregateConfiguration for the Node.
Bad_AggregateNotSupported	The specified Aggregate is not valid for the specified *Node*.
Bad_ArgumentsMissing	See Part 4 for the description of this result code.
Bad_TypeDefinitionInvalid	See Part 4 for the description of this result code.
Bad_SourceNodeIdInvalid	See Part 4 for the description of this result code.
Bad_OutOfRange	See Part 4 for the description of this result code.
Bad_NotSupported	See Part 4 for the description of this result code.
Bad_IndexRangeInvalid	See Part 4 for the description of this result code.
Bad_NotWriteable	See Part 4 for the description of this result code.

Table 17 – Good operation level result codes

Symbolic Id	Description
Good_NoData	No data exists for the requested time range or *Event* filter.
Good_EntryInserted	The data or *Event* was successfully inserted into the historical database
Good_EntryReplaced	The data or *Event* field was successfully replaced in the historical database
Good_DataIgnored	The *Event* field was ignored and was not inserted into the historical database.

It may be noted that there are both Good and Bad Status codes that deal with cases of no data or missing data. In general *Good_NoData* is used for cases where no data was found when performing a simple 'Read' request. *Bad_NoData i*s used in cases where some action is requested on an interval and no data could be found. The distinction exists if users are attempting an action on a given interval where they would expect data to exist, or would like to be notified that the requested action could not be performed.

Good_NoData is returned for cases such as:

- ReadEvents where *startTime=endTime*
- ReadEvent data is requested and does not exist
- ReadRaw where data is requested and does not exist

Bad_NoData is returned for cases such as:

- ReadEvent data is requested and underlying historian does not support the requested field
- ReadProcessed where data is requested and does not exist
- Any Delete requests where data does not exist

The above use cases are illustrative examples. Detailed explanations on when each status code is returned are found in 6.4 and 6.7

6.2.3 Semantics changed

The *StatusCode* in addition contains an informational bit called *Semantics Changed* (see Part 4).

UA *Servers* that implement OPC UA Historical Access should not set this bit; rather they should propagate the *StatusCode* which has been stored in the data repository. The *Client* should be aware that the returned data values may have this bit set.

6.3 Continuation Points

The *continuationPoint* parameter in the *HistoryRead Service* is used to mark a point from which to continue the read if not all values could be returned in one response. The value is opaque for the *Client* and is only used to maintain the state information for the *Server* to continue from. For *HistoricalDataNode* requests, a *Server* may use the timestamp of the last returned data item if the timestamp is unique. This can reduce the need in the *Server* to store state information for the continuation point.

The *Client* specifies the maximum number of results per operation in the request *Message*. A *Server* shall not return more than this number of results but it may return fewer results. The *Server* allocates a *ContinuationPoint* if there are more results to return. The *Server* may return fewer results due to buffer issues or other internal constraints. It may also be required to return a *continuationPoint* due to *HistoryRead* parameter constraints. If a request is taking a long time to calculate and is approaching the timeout time, the *Server* may return partial results with a continuation point. This may be done if the calculation is going to take more time than the *Client* timeout. In some cases it may take longer than the *Client* timeout to calculate even one result. Then the *Server* may return zero results with a continuation point that allows the *Server* to resume the calculation on the next *Client* read call. For additional discussions regarding *ContinuationPoints* and *HistoryRead* please see the individual extensible *HistoryReadDetails* parameter in 6.4

If the *Client* specifies a *ContinuationPoint*, then the *HistoryReadDetails* parameter and the TimestampsToReturn parameter are ignored, because it does not make sense to request different parameters when continuing from a previous call. It is permissible to change the *dataEncoding* parameter with each request.

If the *Client* specifies a *ContinuationPoint* that is no longer valid, then the *Server* shall return a *Bad_ContinuationPointInvalid* error.

If the *releaseContinuationPoints* parameter is set in the request the *Server* shall not return any data and shall release all *ContinuationPoints* passed in the request. If the *ContinuationPoint* for an operation is missing or invalid then the *StatusCode* for the operation shall be *Bad_ContinuationPointInvalid.*

6.4 HistoryReadDetails parameters

6.4.1 Overview

The *HistoryRead Service* defined in Part 4 can perform several different functions. The *HistoryReadDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See Part 4 for the definition of *Extensible Parameter*. Table 18 lists the symbolic names of the valid *Extensible Parameter* structures. Some structures will perform different functions based on the setting of its associated parameters. For simplicity a functionality of each structure is listed. For example, text such as 'using the Read modified functionality' refers to the function the *HistoryRead Service* performs using the *Extensible Parameter* structure ReadRawModifiedDetails with the isReadModified Boolean parameter set to TRUE.

Table 18 – HistoryReadDetails parameterTypeIds

Symbolic Name	Functionality	Description
ReadEventDetails	Read event	This structure selects a set of *Events* from the history database by specifying a filter and a time domain for one or more *Objects* or *Views*. See 6.4.2.1. When this parameter is specified the *Server* returns a *HistoryEvent* structure for each operation (see 6.5.4).
ReadRawModifiedDetails	Read raw	This structure selects a set of values from the history database by specifying a time domain for one or more *Variables*. See 6.4.3.1. When this parameter is specified the *Server* returns a *HistoryData* structure for each operation (see 6.5.2).
ReadRawModifiedDetails	Read modified	This parameter selects a set of *modified values* from the history database by specifying a time domain for one or more *Variables.* See 6.4.3.1. When this parameter is specified the *Server* returns a *HistoryModifiedData* structure for each operation (see 6.5.3).
ReadProcessedDetails	Read processed	This structure selects a set of *Aggregate* values from the history database by specifying a time domain for one or more *Variables*. See 6.4.4.1. When this parameter is specified the *Server* returns a *HistoryData* structure for each operation (see 6.5.2)
ReadAtTimeDetails	Read at time	This structure selects a set of raw or interpolated values from the history database by specifying a series of timestamps for one or more *Variables*. See 6.4.5.1. When this parameter is specified the *Server* returns a *HistoryData* structure for each operation (see 6.5.2).

