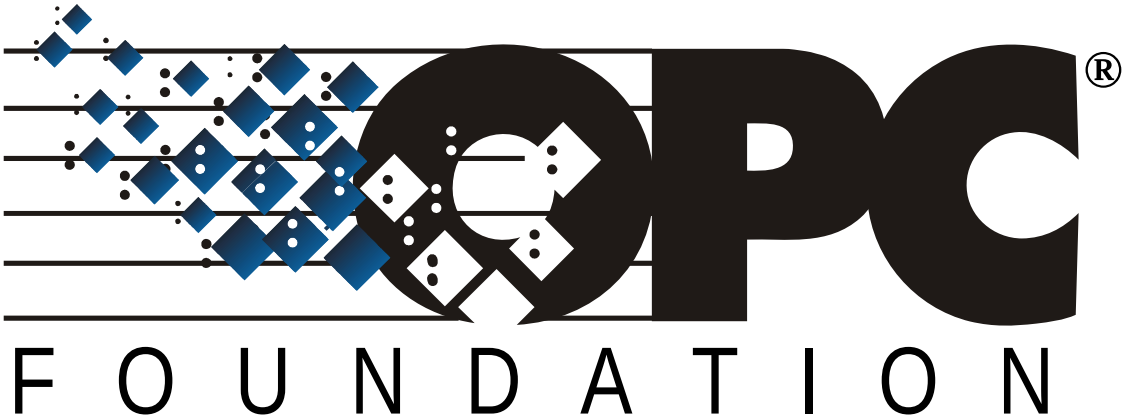

OPC Unified Architecture

Specification

Part 6: Mappings

Release 1.00

February 6, 2009

Specification Type:	Industry Standard Specification	Comments:	Report or view errata: http://www.opcfoundation.org/errata
Title:	OPC Unified Architecture Part 6 :Mappings	Date:	February 6, 2009
Version:	Release 1.00	Software:	MS-Word
		Source:	OPC UA Part 6 - Mappings 1.00 Specification.doc
Author:	OPC Foundation	Status:	Release

CONTENTS

Page

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

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

AGREEMENT OF USE

COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

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: http://www.opcfoundation.org/errata

OPC Unified Architecture Specification

Part 6: Mappings

1 Scope

This document specifies the OPC Unified Architecture (OPC UA) mapping between the security model described in Part 2, the abstract service definitions described Part 4 and the data structures defined in Part 5 and the physical network protocols that can be used to implement the OPC UA specification.

2 Reference Documents

Part 1: OPC UA Specification: Part 1 – Concepts, Version 1.01 or later
http://www.opcfoundation.org/UA/Part1/

Part 2: OPC UA Specification: Part 2 – Security Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part2/

Part 3: OPC UA Specification: Part 3 – Address Space Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part3/

Part 4: OPC UA Specification: Part 4 – Services, Version 1.01 or later
http://www.opcfoundation.org/UA/Part4/

Part 5: OPC UA Specification: Part 5 – Information Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part5/

Part 7: OPC UA Specification: Part 7 – Profiles, Version 1.0 or later
http://www.opcfoundation.org/UA/Part7/

XML Schema Part 1: XML Schema Part 1: Structures
http://www.w3.org/TR/xmlschema-1/

XML Schema Part 2: XML Schema Part 2: Datatypes
http://www.w3.org/TR/xmlschema-2/

SOAP Part 1: SOAP Version 1.2 Part 1: Messaging Framework
http://www.w3.org/TR/soap12-part1/

SOAP Part 2: SOAP Version 1.2 Part 2: Adjuncts
http://www.w3.org/TR/soap12-part2/

XML Encryption: XML Encryption Syntax and Processing
http://www.w3.org/TR/xmlenc-core/

XML Signature: XML-Signature Syntax and Processing
http://www.w3.org/TR/xmldsig-core/

WS Security: SOAP Message Security 1.1
http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

WS Addressing: Web Services Addressing (WS-Addressing)
http://www.w3.org/Submission/ws-addressing/

WS Trust: WS Trust 1.3
http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html

WS Secure Conversation: WS Secure Conversation 1.3
http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html

WS Security Policy: WS Security Policy 1.2
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html

SSL/TLS: RFC 2246 - The TLS Protocol Version 1.0
http://www.ietf.org/rfc/rfc2246.txt

X509: X.509 Public Key Certificate Infrastructure
http://www.itu.int/rec/T-REC-X.509-200003-I/e

WS-I Basic Profile 1.1: WS-I Basic Profile Version 1.1
http://www.ws-i.org/Profiles/BasicProfile-1.1.html

WS-I Basic Security Profile 1.1: WS-I Basic Security Profile Version 1.1
http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html

HTTP: RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1
http://www.ietf.org/rfc/rfc2616.txt

HTTPS: RFC 2818 - HTTP Over TLS
http://www.ietf.org/rfc/rfc2818.txt

Base64: RFC 3548 - The Base16, Base32, and Base64 Data Encodings
http://www.ietf.org/rfc/rfc3548.txt

X690 : ITU-T X.690 - Basic (BER), Canonical (CER) and Distinguished (DER) Encoding Rules
http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

X200 : ITU-T X.200 – Open Systems Interconnection – Basic Reference Model
http://www.itu.int/rec/T-REC-X.200-199407-I/en

IEEE-754: Standard for Binary Floating-Point Arithmetic
http://grouper.ieee.org/groups/754/

HMAC: HMAC - Keyed-Hashing for Message Authentication
http://www.ietf.org/rfc/rfc2104.txt

PKCS #1 : PKCS #1 - RSA Cryptography Specifications Version 2.0
http://www.ietf.org/rfc/rfc2437.txt

PKCS #12 : PKCS 12 v1.0: Personal Information Exchange Syntax
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf

FIPS 180-2: Secure Hash Standard (SHA)
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

FIPS 197: Advanced Encyption Standard (AES)

http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf

UTF8: UTF-8, a transformation format of ISO 10646
http://tools.ietf.org/html/rfc3629

RFC 3280: RFC 3280 - X.509 Public Key Infrastructure Certificate and CRL Profile
http://www.ietf.org/rfc/rfc3280.txt

IPSec: RFC 2401 - Security Architecture for the Internet Protocol
http://tools.ietf.org/html/rfc2401

RFC 4514: RFC 4514 - LDAP: String Representation of Distinguished Names
http://www.ietf.org/rfc/rfc4514.txt

3 Terms, definitions, and conventions

3.1 OPC UA Part 1 terms

The following terms defined in Part 1 apply.

AddressSpace
Attribute
Certificate
Message
Node
Profile

3.2 OPC UA Part 2 terms

The following terms defined in Part 2 apply.

OPC UA application
Authentication
Integrity
Authorization
X.509 Certificate
SoftwareCertificate
SecurityToken
SecureChannel
PublicKey
PrivateKey
Nonce
Application Certificate
Software Certificate

3.3 OPC UA Part 3 terms

The following terms defined in Part 3 apply.

BaseDataType

BaseObjectType

DataType

DataTypeEncoding Structure

3.4 OPC UA Part 4 terms

The following terms defined in Part 4 apply.

SecurityMode (None, Sign and SignAndEncrypt)

3.5 OPC UA Mappings terms

3.5.1 Data Encoding

Data Encoding is way to serialize OPC UA messages and data structures.

3.5.2 Mapping

A *Mapping* specifies how to implement an OPC UA feature with a specific technology. For example, the OPC UA Binary Encoding is a *Mapping* that specifies how to serialize OPC UA data structures as sequences of bytes.

3.5.3 Security Protocol

A *Security Protocol* ensures the integrity and privacy of UA messages that are exchanged between OPC UA applications.

3.5.4 Stack

A *Stack* is a collection of software libraries that implement one or more *Stack Profiles*. *Stacks* have an API which hides the implementation details from the application developer.

3.5.5 Stack Profile

A *StackProfile* is a combination of *DataEncodings*, *SecurityProtocol* and *TransportProtocol Mappings*. OPC UA applications shall implement one or more *StackProfiles* and can only communicate with OPC UA applications that support a *StackProfile* that they support.

3.5.6 Transport Protocol

A *Transport Protocol* is a way to exchange serialized OPC UA messages between OPC UA applications.

3.6 Abbreviations and symbols

API	Application Programming Interface
ASN.1	Abstract Syntax Notation #1 (used in X690)
BP	WS-I Basic Profile Version
BSP	WS-I Basic Security Profile
CSV	Comma Separated Value (File Format)
HTTP	Hypertext Transfer Protocol
RST	Request Security Token
OID	Object Identifier (used with ASN.1)
RSTR	Request Security Token Response
SCT	Security Context Token
SHA1	Secure Hash Algorithm

SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer (Defined in SSL/TLS)
TCP	Transmission Control Protocol
TLS	Transport Layer Security (Defined in SSL/TLS)
UTF8	Unicode Transformation Format (8-bit) (Defined in UTF8
UA	Unified Architecture
UASC	UA Secure Conversation
WS-*	The XML Web Services Specifications.
WSS	WS Security
WS-SC	WS Secure Conversation
XML	Extensible Markup Language

4 Overview

The other parts of the OPC UA specification are written to be independent of the technology used for implementation. This approach means OPC UA is a flexible specification that will continue to be applicable as technology evolves. On the other hand, this approach means that it is not possible to build an OPC UA application with the information contained in Parts 1 through 5 because important implementation details have been left out.