6.4.2 ReadEventDetails structure

6.4.2.1 ReadEventDetails structure details

Table 19 defines the *ReadEventDetails* structure. This parameter is only valid for *Objects* that have the *EventNotifier Attribute* set to TRUE (see Part 3). Two of the three parameters, *numValuesPerNode*, *startTime*, and *endTime* shall be specified.

Table 19 – ReadEventDetails

Name	Type	Description
ReadEventDetails	Structure	Specifies the details used to perform an *Event* history read.
numValuesPerNode	Counter	The maximum number of values returned for any *Node* over the time range. If only one time is specified, the time range shall extend to return this number of values. The default value of 0 indicates that there is no maximum.
startTime	UtcTime	Beginning of period to read. The default value of *DateTime.MinValue*

		indicates that the *startTime* is Unspecified.
endTime	UtcTime	End of period to read. The default value of *DateTime.MinValue* indicates that the endTime is Unspecified.
Filter	EventFilter	A filter used by the *Server* to determine which HistoricalEventNode should be included. This parameter shall be specified and at least one *EventField* is required. The *EventFilter* parameter type is an *Extensible parameter* type. It is defined and used in the same manner as defined for monitored data items which are specified in Part 4. This filter also specifies the *EventFields* that are to be returned as part of the request.

6.4.2.2 Read Event functionality

The *ReadEventDetails* structure is used to read the *Events* from the history database for the specified time domain for one or more *HistoricalEventNodes*. The *Events* are filtered based on the filter structure provided. This filter includes the *EventFields* that are to be returned. For a complete description of filter refer to Part 4.

The *startTime* and *endTime* are used to filter on the Time field for *Events*.

The time domain of the request is defined by *startTime*, *endTime*, and numValuesPerNode; at least two of these shall be specified. If *endTime* is less than *startTime*, or *endTime* and *numValuesPerNode* alone are specified then the data will be returned in reverse order with later/newer data provided first as if time were flowing backward. If all three are specified then the call shall return up to numValuesPerNode results going from *startTime* to *endTime*, in either ascending or descending order depending on the relative values of *startTime* and *endTime*. If numValuesPerNode is 0 then all of the values in the range are returned. The default value is used to indicate when *startTime*, *endTime* or *numValuesPerNode* are not specified.

It is specifically allowed for the *startTime* and the *endTime* to be identical. This allows the *Client* to request the *Event* at a single instance in time. When the *startTime* and *endTime* are identical then time is presumed to be flowing forward. If no data exists at the time specified then the *Server* shall return the *Good_NoData StatusCode*.

If a *startTime*, *endTime* and *numValuesPerNode* are all provided, and if more than *numValuesPerNode Events* exist within that time range for a given *Node*, then only *numValuesPerNode Events* per *Node* are returned along with a *ContinuationPoint*. When a *ContinuationPoint* is returned, a *Client* wanting the next *numValuesPerNode* values should call *HistoryRead* again with the *continuationPoint* set.

If the request takes a long time to process then the *Server* can return partial results with a *ContinuationPoint*. This might be done if the request is going to take more time than the *Client* timeout hint. It may take longer than the *Client* timeout hint to retrieve any results. In this case the *Server* may return zero results with a *ContinuationPoint* that allows the *Server* to resume the calculation on the next *Client HistoryRead* call.

For an interval in which no data exists, the corresponding *StatusCode* shall be *Good_NoData*.

The filter parameter is used to determine which historical *Events* and their corresponding fields are returned. It is possible that the fields of an *EventType* are available for real time updating, but not available from the historian. In this case a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* shall be *Bad_NoData*.

If the requested *TimestampsToReturn* is not supported for a *Node* then the operation shall return the *Bad_TimestampNotSupported StatusCode*. When reading *Events* this only applies to *Event* fields that are of type *DataValue*.

6.4.3 ReadRawModifiedDetails structure

6.4.3.1 ReadRawModifiedDetails structure details

Table 20 defines the *ReadRawModifiedDetails* structure. Two of the three parameters, numValuesPerNode, *startTime*, and *endTime* shall be specified.

Table 20 – ReadRawModifiedDetails

Name	Type	Description
ReadRawModifiedDetails	Structure	Specifies the details used to perform a “raw” or “modified” history read.
isReadModified	Boolean	TRUE for Read Modified functionality, FALSE for Read Raw functionality. Default value is FALSE.
startTime	UtcTime	Beginning of period to read. Set to default value of *DateTime.MinValue* if no specific start time is specified.
endTime	UtcTime	End of period to read. Set to default value of *DateTime.MinValue* if no specific end time is specified.
numValuesPerNode	Counter	The maximum number of values returned for any *Node* over the time range. If only one time is specified, the time range shall extend to return this number of values. The default value 0 indicates that there is no maximum.
returnBounds	Boolean	A Boolean parameter with the following values: TRUE Bounding Values should be returned FALSE All other cases

6.4.3.2 Read raw functionality

When this structure is used for reading *Raw* Values (isReadModified is set to FALSE), it reads the values, qualities, and timestamps from the history database for the specified time domain for one or more *HistoricalDataNodes*. This parameter is intended for use by a *Client* that wants the actual data saved within the historian. The actual data may be compressed or may be all raw data collected for the item depending on the historian and the storage rules invoked when the item values were saved. When returnBounds is TRUE, the Bounding Values for the time domain are returned. The optional Bounding Values are provided to allow the *Client* to interpolate values for the start and end times when trending the actual data on a display.

The time domain of the request is defined by *startTime*, *endTime*, and *numValuesPerNode*; at least two of these shall be specified. If endTime is less than *startTime*, or *endTime* and *numValuesPerNode* alone are specified then the data will be returned in reverse order, with later data coming first as if time were flowing backward. If all three are specified then the call shall return up to *numValuesPerNode* results going from *startTime* to *endTime*, in either ascending or descending order depending on the relative values of *startTime* and *endTime*. If *numValuesPerNode* is 0, then all the values in the range are returned. A default value of *DateTime.MinValue* (see Part 6) is used to indicate when *startTime* or *endTime* is not specified.

It is specifically allowed for the *startTime* and the *endTime* to be identical. This allows the *Client* to request just one value. When the *startTime* and *endTime* are identical then time is presumed to be flowing forward. It is specifically not allowed for the *Server* to return a *Bad_InvalidArgument StatusCode* if the requested time domain is outside of the *Server*'s range. Such a case shall be treated as an interval in which no data exists.

If a *startTime*, *endTime* and *numValuesPerNode* are all provided and if more than *numValuesPerNode* values exist within that time range for a given *Node* then only *numValuesPerNode* values per *Node* are returned along with a *continuationPoint*. When *a continuationPoint* is returned, a *Client* wanting the next numValuesPerNode values should call ReadRaw again with the *continuationPoint* set.

If the request takes a long time to process then the *Server* can return partial results with a *ContinuationPoint*. This might be done if the request is going to take more time than the *Client* timeout hint. It may take longer than the *Client* timeout hint to retrieve any results. In this case the *Server* may return zero results with a *ContinuationPoint* that allows the *Server* to resume the calculation on the next *Client HistoryRead* call.

If Bounding Values are requested and a non-zero *numValuesPerNode* was specified then any Bounding Values returned are included in the *numValuesPerNode* count. If *numValuesPerNode* is 1 then only the start bound is returned (the end bound if the reverse order is needed). If *numValuesPerNode* is 2 then the start bound and the first data point are

returned (the end bound if reverse order is needed). When Bounding Values are requested and no bounding value is found then the corresponding *StatusCode* entry will be set to *Bad_BoundNotFound*, a timestamp equal to the start or end time as appropriate, and a value of null. How far back or forward to look in history for Bounding Values is *Server* dependent.

For an interval in which no data exists, if Bounding Values are not requested, then the corresponding *StatusCode* shall be *Good_NoData*. If Bounding Values are requested and one or both exist, then the result code returned is Success and the bounding value(s) are returned.

For cases where there are multiple values for a given timestamp, all but the most recent are considered to be *Modified values* and the *Server* shall return the most recent value. If the *Server* returns a value which hides other values at a timestamp then it shall set the *ExtraData* bit in the *StatusCode* associated with that value. If the *Server* contains additional information regarding a value then the ExtraData bit shall also be set. It indicates that ModifiedValues are available for retrieval, see 6.4.3.3.

If the requested TimestampsToReturn is not supported for a *Node*, the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.4.3.3 Read modified functionality

When this structure is used for reading *Modified* Values (isReadModified is set to TRUE), it reads the modified values, *StatusCodes*, timestamps, modification type, the user identifier, and the timestamp of the modification from the history database for the specified time domain for one or more *HistoricalDataNodes*. If there are multiple replaced values the *Server* shall return all of them. The updateType specifies what value is returned in the modification record. If the updateType is INSERT the value is the new value that was inserted. If the updateType is anything else the value is the old value that was changed. See 6.8 *HistoryUpdateDetails* parameter for details on what updateTypes are available.

The purpose of this function is to read values from history that have been *Modified.* The returnBounds parameter shall be set to FALSE for this case, otherwise the *Server* returns a *Bad_InvalidArgument StatusCode*.

The domain of the request is defined by *startTime*, *endTime*, and *numValuesPerNode*; at least two of these shall be specified. If endTime is less than *startTime*, or *endTime* and *numValuesPerNode* alone are specified, then the data shall be returned in reverse order with the later data coming first. If all three are specified then the call shall return up to numValuesPerNode results going from StartTime to *EndTime*, in either ascending or descending order depending on the relative values of *StartTime* and *EndTime*. If more than *numValuesPerNode* values exist within that time range for a given *Node* then only *numValuesPerNode* values per *Node* are returned along with a *continuationPoint*. When *a continuationPoint* is returned, a *Client* wanting the next *numValuesPerNode* values should call *ReadRaw* again with the *continuationPoint* set. If *numValuesPerNode* is 0 then all of the values in the range are returned. If the *Server* cannot return all *modified values* for a given timestamp in a single response then it shall return modified values with the same timestamp in subsequent calls.

If the request takes a long time to process then the *Server* can return partial results with a *ContinuationPoint*. This might be done if the request is going to take more time than the *Client* timeout hint. It may take longer than the *Client* timeout hint to retrieve any results. In this case the *Server* may return zero results with a *ContinuationPoint* that allows the *Server* to resume the calculation on the next *Client HistoryRead* call.

If a value has been modified multiple times then all values for the time are returned. This means that a timestamp can appear in the array more than once. The order of the returned values with the same timestamp should be from the most recent to oldest modification timestamp, if *startTime* is less than or equal to *endTime*. If *endTime* is less than *startTime*, then the order of the returned values will be from the oldest modification timestamp to the most recent. It is *Server* dependent whether multiple modifications are kept or only the most recent.

A *Server* does not have to create a modification record for data when it is first added to the historical collection. If it does then it shall set the ExtraData bit and the *Client* can read the modification record using a ReadModified call. If the data is subsequently modified the *Server* shall create a second modification record which is returned along with the original modification record whenever a *Client* uses the ReadModified call if the *Server* supports multiple modification records per timestamp.

If the requested TimestampsToReturn is not supported for a *Node* then the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.4.4 ReadProcessedDetails structure

6.4.4.1 ReadProcessedDetails structure details

Table 21 defines the structure of the ReadProcessedDetails structure.