This document defines *Mappings* between the abstract specifications and technologies that can be used to implement them. The *Mappings* are organized into three groups: *DataEncodings, SecurityProtocols* and *TransportProtocols*. Different *Mappings* are combined together to create *StackProfiles*. All OPC UA applications shall implement at least one *StackProfile* and can only communicate with other OPC UA applications that implement the same *StackProfile.*

This document defines the *DataEncodings* in Clause 0, the *SecurityProtocols* in Clause 6 and the *TransportProtocols* in Clause 6.4.6. The *StackProfiles* are defined in Part 7.

All communication between OPC UA applications is based on the exchange of *Messages*. The parameters contained in the *Messages* are defined in Part 4, however, their format is specified by the *DataEncoding* and *TransportProtocol*. For this reason, each *Message* defined in Part 4 shall have a normative description which specifies exactly what shall be put on the wire. The normative descriptions are defined in the appendices.

A *Stack* is a collection of software libraries that implements one or more *StackProfiles*. The interface between an OPC UA application and the *Stack* is a non-normative API which hides the details of the *Stack* implementation. An API depends on a specific *DevelopmentPlatform*. Note that the datatypes exposed in the API for a *DevelopmentPlatform* may not match the datatypes defined by the specification because of limitations of the *DevelopmentPlatform.* For example, Java does not support unsigned integers which means any Java API will need to map unsigned integers onto a signed integer type.

Figure 1 illustrates the relationships between the different concepts defined in this document.

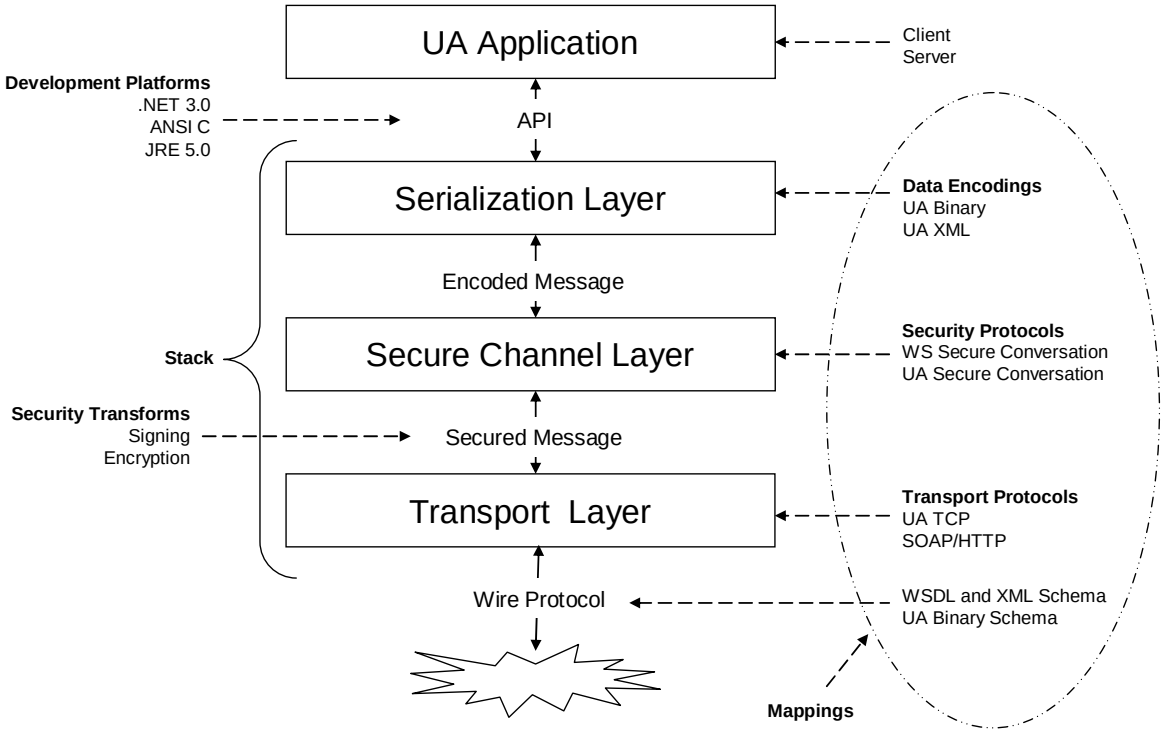

Figure 1 – The OPC UA Stack Overview

The layers described in this specification do not correspond to layers in the OSI 7 layer model [X200]. Each OPC UA *StackProfile* should be treated as a single Layer 7 (Application) protocol that is built on an existing Layer 5, 6 or 7 protocol such as TCP/IP, TLS or HTTP.The *SecureChannel* layer is always present even if the *SecurityMode* is None. In this situation, no security is applied but the *SecurityProtocol* implementation shall maintain a logical channel with a unique identifier. Users and Administrators are expected to understand that a *SecureChannel* with *SecurityMode* set to None cannot be trusted unless the Application is operating on a physically secure network or a low level protocol such as IPSec is being used.

5 Data Encoding

5.1 General

5.1.1 Overview

This specification defines two data encodings: OPC UA Binary and OPC UA XML. It describes how to construct messages using each of these encodings.

5.1.2 Built-in Types

All OPC UA *DataEncodings* are based on rules that are defined for a standard set of built-in types. These built-in types are then used to construct structures, arrays and messages. The built-in types are described in Table 1.

Table 1 – Built-in Data Types

ID	Name	Description
1	Boolean	A two-state logical value (true or false).
2	SByte	An integer value between -128 and 127.
3	Byte	An integer value between 0 and 255.
4	Int16	An integer value between -32768 and 32767.
5	UInt16	An integer value between 0 and 65535.
6	Int32	An integer value between – 2147483648 and 2147483647.
7	UInt32	An integer value between 0 and 4294967295.
8	Int64	An integer value between – 9223372036854775808 and 9223372036854775807
9	UInt64	An integer value between 0 and 18446744073709551615.
10	Float	An IEEE single precision (32 bit) floating point value.
11	Double	An IEEE double precision (64 bit) floating point value.
12	String	A sequence of Unicode characters.
13	DateTime	An instance in time.
14	Guid	A 16 byte value that can be used as a globally unique identifier.
15	ByteString	A sequence of octets.
16	XmlElement	An XML element.
17	NodeId	An identifier for a node in the address space of an OPC UA server.
18	ExpandedNodeId	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	A numeric identifier for a error or condition that is associated with a value or an operation.
20	QualifiedName	A name qualified by a namespace.
21	LocalizedText	Human readable text with an optional locale identifier.
22	ExtensionObject	A structure that contains an application specific data type that may not be recognized by the receiver.
23	DataValue	A data value with an associated status code and timestamps.
24	Variant	A union of all of the types specified above.
25	DiagnosticInfo	A structure that contains detailed error and diagnostic information associated with a StatusCode.

Most of these data types are the same as the abstract types defined in Part 3 and Part 4, however, *the ExtensionObject* and *Variant* types are defined in this document. In addition, this document defines a representation for the *Guid* type defined in Part 3.

5.1.3 Guid

A *Guid* is a 16-byte globally unique identifier with the layout shown in Table 2.

Table 2 – Guid Structure

Component	Data Type
Data1	UInt32
Data2	UInt16
Data3	UInt16
Data4	Byte[8]

Guid values may be represented as a string in this form:

```
<Data1>-<Data2>-<Data3>-<Data4[0:1]>-<Data4[2:7]>
```

Where Data1 is 8 characters wide, Data2 and Data3 are 4 characters wide and each *Byte* in Data4 is 2 characters wide. Each value is formatted as a hexadecimal number padded with zeros. A typical *Guid* value would look like this when formatted as a string:

```
C496578A-0DFE-4b8f-870A-745238C6AEAE
```

5.1.4 ExtensionObject

An *ExtensionObject* is a container for any complex data types which cannot be encoded as one of the other built-in data types. The *ExtensionObject* contains a complex value serialized as a sequence of bytes or as an XML element. It also contains an identifier which indicates what data it contains and how it is encoded.

Complex data types are represented in a *Server* address space as sub-types of the *Structure* data type. The encodings available for any given complex data type are represented as a *DataTypeEncoding Object* in the *Server* address space. The *NodeId* for the *DataTypeEncoding Object* is the identifier stored in the *ExtensionObject*. Clause 5.8 of Part 3 describes how *DataTypeEncoding* node are related to other nodes the address space.

Server implementers should use namespace qualified numeric *NodeIds* for any *DataTypeEncoding Objects* they define. This will minimize the overhead introduced by packing complex data values into *ExtensionObjects*.