Table 21 – ReadProcessedDetails

Name	Type	Description
ReadProcessedDetails	Structure	Specifies the details used to perform a “processed” history read.
startTime	UtcTime	Beginning of period to read.
endTime	UtcTime	End of period to read.
ProcessingInterval	Duration	Interval between returned *Aggregate* values. The value 0 indicates that there is no *ProcessingInterval* defined.
aggregateType[]	NodeId	The *NodeId* of the HistoryAggregate object that indicates the list of *Aggregates* to be used when retrieving the processed history. See Part 13 for details.
aggregateConfiguration	Aggregate Configuration	*Aggregate* configuration structure.
useSeverCapabilitiesDefaults	Boolean	As described in Part 4.
TreatUncertainAsBad	Boolean	As described in Part 13.
PercentDataBad	UInt8	As described in Part 13.
PercentDataGood	UInt8	As described in Part 13.
UseSlopedExtrapolation	Boolean	As described in Part 13.

See Part 13 for details on possible *NodeId* values for the *aggregateType* parameter.

6.4.4.2 Read processed functionality

This structure is used to compute *Aggregate* values, qualities, and timestamps from data in the history database for the specified time domain for one or more *HistoricalDataNodes*. The time domain is divided into intervals of duration *ProcessingInterval*. The specified *Aggregate* Type is calculated for each interval beginning with *startTime* by using the data within the next *ProcessingInterval*.

For example, this function can provide hourly statistics such as Maximum, Minimum, and Average for each item during the specified time domain when *ProcessingInterva*l is 1 hour.

The domain of the request is defined by *startTime*, *endTime*, and *ProcessingInterval*. All three shall be specified. If *endTime* is less than *startTime* then the data shall be returned in reverse order with the later data coming first. If *startTime* and *endTime* are the same then the *Server* shall return *Bad_InvalidArgument* as there is no meaningful way to interpret such a case. If the *ProcessingInteval* is specified as 0 then *Aggregates* shall be calculated using one interval starting at *startTime* and ending at *endTime*.

The aggregateType[] parameter allows a *Client* to request multiple *Aggregate* calculations per requested NodeId. If multiple *Aggregates* are requested then a corresponding number of entries are required in the NodesToRead array.

For example, to request Min *Aggregate* for *NodeId* FIC101, FIC102, and both Min and Max *Aggregates* for *NodeId* FIC103 would require *NodeId* FIC103 to appear twice in the NodesToRead array request parameter.

aggregateType[]	NodesToRead[]
Min	FIC101
Min	FIC102
Min	FIC103
Max	FIC103

If the array of *Aggregates* does not match the array of *NodesToRead* then the *Server* shall return a *StatusCode* of *Bad_AggregateListMismatch.*

The aggregateConfiguration parameter allows a *Client* to override the *Aggregate* configuration settings supplied by the AggregateConfiguration *Object* on a per call basis. See Part 13 for more information on *Aggregate* configurations. If the *Server* does not support the ability to override the *Aggregate* configuration settings then it shall return a *StatusCode* of *Bad_ AggregateConfigurationRejected.* If the *Aggregate* is not valid for the *Node* then the *StatusCode* shall be *Bad_AggregateNotSupported.*

The values used in computing the *Aggregate* for each interval shall include any value that falls exactly on the timestamp at the beginning of the interval, but shall not include any value that falls directly on the timestamp ending the interval. Thus, each value shall be included only once in the calculation. If the time domain is in reverse order then we consider the later timestamp to be the one beginning the subinterval, and the earlier timestamp to be the one ending it. Note that this means that simply swapping the start and end times will not result in getting the same values back in reverse order as the intervals being requested in the two cases are not the same.

If an *Aggregate* is taking a long time to calculate then the *Server* can return partial results with a continuation point. This might be done if the calculation is going to take more time than the *Client* timeout hint. In some cases it may take longer than the *Client* timeout hint to calculate even one *Aggregate* result. Then the *Server* may return zero results with a continuation point that allows the *Server* to resume the calculation on the next *Client* read call.

Refer to Part 13 for handling of *Aggregate* specific cases.

6.4.5 ReadAtTimeDetails structure

6.4.5.1 ReadAtTimeDetails structure details

Table 22 defines the ReadAtTimeDetails structure.

Table 22 – ReadAtTimeDetails

Name	Type	Description
ReadAtTimeDetails	Structure	Specifies the details used to perform an “at time” history read.
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be read.
useSimpleBounds	Boolean	Use SimpleBounds to determine the value at the specific timestamp.

6.4.5.2 Read at time functionality

The ReadAtTimeDetails structure reads the values and qualities from the history database for the specified timestamps for one or more *HistoricalDataNodes*. This function is intended to provide values to correlate with other values with a known timestamp. For example, a *Client* may need to read the values of sensors when lab samples were collected.

The order of the values and qualities returned shall match the order of the timestamps supplied in the request.

When no value exists for a specified timestamp, a value shall be *Interpolated* from the surrounding values to represent the value at the specified timestamp. The interpolation will follow the same rules as the standard *Interpolated Aggregate* as outlined in Part 13.

If the useSimpleBounds flag is True and Interpolation is required then *simple bounding values* will be used to calculate the data value. If useSimpleBounds is False and Interpolation is required then *interpolated bounding values* will be used to calculate the data value. See Part 13 for the definition of *simple bounding values* and *interpolated bounding values*.

If a value is found for the specified timestamp, then the *Server* will set the *StatusCode InfoBits* to be *Raw*. If the value is *Interpolated* from the surrounding values, then the *Server* will set the *StatusCode InfoBits* to be *Interpolated.*

If the read request is taking a long time to calculate then the *Server* may return zero results with a *ContinuationPoint* that allows the *Server* to resume the calculation on the next *Client HistoryRead* call.

If the requested TimestampsToReturn is not supported for a *Node*, then the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.5 HistoryData parameters returned

6.5.1 Overview

The *HistoryRead Service* returns different types of data depending on whether the request asked for the value *Attribute* of a *Node* or the history *Events* of a *Node*. The historyData is an *Extensible Parameter* whose structure depends on the functions to perform for the *HistoryReadDetails* parameter. See Part 4 for details on *Extensible Parameters*.

6.5.2 HistoryData type

Table 23 defines the structure of the *HistoryData* used for the data to return in a *HistoryRead*.

Table 23 – HistoryData Details

Name	Type	Description
dataValues[]	DataValue	An array of values of history data for the *Node*. The size of the array depends on the requested data parameters.

6.5.3 HistoryModifiedData type

Table 24 defines the structure of the *HistoryModifiedData* used for the data to return in a *HistoryRead* when IsReadModified = True.

Table 24 – HistoryModifiedData Details

Name	Type	Description
dataValues[]	DataValue	An array of values of history data for the *Node*. The size of the array depends on the requested data parameters.
modificationInfos[]	ModificationInfo	
Username	String	The name of the user that made the modification. Support for this field is optional. A null shall be returned if it is not defined.
modificationTime	UtcTime	The time the modification was made. Support for this field is optional. A null shall be returned if it is not defined.
updateType	HistoryUpdateType	The modification type for the item.

6.5.4 HistoryEvent type

Table 25 defines the HistoryEvent parameter used for Historical *Event* reads.

The HistoryEvent defines a table structure that is used to return *Event* fields to a *Historical Read*. The structure is in the form of a table consisting of one or more *Events*, each containing an array of one or more fields. The selection and order of the fields returned for each *Event* are identical to the selected parameter of the *EventFilter*.

Table 25 – HistoryEvent Details

Name	Type	Description
Events []	HistoryEventFieldList	The list of *Events* being delivered.
eventFields []	BaseDataType	List of selected *Event* fields. This will be a one-to-one match with the fields selected in the *EventFilter*.

6.6 HistoryUpdateType Enumeration

Table 26 defines the HistoryUpdate enumeration.

Table 26 – HistoryUpdateType Enumeration

Name	Description
INSERT_1	Data was inserted.
REPLACE_2	Data was replaced.
UPDATE_3	Data was inserted or replaced.
DELETE_4	Data was deleted.

6.7 PerformUpdateType Enumeration

Table 27 defines the PerformUpdateType enumeration.

Table 27 – PerformUpdateType Enumeration

Name	Description
INSERT_1	Data was inserted.
REPLACE_2	Data was replaced.
UPDATE_3	Data was inserted or replaced.
DELETE_4	Data was deleted.

6.8 HistoryUpdateDetails parameter

6.8.1 Overview

The *HistoryUpdate Service* defined in Part 4 can perform several different functions. The *historyUpdateDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See Part 4 for the definition of *Extensible Parameter*. Table 28 lists the symbolic names of the valid *Extensible Parameter* structures. Some structures will perform different functions based on the setting of its associated parameters. For simplicity a functionality of each structure is listed. For example text such as ‘using the Replace data functionality’ refers to the function the *HistoryUpdate Service* performs using the *Extensible Parameter* structure *UpdateDataDetails* with the performInsertReplace enumeration parameter set to REPLACE_2.

Table 28 – HistoryUpdateDetails parameter TypeIds

Symbolic Name	Functionality	Description
UpdateDataDetails	Insert data	This function inserts new values into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateDataDetails	Replace data	This function replaces existing values into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateDataDetails	Update data	This function inserts or replaces values into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Insert data	This function inserts new *Structured History Data* or *Annotations* into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Replace data	This function replaces existing *Structured History Data* or *Annotations* into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Update data	This function inserts or replaces *Structured History Data* or *Annotations* into the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Remove data	This function removes *Structured History Data* or *Annotations* from the history database at the specified timestamps for one or more *HistoricalDataNodes*. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateEventDetails	Insert events	This function inserts new *Events* into the history database for one or more *HistoricalEventNodes*.
UpdateEventDetails	Replace events	This function replaces values of fields in existing *Events* into the history database for one or more *HistoricalEventNodes*.
UpdateEventDetails	Update events	This function inserts new *Events* or replaces existing *Events* in the history database for one or more *HistoricalEventNodes*.
DeleteRawModifiedDetails	Delete raw	This function deletes all values from the history database for the specified time domain for one or more *HistoricalDataNodes*.
DeleteRawModifiedDetails	Delete modified	Some historians may store multiple values at the same Timestamp. This function will delete specified values and qualities for the specified timestamp for one or more *HistoricalDataNodes*.
DeleteAtTimeDetails	Delete at time	This function deletes all values in the history database for the specified timestamps for one or more *HistoricalDataNodes*.
DeleteEventDetails	Delete event	This function deletes *Events* from the history database for the specified filter for one or more *HistoricalEventNodes*.

The *HistoryUpdate Service* is used to update or delete both *DataValues* and *Events*. For simplicity the term "entry" will be used to mean either *DataValue* or *Event* depending on the context in which it is used. Auditing requirements for History *Services* are described in Part 4. This description assumes the user issuing the request and the *Server* that is processing the request support the capability to update entries. See Part 3 for a description of *Attributes* that expose the support of Historical Updates.

If the *HistoryUpdate Service* is called with both *DataValues* and *Events* in the same call the *Server* operational limits *MaxNodesPerHistoryUpdateData* and

MaxNodesPerHistoryUpdateEvents (See Part 5) may be ignored. The *Server* may return the service result code *Bad_TooManyOperations* if it is not able to handle the combination of *DataValues* and *Events*. It is recommended to call the *HistoryUpdate Service* twice, once with *DataValues* and then with *Events*.

6.8.2 UpdateDataDetails structure

6.8.2.1 UpdateDataDetails structure details

Table 29 defines the UpdateDataDetails structure.

Table 29 – UpdateDataDetails

Name	Type	Description
UpdateDataDetails	Structure	The details for insert, replace, and insert/replace history updates.
nodeId	NodeId	Node id of the *Object* to be updated.
performInsertReplace	PerformUpdateType	Value determines which action of insert, replace, or update is performed.
updateValues[]	DataValue	New values to be inserted or to replace.

Value	Description
INSERT_1	See 6.8.2.2.
REPLACE_2	See 6.8.2.3.
UPDATE_3	See 6.8.2.4.