5.1.5 Variant

A *Variant* is a union of all built-in data types including an *ExtensionObject*. *Variants* can also contain arrays of any of these built-in types. *Variants* are used to store any value or parameter with a data type of *BaseDataType* or one of its subtypes.

Variants can be empty. An empty *Variant* is described as having a *Null* value and should be treated like a NULL column in a SQL database. A *Null* value in a *Variant* may not be the same as a *Null* value for data types that support *Nulls* such as *Strings*. For this reason, all *DataEncodings* shall preserve this distinction when encoding *Variants*.

Variants can contain arrays of *Variants* but they cannot directly contain another *Variant*.

DiagnosticInfo type only has meaning when returned in a response message with an associated *StatusCode*. As a result, *Variants* cannot contain instances of *DiagnosticInfo.*

Variables with a *DataType* of *BaseDataType* are mapped to a *Variant*, however, the *ValueRank* and *ArrayDimensions Attributes* place restrictions on what is allowed in the *Variant*. For example, if the *ValueRank* is *Scalar* then the *Variant* may only contain scalar values.

5.2 OPC UA Binary

5.2.1 General

The OPC UA Binary Encoding is a data format developed to meet the performance needs of OPC UA applications. This format is designed primarily for fast encoding and decoding, however, the size of the encoded data on the wire was also a consideration.

The OPC UA Binary Encoding relies on several primitive data types with clearly defined encoding rules that can be sequentially written to or read from a binary stream. A structure is encoded by sequentially writing the encoded form of each field. If a given field is also a structure then the values of its fields are written sequentially before writing the next field in the containing structure. All fields shall be written to the stream even if they contain *Null* values. The encodings for each primitive type specify how to encode either a *Null* or a default value for the type.

The OPC UA Binary Encoding does not include any type or field name information because all OPC UA applications are expected to have advance knowledge of the services and structures that they support. An exception is an *ExtensionObject* which provides an identifier and a size for the complex structure it represents. This allows a decoder to skip over types that it does not recognize.

5.2.2 Built-in Types

5.2.2.1 Boolean

A *Boolean* value shall be encoded as a single byte where a value of 0 (zero) is false and any non-zero value is true.

Encoders shall use the value of 1 to indicate a true value; however, decoders shall treat any non-zero value as true.

5.2.2.2 Integer

All integer types shall be encoded as little endian values where the least significant byte appears first in the stream.

The Figure 2 illustrates how the value 1,000,000,000 (Hex: 3B9ACA00) should be encoded as a 32 bit integer in the stream.

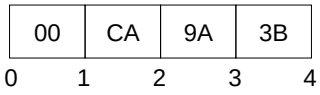

Figure 2 – Encoding Integers in a Binary Stream

5.2.2.3 Floating Point

All floating point values shall be encoded with the appropriate IEEE-754 binary representation which has three basic components: the sign, the exponent, and the fraction. The bit ranges assigned to each component depend on the width of the type. Table 3 lists the bit ranges for the supported floating point types.

Table 3 – Supported Floating Point Types

Name	Width (bits)	Fraction	Exponent	Sign
Float	32	0-22	23-30	31
Double	64	0-51	52-62	63

In addition, the order of bytes in the stream is significant. All floating point values shall be encoded with the least significant byte appearing first (i.e. little endian).

The Figure 3 illustrates how the value -6.5 (Hex: C0D00000) should be encoded as a *Float*.

The floating point type supports positive and negative infinity and not-a-number (NaN). The IEEE specification allows for multiple NaN variants, however, the encoders/decoders may not preserve the distinction. Encoders shall encode a NaN value as an IEEE quiet-NAN (000000000000F8FF) or (0000C0FF). Any unsupported types such as denormalized numbers shall also be encoded as a IEEE quiet-NAN.

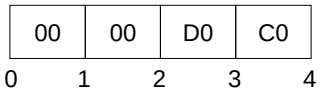

Figure 3 – Encoding Floating Points in a Binary Stream

5.2.2.4 String

All *String* values are encoded as a sequence of UTF8 characters without a null terminator and preceded by the length in bytes.

The length in bytes is encoded as *Int32*. A value of -1 is used to indicate a 'null' string.

Figure 4 illustrates how the multilingual string " Boy" should be encoded in a byte stream.

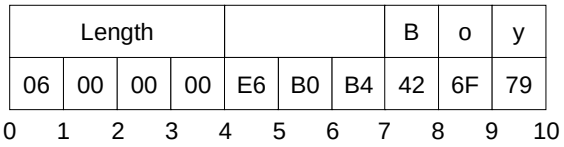

Figure 4 – Encoding Strings in a Binary Stream

5.2.2.5 DateTime

A *DateTime* value shall be encoded as a 64-bit signed integer (see Clause 5.2.2.2) which represents the number of 100 nanosecond intervals since January 1, 1601 (UTC)[1].

Not all platforms will be able to represent the full range of dates and times that can be represented with this encoding. For example, the UNIX time_t structure only has a 1 second resolution and cannot represent dates prior to 1970. For this reason, a number of rules shall be applied when dealing with date/time values that exceed the dynamic range of a platform. These rules are:

1) A date/time value is encoded as 0 if either:
    a. The value equal to or earlier than 1601-01-01 12:00AM
    b. The value is the earliest date that can be represented with the platform's encoding.
2) A date/time is encoded as the maximum value for an *Int64* if either:
    a. The value is equal to or greater than 9999-01-01 11:59:59PM
    b. The value is the latest date that can be represented with the platform's encoding
3) A date/time is decoded as the earliest time that can be represented on the platform if either:
    a. The encoded value is 0
    b. The encoded value represents a time earlier than the earliest time that can be represented with the platform's encoding.
4) A date/time is decoded as the latest time that can be represented on the platform if either:
    a. The encoded value is the maximum value for an *Int64*
    b. The encoded value represents a time later than the latest time that can be represented with the platform's encoding.

These rules imply that the earliest and latest times that can be represented on a given platform are invalid date/time values and should be treated that way by applications.

A decoder shall truncate the value if a decoder encounters a DateTime value with a resolution that is greater than the resolution supported on the platform.

[1] This is the same as the binary representation of the WIN32 FILETIME type, within the OPC UA allowed range. UTC is the timezone used for the starting time

5.2.2.6 Guid

A *Guid* is encoded a the structure shown in Table 2. The fields are encoded sequentially according to the data type for the field.

Figure 5 illustrates how the *Guid* “72962B91-FA75-4AE6-8D28-B404DC7DAF63” should be encoded in a byte stream.

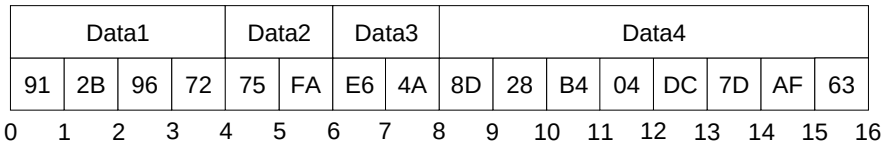

Figure 5 – Encoding Guids in a Binary Stream

5.2.2.7 ByteString

A *ByteString* is encoded as sequence of bytes preceded by its length in bytes. The length is encoded as a 32-bit signed integer as described above.

If the length of the byte string is -1 then the byte string is ‘null’.

5.2.2.8 XmlElement

An *XmlElement* is an XML fragment serialized as UTF-8 string and then encoded as *ByteString*

Figure 6 illustrates how the *XmlElement* “<A>Hot </A>” should be encoded in a byte stream.

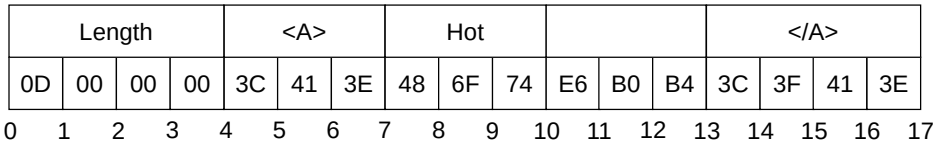

Figure 6 – Encoding XmlElements in a Binary Stream

5.2.2.9 NodeId

The components of a *NodeId* are described the Table 4.

Table 4 – NodeId Components

Name	Data Type	Description
Namespace	UInt16	The index for a namespace URI. An index of 0 is used for OPC UA defined *NodeIds.*
IdentifierType	Enum	The format and data type of the identifier The value may be one of the following: NUMERIC - the value is an *UInteger;* STRING - the value is *String*; GUID - the value is a *Guid*; OPAQUE - the value is a *ByteString*;
Value	*	The identifier for a node in the address space of an OPC UA server.

The encoding of a *NodeId* varies according to the contents of the instance. For that reason the first byte of the encoded form indicates the format of the rest of the encoded *NodeId*. The possible encoding formats are shown in Table 5. The tables that follow describe the structure of the each possible format (they exclude the byte which indicates the format).