6.8.2.2 Insert data functionality

Setting performInsertReplace = INSERT_1 inserts entries into the history database at the specified timestamps for one or more *HistoricalDataNodes*. If an entry exists at the specified timestamp, then the new entry shall not be inserted; instead the *StatusCode* shall indicate *Bad_EntryExists*.

This function is intended to insert new entries at the specified timestamps, e.g., the insertion of lab data to reflect the time of data collection.

If the *Time* does not fall within range that can be stored then the related *operationResults entry* shall indicate *Bad_OutOfRange.*

6.8.2.3 Replace data functionality

Setting performInsertReplace = REPLACE_2 replaces entries in the history database at the specified timestamps for one or more *HistoricalDataNodes*. If no entry exists at the specified timestamp, then the new entry shall not be inserted; otherwise the *StatusCode* shall indicate *Bad_NoEntryExists*.

This function is intended to replace existing entries at the specified timestamp, e.g., correct lab data that was improperly processed, but inserted into the history database.

6.8.2.4 Update data functionality

Setting performInsertReplace = UPDATE_3 inserts or replaces entries in the history database for the specified timestamps for one or more *HistoricalDataNodes*. If the item has an entry at the specified timestamp, then the new entry will replace the old one. If there is no entry at that timestamp, then the function will insert the new data.

A *Server* can create a *modified value* for a value being replaced or inserted (see 3.1.6) however it is not required.

This function is intended to unconditionally insert/replace values and qualities, e.g., correction of values for bad sensors.

Good as a *StatusCode* for an individual entry is allowed when the *Server* is unable to say whether there was already a value at that timestamp. If the *Server* can determine whether the

new entry replaces an entry that was already there, then it should use *Good_EntryInserted* or *Good_EntryReplaced* to return that information.

If the *Time* does not fall within range that can be stored then the related *operationResults entry* shall indicate *Bad_OutOfRange.*

6.8.3 UpdateStructureDataDetails structure

6.8.3.1 UpdateStructureDataDetails structure details

Table 29 defines the UpdateStructureDataDetails structure.

Table 30 – UpdateStructureDataDetails

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>UpdateStructureDataDetails</td><td>Structure</td><td>The details for data history updates.</td></tr>
<tr><td>nodeId</td><td>NodeId</td><td>Node id of the Object to be updated.</td></tr>
<tr><td>performInsertReplace</td><td>PerformUpdateType</td><td>Value determines which action of insert, replace, or update is performed.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>INSERT_1</td><td>See 6.8.3.3.</td></tr>
<tr><td>REPLACE_2</td><td>See 6.8.3.4.</td></tr>
<tr><td>UPDATE_3</td><td>See 6.8.3.5.</td></tr>
<tr><td>REMOVE_4</td><td>See 6.8.3.6.</td></tr>
</table></td></tr>
<tr><td>updateValue[]</td><td>DataValue</td><td>New values to be inserted, replaced or removed.</td></tr>
</table>

6.8.3.2 Specfied Uniqueness of Structured History Data

S*tructured History Data* provides metadata describing an entry in the history database. The *Server* shall define what uniqueness means for each *Structured History Data* structure type. For example, a *Server* may only allow one *Annotation* per timestamp which means the timestamp is the unique key for the structure*.* Another *Server* may allow *Annotations* to exist per user, so a combination of a username and timestamp may be used as the unique key for the structure. In 6.8.3.3, 6.8.3.4, 6.8.3.5, and 6.8.3.6 the terms '*Structured History Data* exists' and 'at the specified parameters' means a matching entry has been found at the specified timestamp using the *Server's* criteria for uniqueness.

In the case where the Client wishes to Replace a parameter that is part of the uniqueness criteria, then the resulting StatusCode would be *Bad_NoEntryExists*. The Client shall Remove the existing structure and then Insert the new structure.

6.8.3.3 Insert functionality

Setting performInsertReplace = INSERT_1 inserts *Structured History Data* such as *Annotations* into the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structured History Data* entry already exists at the specified parameters the *StatusCode* shall indicate *Bad_EntryExists*.

If the *Time* does not fall within range that can be stored then the related *operationResults entry* shall indicate *Bad_OutOfRange.*

6.8.3.4 Replace functionality

Setting performInsertReplace = REPLACE_2 replaces *Structured History Data* such as *Annotations* in the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structured History Data* entry does not already exist at the specified parameters, then the *StatusCode* shall indicate *Bad_NoEntryExists*.

6.8.3.5 Update functionality

Setting performInsertReplace = UPDATE_3 inserts or replaces *Structured History Data* such as *Annotations* in the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structure History Data* entry already exists at the specified parameters then it is deleted and the value provided by the *Client* is inserted. If no existing entry exists then the new entry is inserted.

If an existing entry was replaced successfully then the *StatusCode* shall be *Good_EntryReplaced.* If a new entry was created the *StatusCode* shall be *Good_EntryInserted.* If the *Server* cannot determine whether it replaced or inserted an entry then the *StatusCode* shall be *Good.*

If the *Time* does not fall within range that can be stored then the related *operationResults entry* shall indicate *Bad_OutOfRange.*

6.8.3.6 Remove functionality

Setting performInsertReplace = REMOVE_4 removes *Structured History Data* such as *Annotations* from the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structure History Data* entry exists at the specified parameters it is deleted. If *Structured History Data* does not already exist at the specified parameters, then the *StatusCode* shall indicate *Bad_NoEntryExists*.

6.8.4 UpdateEventDetails structure

6.8.4.1 UpdateEventDetails structure detail

Table 31 defines the UpdateEventDetails structure.