Table 5 – NodeId Encoding Values

Name	Value	Description
Two Byte	0x00	A numeric value that fits into the two byte representation.
Four Byte	0x01	A numeric value that fits into the four byte representation.
Numeric	0x02	A numeric value that does not fit into the two or four byte representations.
String	0x03	A String value.
Guid	0x04	A Guid value.
ByteString	0x05	An opaque (ByteString) value.
NamespaceUri Flag	0x80	See discussion of *ExpandedNodeId* in Clause 5.2.2.10.
ServerIndex Flag	0x40	See discussion of *ExpandedNodeId* in Clause 5.2.2.10.

The standard *NodeId* encoding has the structure shown in Table 6. The standard encoding is used for all formats that do not have an explicit format defined.

Table 6 – Standard NodeId Binary Encoding

Name	Data Type	Description
Namespace	UInt16	The Namespace index.
Identifier	*	The identifier which is encoded according to the following rules: NUMERIC UInt32 STRING String GUID Guid OPAQUE ByteString

An example of a String *NodeId* with Namespace = 1 and Identifier = “Hot ” is shown in Figure 7.

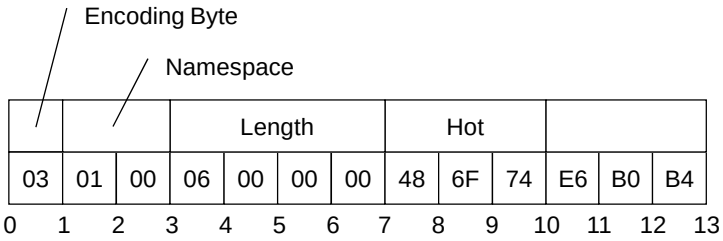

Figure 7 – A String NodeId

The Two Byte *NodeId* encoding has the structure shown in Table 7.

Table 7 – Two Byte NodeId Binary Encoding

Name	Data Type	Description
Identifier	Byte	The Namespace is the default OPC UA namespace (i.e. 0). The Identifier Type is ‘Numeric’. The Identifier shall be in the range 0 to 255.

An example of a Two Byte *NodeId* with Identifier = 72 is shown in Figure 8.

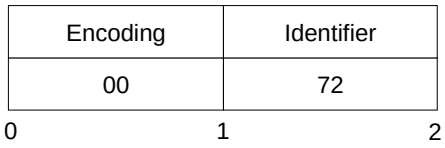

Figure 8 – A Two Byte NodeId

The Four Byte *NodeId* encoding has the structure shown in Table 8.

Table 8 – Four Byte NodeId Binary Encoding

Name	Data Type	Description
Namespace	Byte	The Namespace shall be in the range 0-255.
Identifier	UInt16	The Identifier Type is 'Numeric'. The Identifier shall be an integer in the range 0-65535.

An example of a Four Byte *NodeId* with Namespace = 5 and Identifier = 1025 is shown in Figure 9.

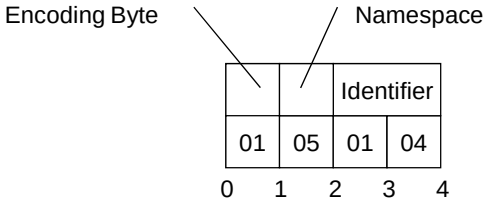

Figure 9 – A Four Byte NodeId

5.2.2.10 ExpandedNodeId

An *ExpandedNodeId* extends the *NodeId* structure by allowing the *NamespaceUri* to be explicitly specified instead of using the *NamespaceIndex*. The *NamespaceUri* is optional. If it is specified then the *NamespaceIndex* inside the *NodeId* shall be ignored.

The *ExpandedNodeId* is encoded by first encoding a *NodeId* as described in Clause 5.2.2.9 and then encoding *NamespaceUri* as a *String*.

An instance of an *ExpandedNodeId* may still use the *NamespaceIndex* instead of the *NamespaceUri*. In this case, the *NamespaceUri* is not encoded in the stream. The presence of the *NamespaceUri* in the stream is indicated by setting the *NamespaceUri* flag in the encoding format byte for the *NodeId*.

If the *NamespaceUri* is present then the encoder shall encode the *NamespaceIndex* as 0 in the stream when the *NodeId* portion is encoded. The unused *NamespaceIndex* is included in the stream for consistency,

An *ExpandedNodeId* may also have a *ServerIndex* which is encoded as a *UInt32* after the *NamespaceUri.* The *ServerIndex* flag in the *NodeId* encoding byte indicates whether the *ServerIndex* is present in the stream. The *ServerIndex* is omitted if it is equal to zero.

The *ExpandedNodeId* encoding has the structure shown in Table 9.

Table 9 – ExpandedNodeId Binary Encoding

Name	Data Type	Description
NodeId	NodeId	The NamespaceUri and ServerIndex flags in the NodeId encoding indicate whether those fields are present in the stream.
NamespaceUri	String	Not present if Null or Empty.
ServerIndex	UInt32	Not present if 0.

5.2.2.11 StatusCode

A *StatusCode* is encoded as a *UInt32*.

5.2.2.12 DiagnosticInfo

A *DiagnosticInfo* structure is described in Part 4 Clause 7.8. It specifies a number of fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

As described in Part 4, the SymbolicId, NamespaceUri, LocalizedText and Locale fields are indexes in a string table which is returned in the response header. Only the index of the string in this table is encoded. An index of -1 indicates that there is no value for the string.

Table 10 – DiagnosticInfo Binary Encoding

Name	Data Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 Symbolic Id 0x02 Namespace 0x04 LocalizedText 0x08 Locale 0x10 Additional Info 0x20 InnerStatusCode 0x40 InnerDiagnosticInfo
SymbolicId	Int32	A symbolic name for the status code.
NamespaceUri	Int32	A namespace that qualifies the symbolic id.
LocalizedText	Int32	A human readable summary of the status code.
Locale	Int32	The locale used for the localized text.
Additional Info	String	Detailed application specific diagnostic information.
Inner StatusCode	StatusCode	A status code provided by an underlying system.
Inner DiagnosticInfo	DiagnosticInfo	Diagnostic info associated with the inner status code.

5.2.2.13 QualifiedName

A *QualifiedName* structure is encoded as shown in Table 11.

The abstract *QualifiedName* structure is defined in Part 3 Clause 8.4.

Table 11 – QualifiedName Binary Encoding

Name	Data Type	Description
NamespaceIndex	UInt16	The namespace index.
Name	String	The name.

5.2.2.14 LocalizedText

A *LocalizedText* structure contains two fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

The abstract *LocalizedText* structure is defined in Part 3 Clause 8.6.

Table 12 – LocalizedText Binary Encoding

Name	Data Type	Description
EncodingMask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 Locale 0x02 Text
Locale	String	The locale. Omitted is null or empty.
Text	String	The text in the specified locale. Omitted is null or empty.

5.2.2.15 ExtensionObject

An *ExtensionObject* is encoded as sequence of bytes prefixed by the *NodeId* of its *DataTypeEncoding* and the number of bytes encoded.

An *ExtensionObject* may be encoded by the application which means it is passed as a *ByteString* or an *XmlElement* to the encoder. In this case, the encoder will be able to write the number of bytes in the object before it encodes the bytes. However, an *ExtensionObject* may know how to encode/decode itself which means the encoder shall calculate the number of bytes before it encodes the object or it shall be able seek backwards in the stream and update the length after encoding the body.

When a decoder encounters an *ExtensionObject* it shall check if it recognizes the *DataTypeEncoding* identifier. If it does then it can call the appropriate function to decode the object body. If the decoder does not recognize the type it shall use the EncodingMask to determine if the body is a *ByteString* or an *XmlElement* and then decode the object body.

The serialized form of a *ExtensionObject* is shown in Table 13.

Table 13 – Extension Object Binary Encoding

Name	Data Type	Description
TypeId	NodeId	The identifier for the DataTypeEncoding node in the server's address space. *ExtensionObjects* defined by the OPC UA specification have a numeric node identifier assigned to them with a *NamespaceIndex* of 0. The numeric identifiers are defined in Annex A.1.
Encoding	Byte	An enumeration that indicates how the body is encoded. The parameter may have the following values: 0x00 No body is encoded, 0x01 The body is encoded as a ByteString. 0x02 The body is encoded as a XmlElement.
Length	Int32	The length of the object body. The length shall always be specified.
Body	Byte[*]	The object body. This field contains the raw bytes for ByteString bodies. For XmlElement bodies this field contains the XML encoded as a UTF-8 string without any null terminator.

ExtensionObjects are used in two contexts: as values contained in *Variant* structures or as parameters in OPC UA messages.

5.2.2.16 Variant

An *Variant* is a union of the built-in types.

The structure of an *Variant* is shown in Table 14.

Table 14 – Variant Binary Encoding

Name	Data Type	Description
EncodingMask	Byte	The type of data encoded in the stream. The mask has the following bits assigned: 0:5 Built-in Type Id (see Table 1) 6 True if the Array Dimensions field is encoded. 7 True if an array of values is encoded.
ArrayLength	Int32	The number of elements in the array. This field is only present if the array bit is set in the encoding mask. Multi-dimensional arrays are encoded as a one dimensional array and this field specifies the total number of elements. The original array can be reconstructed from the dimensions that are encoded after the value field. Higher rank dimensions are serialized first. e.g. an array with dimensions [2,2,2] is written in this order: [0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], [1,1,1]
Value	*	The value encoded according to its data type. If the array bit is set in the encoding mask then each element in the array is encoded sequentially. Since many types have variable length encoding each element shall be decoded in order. The value shall not be an *Variant* but it could be an array of *Variants*. Many implementation platforms do not distinguish between one dimensional Arrays of Bytes and ByteStrings. For this reason, decoders are allowed to automatically convert an Array of Bytes to a ByteString.
ArrayDimensions	Int32[]	The length of each dimension. This field is only present if the array dimensions flag is set in the encoding mask. The lower rank dimensions appear first in the array.

The possible types and their identifiers that can be encoded in an Variant are shown in Table 1.

5.2.2.17 DataValue

A *DataValue* is always preceded by a mask that indicates which fields are present in the stream.

The fields of a *DataValue* are described in Table 15.

Table 15 – Data Value Binary Encoding

Name	Data Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 False if the Value is *Null* 0x02 False if the StatusCode is Good 0x04 False if the Source Timestamp is *DateTime.MinValue*. 0x08 False if the Server Timestamp is *DateTime.MinValue*. 0x10 False if the Source Picoseconds is 0. 0x20 False if the Server Picoseconds is 0.
Value	Variant	The value. Not present if the Value bit in the EncodingMask is False.
Status	StatusCode	The status associated with the value. Not present if the StatusCode bit in the EncodingMask is False.
SourceTimestamp	DateTime	The source timestamp associated with the value. Not present if the SourceTimestamp bit in the EncodingMask is False.
SourcePicoseconds	UInt16	The number of 10 picoseconds intervals for the SourceTimestamp. Not present if the SourcePicoseconds bit in the EncodingMask is False. If the source timestamp is missing the picoseconds are ignored.
ServerTimestamp	DateTime	The server timestamp associated with the value. Not present if the ServerTimestamp bit in the EncodingMask is False.
ServerPicoseconds	UInt16	The number of 10 picoseconds intervals for the ServerTimestamp. Not present if the ServerPicoseconds bit in the EncodingMask is False. If the server timestamp is missing the picoseconds are ignored.

The Picoseconds fields store the difference between a high resolution timestamp with a resolution of 10ps and the Timestamp field value which only has a 100ns resolution. The Picoseconds fields shall contain values less than 10000. The decoder shall treat values greater than or equal to 10000 as the value ‘9999’.

5.2.3 Enumerations

Enumerations are encoded as Int32 values.

5.2.4 Arrays

Arrays that occur outside of a *Variant* are encoded as a sequence of elements preceded by the number of elements encoded as an Int32 value. If an array is *Null* then its length is encoded as -1. An array of zero length is different from an array that is *Null* so encoders and decoders shall preserve this distinction.

Multi-dimensional arrays can only be encoded within a Variant.

5.2.5 Structures

Structures are encoded as a sequence of fields in the order that they appear in the definition. The encoding for each field is determined by the data type for the field.

All fields specified in the complex type shall be encoded.

Structures do not have a *Null* value. If an encoder is written in a programming language that allows structures to have null values then the encoder shall create a new instance with default values for all fields and serialize that. Encoders shall not generate an encoding error in this situation.

The following is an example of a structure using C++ syntax:

```
class Type2
{
      int A;
      int B;
};

class Type1
{
      int    X;
      int    NoOfY;
      Type2* Y;
      int    Z;
};
```

The Y field is a pointer to an array with a length stored in NoOfY.

An instance of *Type1* which contains an array of two Type2 instances would be encoded as 37 byte sequence. If the instance of Type1 was encoded in an *ExtensionObject* it would have the encoded form shown in Table 16. The TypeId, Encoding and the Length are fields defined by the *ExtensionObject*. The encoding of the Type2 instances do not include any type identifier because it is explicitly defined in Type1.

Table 16 – Sample OPC UA Binary Encoded Structure

Field	Bytes	Value
Type Id	4	The identifier for Type1
Encoding	1	0x1 for ByteString
Length	4	28
X	4	The value of field ‘X’
NoOfY	4	2
Y.A	4	The value of field ‘Y[0].A’
Y.B	4	The value of field ‘Y[0].B’
Y.A	4	The value of field ‘Y[1].A’
Y.B	4	The value of field ‘Y[1].B’
Z	4	The value of field ‘Z’

5.2.6 Messages

Messages are encoded as *ExtensionObjects*. The parameters in each message are serialized in the same way the fields of a structure are serialized. The Type Id field contains the DataTypeEncoding identifier for the message. The Length field is omitted since the messages are defined by the OPC UA specification.

Each UA service described in Part 4 has a request and response message. The DataTypeEncoding ids assigned to each service are in Annex A.1.

5.3 XML

5.3.1 Built-in Types

5.3.1.1 General

Most built-in types are encoded in XML using the formats defined in XML Schema Part 2 specification. Any special restrictions or usages are discussed below. Some of the built-in types have an XML Schema defined for them using the syntax defined in XML Schema Part 1.

The prefix *xs:* is used to denote a symbol defined by the XML Schema specification.

5.3.1.2 Boolean

A Boolean value is encoded as an *xs:boolean* value.

5.3.1.3 Integer

Integer values are encoded using one of the sub types of the *xs:decimal* type. The mappings between the OPC UA integer types and XML schema data types are shown in Table 17.

Table 17 – XML Data Type Mappings for Integers

Name	XML Type
SByte	xs:byte
Byte	xs:unsignedByte
Int16	xs:short
UInt16	xs:unsignedShort
Int32	xs:int
UInt32	xs:unsignedInt
Int64	xs:long
UInt64	xs:unsignedLong

5.3.1.4 Floating Point

Floating point values are encoded using one of the XML floating point types. The mappings between the OPC UA floating point types and XML schema data types are shown in Table 18.

Table 18 – XML Data Type Mappings for Floating Points

Name	XML Type
Float	xs:float
Double	xs:double

The XML floating point type supports positive infinity (INF), negative infinity (-INF) and not-a-number (NaN).

5.3.1.5 String

A *String* value is encoded as an *xs:string* value.

5.3.1.6 DateTime

A *DateTime* value is encoded as an *xs:dateTime* value.

All DateTime values shall be encoded as UTC times or with the time zone explicitly specified.

Correct:

```
2002-10-10T00:00:00+05:00
2002-10-09T19:00:00Z
```

Incorrect:

```
2002-10-09T19:00:00
```

It is recommended that all *xs:dateTime* values be represented in UTC format.

The earliest and latest date/time values that can be represented on a platform have special meaning and shall not be literally encoded in XML.

The earliest date/time value on a platform shall be encoded in XML as '0001-01-01T00:00:00Z'.

The latest date/time value on a platform shall be encoded in XML as '9999-12-31T11:59:59Z'

If a decoder encounters a *xs:dateTime* value that cannot be represented on the platform it should convert the value to either the earliest or latest date/time that can be represented on the platform. The XML decoder should not generate an error if it encounters an out of range date value.

The earliest date/time value on a platform is equivalent to a *Null* date/time value.

5.3.1.7 Guid

A *Guid* is encoded using the string representation defined in Section 5.1.3.

The XML schema for a *Guid* is:

```
<xs:complexType name="Guid">
  <xs:sequence>
    <xs:element name="String" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.8 ByteString

A *ByteString* value is encoded as an *xs:base64Binary* value.

The XML schema for a *ByteString* is:

```
<xs:element name="ByteString" type="xs:base64Binary" nillable="true"
/>
```

5.3.1.9 XmlElement

An *XmlElement* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="XmlElement">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="1" processContents="lax" />
  </xs:sequence>
</xs:complexType>
```

XmlElements may only be used inside *Variant* or *ExtensionObject* values.

5.3.1.10 NodeId

A *NodeId* value is encoded as a xs:string with the syntax:

```
ns=<namespaceindex>;<type>=<value>
```

The elements of the syntax are described in Table 19.

Table 19 – Components of NodeId

Field	Data Type	Description
<namespaceindex>	UInt16	The namespace index formatted as a base 10 number. If the index is 0 then the entire 'ns=0;' clause shall be omitted.
<type>	Enum	A flag that specifies the identifier type. The flag has the following values: i NUMERIC (UInteger) s STRING (String) g GUID (Guid) b OPAQUE (ByteString)
<value>	*	The identifier encoded as string. The identifier is formatted using the XML data type mapping for the identifier type. Note that the identifier may contain any non-null UTF8 character including whitespace.