Table 31 – UpdateEventDetails

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>UpdateEventDetails</td><td>Structure</td><td>The details for insert, replace, and insert/replace history Event updates.</td></tr>
<tr><td>nodeId</td><td>NodeId</td><td>Node id of the Object to be updated.</td></tr>
<tr><td>performInsertReplace</td><td>PerformUpdateType</td><td>Value determines which action of insert, replace, or update is performed.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>INSERT_1</td><td>Perform Insert Event (see 6.8.4.2).</td></tr>
<tr><td>REPLACE_2</td><td>Perform Replace Event (see 6.8.4.3).</td></tr>
<tr><td>UPDATE_3</td><td>Perform Update Event (see 6.8.4.4).</td></tr>
</table></td></tr>
<tr><td>filter</td><td>EventFilter</td><td>If the history of Notification conforms to the EventFilter, the history of the Notification is updated.</td></tr>
<tr><td>eventData[]</td><td>HistoryEventFieldList</td><td>List of Event Notifications to be inserted or updated (see 6.5.4 for HistoryEventFieldList definition).</td></tr>
</table>

6.8.4.2 Insert event functionality

This function is intended to insert new entries, e.g., backfilling of historical *Events*.

Setting performInsertReplace = INSERT_1 inserts entries into the *Event* history database for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* shall be empty. The *SelectClause* shall as a minimum provide the following Event fields: *EventType* and Time. It is also recommended that the *SourceNode* and the *SourceName* fields are provided. If one of the required fields is not provided then the *statusCode* shall indicate *Bad_ArgumentsMissing*. If the historian does not support archiving the specified *EventType* then the *statusCode* shall indicate *Bad_TypeDefinitionInvalid*. If the *SourceNode* is not a valid

source for *Events* then the related *operationResults entry* shall indicate *Bad_SourceNodeIdInvalid.* If the *Time* does not fall within range that can be stored then the related *operationResults entry* shall indicate *Bad_OutOfRange.* If the *selectClause* does not include fields which are mandatory for the *EventType* then the *statusCode* shall indicate *Bad_ArgumentsMissing.* If the *selectClause* specifies fields which are not valid for the *EventType* or cannot be saved by the historian then the related *operationResults entry* shall indicate *Good_DataIgnored*. Additional information about the ignored fields shall be provided through *DiagnosticInformation* related to the *operationResults*. The symbolicId contains the index of each ignored field separated with a space and the localizedText contains the symbolic names of the ignored fields.

The *EventId* is a *Server* generated opaque value and a *Client* cannot assume that it knows how to create valid *EventIds*. A *Server* shall be able to generate an appropriate default value for the *EventId* field. If a *Client* does specify the *EventId* in the *selectClause* and it matches an existing *Event* then the *statusCode* shall indicate *Bad_EntryExists*. A *Client* shall use a *HistoryRead* to discover any automatically generated *EventIds*.

If any errors occur while processing individual fields then the related o*perationResults entry* shall indicate *Bad_InvalidArgument* and the invalid fields shall be indicated in the *DiagnosticInformation* related to the *operationResults entry*.

The *IndexRange* parameter of the *SimpleAttributeOperand* is not valid for insert operations and the *StatusCode* shall specify Bad*_IndexRangeInvalid* if one is specified.

A *Client* may instruct the *Server* to choose a suitable default value for a field by specifying a value of null. If the *Server* is not able to select a suitable default then the corresponding entry in the operationResults array for the affected *Event* shall be *Bad_InvalidArgument*.

6.8.4.3 Replace event functionality

This function is intended to replace fields in existing *Event* entries, e.g., correct *Event* data that contained incorrect data due to a bad sensor.

Setting performInsertReplace = REPLACE_2 replaces entries in the *Event* history database for the specified *EventIds* for one or more *HistoricalEventNodes*. The *SelectClause* parameter of the *EventFilter* shall specify the *EventId Property* and the *eventData* shall contain the *EventId* which will be used to find the *Event* to be replaced. If no entry exists matching the specified *EventId* then no replace operation will be performed; instead the *operationResults entry* for the *eventData* entry shall indicate *Bad_NoEntryExists*. The *whereClause* parameter of the *EventFilter* shall be empty.

If the *selectClause* specifies fields which are not valid for the *EventType* or cannot be saved or changed by the historian then the *operationResults entry* for the affected *Event* shall indicate *Good_DataIgnored*. Additional information about the ignored fields shall be provided through *DiagnosticInformation* related to the *operationResults*. The symbolicId contains the index of each ignored field separated with a space and the localizedText contains the symbolic names of the ignored fields.

If fatal errors occur while processing individual fields then the *operationResults entry* for the affected Event shall indicate *Bad_InvalidArgument* and the invalid fields shall be indicated in the *DiagnosticInformation* related to the *operationResults entry*.

6.8.4.4 Update event functionality

This function is intended to unconditionally insert/replace *Events*, e.g., synchronizing a backup *Event* database.

Setting performInsertReplace = UPDATE_3 inserts or replaces entries in the *Event* history database for the specified filter for one or more *HistoricalEventNodes*.

The *Server* will, based on its own criteria, attempt to determine if the *Event* already exists; if it does exist then the *Event* will be deleted and the new *Event* will be inserted (retaining the *EventId*). If the *EventID* was provided then the *EventID* will be used to determine if the *Event*

already exists. If the *Event* does not exist then a new *Event* will be inserted, including the generation of a new *EventId*.

All of the restrictions, behaviours, and errors specified for the Insert functionality (see 6.8.4.2) also apply to this function.

If an existing *Event* entry was replaced successfully then the related *operationResults entry* shall be *Good_EntryReplaced.* If a new *Event* entry was created then the related *operationResults entry* shall be *Good_EntryInserted.* If the *Server* cannot determine whether it replaced or inserted an entry then the related *operationResults entry* shall be *Good.*

6.8.5 DeleteRawModifiedDetails structure

6.8.5.1 DeleteRawModifiedDetails structure detail

Table 32 defines the DeleteRawModifiedDetails structure.

Table 32 – DeleteRawModifiedDetails

Name	Type	Description
DeleteRawModifiedDetails	Structure	The details for delete raw and delete modified history updates.
nodeId	NodeId	Node id of the *Object* for which history values are to be deleted.
isDeleteModified	Boolean	TRUE for MODIFIED, FALSE for RAW. Default value is FALSE.
startTime	UtcTime	Beginning of period to be deleted.
endTime	UtcTime	End of period to be deleted.