Examples of *NodeIds:*

```
i=13
ns=10;i=-1
ns=10;s=Hello:World
g=09087e75-8e5e-499b-954f-f2a9603db28a
n=1;b=M/RbKBsRVkePCePcx24oRA==
```

The XML schema for a *NodeId* is:

```
<xs:complexType name="NodeId">
  <xs:sequence>
    <xs:element name="Identifier" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.11 ExpandedNodeId

An *ExpandedNodeId* value is encoded as a xs:string with the syntax:

```
svr=<serverindex>;ns=<namespaceindex>;<type>=<value>
or
svr=<serverindex>;nsu=<uri>;<type>=<value>
```

Table 20 – Components of ExpandedNodeId

Field	Data Type	Description
<serverindex>	UInt32	The server index formatted as a base 10 number. If the server index is 0 then the entire 'svr=0;' clause shall be omitted.
<namespaceindex>	UInt16	The namespace index formatted as a base 10 number. If the namespace index is 0 then the entire 'ns=0;' clause shall be omitted. The namespace index shall not be present if the URI is present.
<uri>	String	The namespace URI formatted as a string. Any reserved characters in the URI shall be replaced with a '%' followed by its 8 bit ANSI value encoded as two hexadecimal digits (case insensitive). For example, the character ';' would be replaced by '%3B'. The reserved characters are ';' and '%'. If the namespace URI is null or empty then 'nsu=;' clause shall be omitted.
<type>	Enum	A flag that specifies the identifier type. This field is described in Table 19.
<value>	*	The identifier encoded as string. This field is described in Table 19.

The XML schema for a *ExpandedNodeId* is:

```
<xs:complexType name="ExpandedNodeId">
  <xs:sequence>
    <xs:element name="Identifier" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.12 StatusCode

A *StatusCode* is formatted in an *xs:string* as an 8 digit hexadecimal number.

The XML schema for a *StatusCode* is:

```
<xs:complexType name="StatusCode">
  <xs:sequence>
    <xs:element name="Code" type="xs:unsignedInt" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.13 DiagnosticInfo

An *DiagnosticInfo* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="DiagnosticInfo">
  <xs:sequence>
    <xs:element name="SymbolicId" type="xs:int" minOccurs="0" />
    <xs:element name="NamespaceUri" type="xs:int" minOccurs="0" />
    <xs:element name="LocalizedText" type="xs:int" minOccurs="0/>
    <xs:element name="Locale" type="xs:int" minOccurs="0/>
    <xs:element  name="AdditionalInfo"  type="xs:string"  minOccurs="0"
/>
```

```
    <xs:element       name="InnerStatusCode"       type="tns:StatusCode"
minOccurs="0" />
    <xs:element name="InnerDiagnosticInfo" type="tns:DiagnosticInfo"
      minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.14 QualifiedName

A *QualifiedName* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="QualifiedName">
   <xs:sequence>
     <xs:element name="NamespaceIndex" type="xs:int" minOccurs="0" />
     <xs:element name="Name" type="xs:string" minOccurs="0" />
   </xs:sequence>
</xs:complexType>
```

5.3.1.15 LocalizedText

A *LocalizedText* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="LocalizedText">
  <xs:sequence>
    <xs:element name="Locale" type="xs:string" minOccurs="0" />
    <xs:element name="Text" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

5.3.1.16 ExtensionObject

An *ExtensionObject* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="ExtensionObject">
  <xs:sequence>
    <xs:element name="TypeId" type="tns:NodeId" minOccurs="0" />
    <xs:element name="Body" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

The body of the *ExtensionObject* contains a single element which is either a *ByteString* or XML encoded *Structure*. A decoder can distinguish between the two by inspecting the top level element. An element with the name tns:ByteString contains a OPC UA Binary encoded body. Any other name shall contain an OPC UA XML encoded body.

The *TypeId* is the *NodeId* for the *DataTypeEncoding Object*.

5.3.1.17 Variant

A *Variant* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="Variant">
  <xs:sequence>
    <xs:element name="Value" minOccurs="0" nillable="true">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

If the *Variant* represents a scalar value then it shall contain a single child element with the name of the built-in type. For example, the single precision floating point value 3.1415 would be encoded as:

```
<tns:Float>3.1415</tns:Float>
```

If the *Variant* represents a single dimensional array then it shall contain a single child element with the prefix 'ListOf' and the name built-in type. For example an array of strings would be encoded as:

```
<tns:ListOfString>
  <tns:String>Hello</tns:String>
  <tns:String>World</tns:String>
</tns:ListOfString>
```

If the Variant represents a Multidimensional array then it shall contain a child element with the name ‘Matrix’ with the two sub-elements shown in this example:

```
<tns:Matrix>
  <tns:Dimensions>
    <tns:Int32>2</tns:Int32>
    <tns:Int32>2</tns:Int32>
  </tns:Dimensions>
  <tns:Elements>
    <tns:String>A</tns:String>
    <tns:String>B</tns:String>
    <tns:String>C</tns:String>
    <tns:String>D</tns:String>
  </tns:Elements>
</tns:Matrix>
```

In this example, the array has the following elements:

```
[0,0] = "A"; [0,1] = "B"; [1,0] = "C"; [1,1] = "D"
```

The elements of a multi-dimensional array are always flattened into a single dimensional array where the higher rank dimensions are serialized first. This single dimensional array is encoded as a child of the ‘Elements’ element. The ‘Dimensions’ element is an array of Int32 values that specify the dimensions of the array starting with the lowest rank dimension. The multi-dimensional array can be reconstructed by using the dimensions encoded.

The complete set of built-in type names is found in Table 1.

5.3.1.18 DataValue

A *DataValue* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="DataValue">
  <xs:sequence>
    <xs:element    name="Value"    type="tns:Variant"    minOccurs="0"
nillable="true" />
    <xs:element  name="StatusCode"  type="tns:StatusCode"  minOccurs="0"
/>
    <xs:element        name="SourceTimestamp"        type="xs:dateTime"
minOccurs="0" />
    <xs:element    name="SourcePicoseconds"    type="xs:unsignedShort"
minOccurs="0"/>
    <xs:element        name="ServerTimestamp"        type="xs:dateTime"
minOccurs="0" />
    <xs:element    name="ServerPicoseconds"    type="xs:unsignedShort"
minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

5.3.2 Enumerations

Enumerations that are used as parameters in the Messages defined in Part 4 are encoded as *xs:string* with the following syntax:

```
<symbol>_<value>
```

The elements of the syntax are described in Table 21.

Table 21 – Components of Enumeration

Field	Type	Description
<symbol>	String	The symbolic name for the enumerated value.
<value>	UInt32	The numeric value associated with enumerated value.

For example, the XML schema for the *NodeClass* enumeration is:

```
<xs:simpleType name="NodeClass">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Unspecified_0" />
    <xs:enumeration value="Object_1" />
    <xs:enumeration value="Variable_2" />
    <xs:enumeration value="Method_4" />
    <xs:enumeration value="ObjectType_8" />
    <xs:enumeration value="VariableType_16" />
    <xs:enumeration value="ReferenceType_32" />
    <xs:enumeration value="DataType_64" />
    <xs:enumeration value="View_128" />
  </xs:restriction>
</xs:simpleType>
```

Enumerations that are stored in a Variant are encoded as an Int32 value.

For example, any *Variable* could have a value with a *DataType* of *NodeClass*. In this case the corresponding numeric value is placed in the Variant (e.g. *NodeClass*::*Object* would be stored as a 1).

5.3.3 Arrays

Arrays parameters are always encoded by wrapping the elements in a container element and inserting the container into the structure. The name of the container element should be the name of the parameter. The name of the element in the array shall be the type name.

For example, the *Read* service takes an array of *ReadValueIds.* The XML schema would look like:

```
<xs:complexType name="ListOfReadValueId">
  <xs:sequence>
    <xs:element name="ReadValueId" type="tns:ReadValueId"
       minOccurs="0" maxOccurs="unbounded" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

The nillable attribute shall be specified because XML encoders will drop elements in arrays if those elements are empty.

5.3.4 Structures

Structures are encoded as a *xs:complexType* with all of the fields appearing in a sequence. All fields are encoded as an *xs:element* and have *xs:maxOccurs* set to 1.

For example, the Read service has a *ReadValueId* structure in the request. The XML schema would look like:

```
<xs:complexType name="ReadValueId">
  <xs:sequence>
    <xs:element name="NodeId" type="tns:NodeId" minOccurs="1" />
    <xs:element name="AttributeId" type="xs:int" minOccurs="1" />
    <xs:element name="IndexRange" type="xs:string"
       minOccurs="0" nillable="true" />
    <xs:element name="DataEncoding" type="tns:NodeId" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

5.3.5 Messages

Messages are encoded as an *xs:complexType*. The parameters in each message are serialized in the same way the fields of a structure are serialized.

6 Security Protocols

6.1 Security Handshake

All *SecurityProtocols* shall implement the *OpenSecureChannel* and *CloseSecureChannel* services defined in Part 4. These services specify how to establish a *SecureChannel* and how to apply security to messages exchanged over that *SecureChannel*. The messages exchanged and the security algorithms applied to them are shown in Figure 10.

SecurityProtocols shall support three *SecurityModes*: None, Sign and SignAndEncrypt. If the SecurityMode is None then no security is used and the security handshake shown in Figure 10 is not required. However, a *SecurityProtocol* implementation shall still maintain a logical channel and provide a unique identifier for the *SecureChannel*.

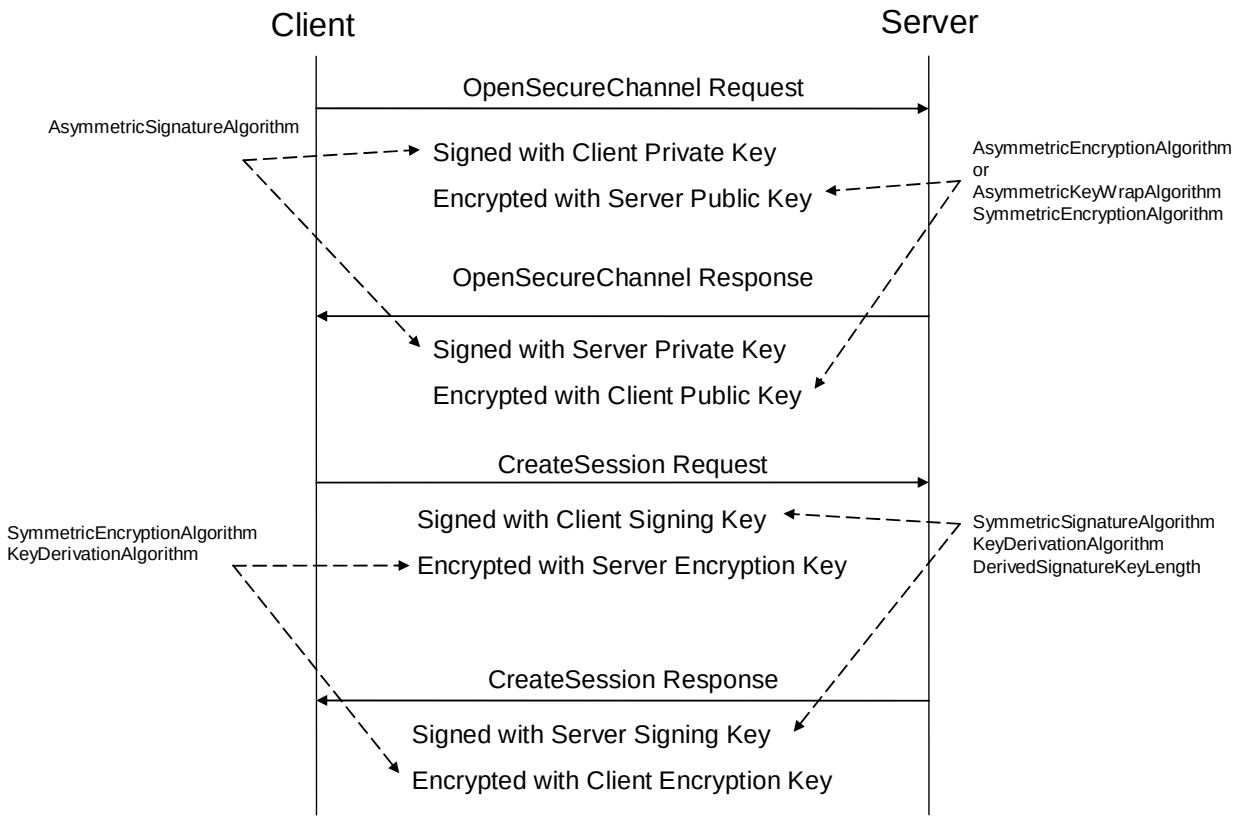

Figure 10 – The Security Handshake

Each *SecurityProtocol* mapping specifies exactly how to apply the security algorithms to the message. A set of security algorithms that shall be used together during a security handshake is called a *SecurityPolicy*. Part 7 defines standard *SecurityPolicies* as parts of the standard *Profiles* which OPC UA applications are expected to support. Part 7 also defines a URI for each standard *SecurityPolicy*.

A *Stack* is expected to have built in knowledge of the *SecurityPolicies* that it supports. Applications specify the *SecurityPolicy* they wish to use by passing the URI to the *Stack*.

Table 22 defines the contents of a *SecurityPolicy.* Each *SecurityProtocol* mapping specifies how to use each of the parameters in the *SecurityPolicy. A SecurityProtocol* mapping may not make use of all of the parameters.

Table 22 – SecurityPolicy

Name	Description
PolicyUri	The URI assigned to the SecurityPolicy.
SymmetricSignatureAlgorithm	The URI of the symmetric signature algorithm to use.
SymmetricEncryptionAlgorithm	The URI of the symmetric key encryption algorithm to use.
AsymmetricSignatureAlgorithm	The URI of the asymmetric signature algorithm to use.
AsymmetricKeyWrapAlgorithm	The URI of the asymmetric key wrap algorithm to use.
AsymmetricEncryptionAlgorithm	The URI of the asymmetric key encryption algorithm to use.
KeyDerivationAlgorithm	The key derivation algorithm to use.
DerivedSignatureKeyLength	The length in bits of the derived key used for message authentication.

The *AsymmetricEncryptionAlgorithm* is used when encrypting the entire message with an asymmetric key. Some *SecurityProtocols* do not encrypt the entire message with an asymmetric key. Instead, they use the *AsymmetricKeyWrapAlgorithm* to encrypt a symmetric key and then use the *SymmetricEncryptionAlgorithm* to encrypt the message.

The *AsymmetricSignatureAlgorithm* is used to sign a message with an asymmetric key.

The *KeyDerivationAlgorithm* is used to create the keys used to secure messages sent over the *SecureChannel*. The length of the keys used for encryption are implied by the *SymmetricEncryptionAlgorithm*. The length of the keys used for creating symmetric signatures depend on the *SymmetricSignatureAlgorithm* and may be different from the encryption key length.

6.2 Certificates

6.2.1 General

OPC UA *Applications* use *Certificates* to store the public keys needed for asymmetric cryptography operations. All *SecurityProtocols* use X509 Version 3 *Certificates* (see X509) encoded using the DER format (see X690). *Certificates* used by OPC UA *Applications* shall also conform to RFC 3280 which defines a profile for X509 *Certificates* when they are used as part of an Internet based application.

The *ServerCertificate* and *ClientCertificate* parameters used in the abstract *OpenSecureChannel* service are instances of the *ApplicationInstanceCertificate* data type. Clause 6.2.2 describes how to create an X509 certificate that can be used as an *ApplicationInstanceCertificate*.

The *ServerSoftwareCertificates* and *ClientSoftwareCertificates* parameters in the abstract *CreateSession* and *ActivateSession* services are instances of the *SignedSoftwareCertificate* data type. Clause 6.2.3 describes how to create an X509 *Certificate* that can be used as an *SignedSoftwareCertificate*.

6.2.2 Application Instance Certificate

An *ApplicationInstanceCertificate* is a *ByteString* containing the DER encoded form of an X509v3 *Certificate*. This *Certificate* is issued by certifying authority and identifies an instance of an application running on a single host. The X509v3 fields contained in an *ApplicationInstance Certificate* are described in Table 23. The fields are defined completely in RFC 3280.

Table 23 – ApplicationInstanceCertificate

Name	Abstract Parameter	Description
ApplicationInstanceCertificate		An X509v3 Certificate.
version	version	shall be “V3”
serialNumber	serialNumber	The serial number assigned by the issuer.
signatureAlgorithm	signatureAlgorithm	The algorithm used to sign the Certificate.
signature	signature	The signature created by the Issuer.
issuer	issuer	The distinguished name of the Certificate used to create the signature. The *issuer* field is completely described in RFC 3280.
validity	validTo, validFrom	When the *Certificate* becomes valid and when it expires.
subject	subject	The distinguished name of the application instance. The Common Name attribute shall be specified and should be the *productName* or a suitable equivalent. The Organization Name attribute shall be the name of the Organization that executes the application instance. This organization is usually not the vendor of the application. Other attributes may be specified. The *subject* field is completely described in RFC 3280.
subjectAltName	applicationUri, hostnames	The alternate names for the application instance. Shall include a uniformResourceIdentifier which is equal to the *applicationUri*. *Servers* shall specify a dNSName *or* IPAddress which identifies the machine where the application instance runs. Additional dNSNames may be specified if the machine has multiple names. The IPAddress should not be specified if the *Server* has dNSName. The subjectAltName field is completely described in RFC 3280.
publicKey	publicKey	The public key associated with the *Certificate*.
keyUsage	keyUsage	Specifies how the certificate key may be used. Shall include digitalSignature, nonRepudiation, keyEncipherment and dataEncipherment. Other key uses are allowed.
extendedKeyUsage	keyUsage	Specifies additional key uses for the *Certificate*. Shall specify 'serverAuth and/or clientAuth. Other key uses are allowed.