These functions are intended to be used to delete data that has been accidentally entered into the history database, e.g., deletion of data from a source with incorrect timestamps. Both *startTime* and *endTime* shall be defined. The *startTime* shall be less than the *endTime*, and values up to but not including the *endTime* are deleted. It is permissible for *startTime* = *endTime*, in which case the value at the *startTime* is deleted.

6.8.5.2 Delete raw functionality

Setting isDeleteModified = FALSE deletes all *Raw* entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

If no data is found in the time range for a particular *HistoricalDataNode*, then the *StatusCode* for that item is *Bad_NoData.*

6.8.5.3 Delete modified functionality

Setting isDeleteModified = TRUE deletes all *Modified* entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

If no data is found in the time range for a particular HistoricalDataNode, then the *StatusCode* for that item is *Bad_NoData.*

6.8.6 DeleteAtTimeDetails structure

6.8.6.1 DeleteAtTimeDetails structure detail

Table 33 defines the structure of the DeleteAtTimeDetails structure.

Table 33 – DeleteAtTimeDetails

Name	Type	Description
DeleteAtTimeDetails	Structure	The details for delete raw history updates
nodeId	NodeId	Node id of the *Object* for which history values are to be deleted.
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be deleted.

6.8.6.2 Delete at time functionality

The DeleteAtTime structure deletes all raw values, modified values, and annotations in the history database for the specified timestamps for one or more *HistoricalDataNodes*.

This parameter is intended to be used to delete specific data from the history database, e.g., lab data that is incorrect and cannot be correctly reproduced.

6.8.7 DeleteEventDetails structure

6.8.7.1 DeleteEventDetails structure detail

Table 34 defines the structure of the DeleteEventDetails structure.

Table 34 – DeleteEventDetails

Name	Type	Description
DeleteEventDetails	Structure	The details for delete raw and delete modified history updates.
nodeId	NodeId	Node id of the *Object* for which history values are to be deleted.
eventId[]	ByteString	An array of *EventIds* to identify which *Events* are to be deleted.

6.8.7.2 Delete event functionality

The DeleteEventDetails structure deletes all *Event* entries from the history database matching the *EventId* for one or more *HistoricalEventNodes*.

If no Events are found that match the specified filter for a HistoricalEventNode, then the *StatusCode* for that Node is *Bad_NoData.*

Annex A
(informative)

Client conventions

A.1 How clients may request timestamps

The OPC HDA COM based specifications allowed a *Client* to programmatically request historical time periods as absolute time (Jan 01, 2006 12:15:45) or a string representation of relative time (NOW -5M). The OPC UA specification does not allow for using a string representation to pass date/time information using the standard *Services*.

OPC UA *Client* applications that wish to visually represent date/time in a relative string format shall convert this string format to UTC DateTime values before sending requests to the UA *Server*. It is recommended that all OPC UA *Clients* use the syntax defined in this clause to represent relative times in their user interfaces.

The format for the relative time is:

```
keyword+/-offset+/-offset…
```

where keyword and offset are as specified in Table A.1 below. Whitespace is ignored. The time string shall begin with a keyword. Each offset shall be preceded by a signed integer that specifies the number and direction of the offset. If the integer preceding the offset is unsigned then the value of the preceding sign is assumed (beginning default sign is positive). The keyword refers to the beginning of the specified time period. DAY means the timestamp at the beginning of the current day (00:00 hours, midnight). MONTH means the timestamp at the beginning of the current month, etc.

For example, “DAY -1D+7H30M” could represent the start time for data requested for a daily report beginning at 7:30 in the morning of the previous day (DAY = the first timestamp for today, -1D would make it the first timestamp for yesterday, +7H would take it to 7 a.m. yesterday, +30M would make it 7:30 a.m. yesterday (the + on the last term is carried over from the last term)).

Similarly, “MONTH-1D+5H” would be 5 a.m. on the last day of the previous month, “NOW-1H15M” would be an hour and fifteen minutes ago, and “YEAR+3MO” would be the first timestamp of April 1 this year.

Resolving relative timestamps is based upon what Microsoft has done with Excel, thus for various questionable time strings we have these results:

10-Jan-2001 + 1 MO = 10-Feb-2001

29-Jan-1999 + 1 MO = 28-Feb-1999

31-Mar-2002 + 2 MO = 30-May-2002

29-Feb-2000 + 1 Y = 28-Feb-2001

In handling a gap in the calendar (due to different numbers of days in the month, or in the year), when one is adding or subtracting months or years:

Month: If the answer falls in the gap then it is backed up to the same time of day on the last day of the month.

Year: If the answer falls in the gap (February 29) then it is backed up to the same time of day on February 28.

Note that the above does not hold true for cases of adding or subtracting weeks or days, but only for adding or subtracting months or years which may have different numbers of days in them.

Note that all keywords and offsets are specified in uppercase.

Table A.1 – Time keyword definitions

Keyword	Description
NOW	The current UTC time as calculated on the *Server*.
SECOND	The start of the current second.
MINUTE	The start of the current minute.
HOUR	The start of the current hour.
DAY	The start of the current day.
WEEK	The start of the current week.
MONTH	The start of the current month.
YEAR	The start of the current year.

Table A.2 –Time offset definitions

Offset	Description
S	Offset from time in seconds.
M	Offset from time in minutes.
H	Offset from time in hours.
D	Offset from time in days.
W	Offset from time in weeks.
MO	Offset from time in months.
Y	Offset from time in years.

A.2 Determining the first historical data point

In some cases *Servers* are required to return the first available data point for a historical *Node;* this clause recommends the way that a *Client* should request this information so that Servers can optimize this call, if desired. Although there are multiple calls that could return the first data value, the recommended practice will be to use the *StartOfArchive Property*. If this *Property* isn't available then use the following ReadRawModifiedDetails parameters:

returnBounds=false

numValuesPerNode=1

startTime=DateTime.MinValue+1 second

endTime= DateTime.MinValue

_____