6.2.3 Signed Software Certificate

A *SignedSoftwareCertificate* is a *ByteString* containing the DER encoded form of an X509v3 *Certificate*. This *Certificate* is issued by a certifying authority and contains an X509v3 extension with the *SoftwareCertificate* which specifies the claims verified by the certifying authority. The X509v3 fields contained in a *SignedSoftwareCertificate* are described in Table 24. The fields are defined completely in RFC 3280.

Table 24 – SignedSoftwareCertificate

Name		Description
SignedSoftwareCertificate		An X509v3 Certificate.
version	version	Shall be “V3”
serialNumber	serialNumber	The serial number assigned by the issuer.
signatureAlgorithm	signatureAlgorithm	The algorithm used to sign the Certificate.
signature	signature	The signature created by the Issuer.
issuer	issuer	The distinguished name of the Certificate used to create the signature. The *issuer* field is completely described in RFC 3280.
validity	validTo, validFrom	When the *Certificate* becomes valid and when it expires.
subject	subject	The distinguished name of the product. The Common Name attribute shall be the same as the *productName* in the *SoftwareCertificate* and the Organization Name attribute shall the *vendorName* in the *SoftwareCertificate*. Other attributes may be specified. The *subject* field is completely described in RFC 3280.
subjectAltName	productUri	The alternate names for the product. shall include a ‘uniformResourceIdentifier’ which is equal to the *productUri* specified in the *SoftwareCertificate*. The subjectAltName field is completely described in RFC 3280.
publicKey	publicKey	The public key associated with the *Certificate*.
keyUsage	keyUsage	Specifies how the certificate key may be used. shall be ‘digitalSignature’ and ‘nonRepudiation’ Other key uses are not allowed.
extendedKeyUsage	keyUsage	Specifies additional key uses for the Certificate. May specify ‘codeSigning’. Other key usages are not allowed.
softwareCertificate	softwareCertificate	The XML encoded form of the *SoftwareCertificate* stored as UTF8 text. Clause 5.3.4 describes how to encode a *SoftwareCertificate in XML.* The ASN.1 Object Identifier (OID) for this extension is: 1.2.840.113556.1.8000.2264.1.6.1

6.3 WS Secure Conversation

6.3.1 Overview

Any message sent via SOAP may be secured with the WS Secure Conversation protocol. This protocol specifies a way to negotiate shared secrets via WS Trust and then use these secrets to secure messages exchanged with the mechanisms defined in WS Security.

The mechanisms for actually signing XML elements are described in the XML Signature specification. The mechanisms for encrypting XML elements are described the XML Encryption specification.

WS Security Policy defines standard algorithm suites which can be used to secure SOAP messages. These algorithm suites map directly onto the *SecurityPolicies* that are defined in Part 7. WS-I Basic Security Profile 1.1 defines best practices when using WS-Security which will help ensure interoperability. All OPC UA implementations shall conform to this specification.

The Timestamp header defined by WS Security is used to prevent replay attacks and shall be present and signed in all messages exchanged.

Figure 11 illustrates the relationship between the different WS-* specifications that are used by this mapping. The versions of the WS-* specifications shown in the diagram were the most

current versions at the time of publication. Part 7 may define *Profiles* that require support for future versions of these specifications.

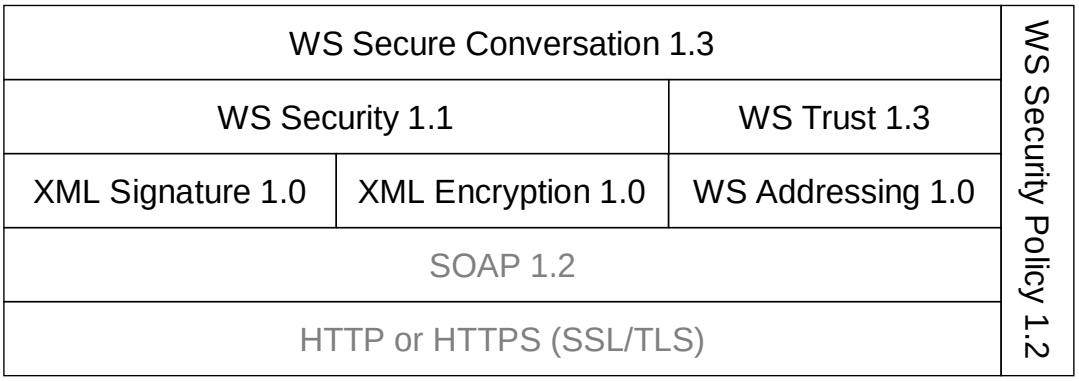

Figure 11 – The XML Web Services Stack

Figure 12 illustrates how these WS-* specifications are used in the security handshake.

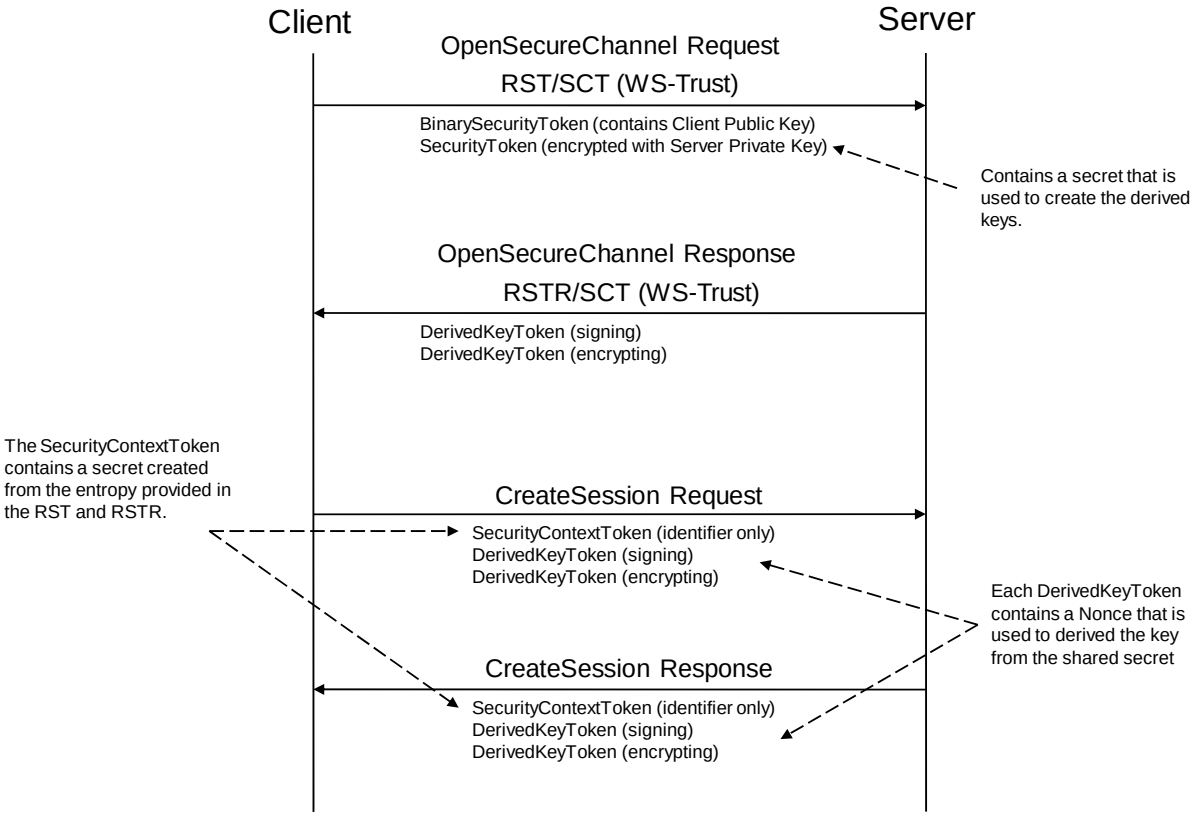

Figure 12 – The WS Secure Conversation Handshake

The RST (Request Security Token) and RSTR (Request Security Token Response) messages are defined by WS Trust. WS Secure Conversation defines new actions for these messages that tell the server that the client wants to create a SCT (Security Context Token). The SCT contains the shared keys that the applications use to secure messages sent over the *SecureChannel*.

Individual messages are secured with keys derived from the SCT using the mechanism defined in WS Secure Conversation. The sections below specify the structure of the individual messages and illustrate which features from the WS-* specifications are required to implement the OPC UA security handshake.

6.3.2 Notation

SOAP messages use XML elements defined in a number of different specifications. This document uses the prefixes in Table 25 to identify the specification that defines an XML element.

Table 25 – WS-* Namespace Prefixes

Prefix	Specification
wsu	WS-Security Utilities
wsse	WS-Security Extensions
wst	WS-Trust
wsc	WS-Secure Conversation
wsa	WS-Addressing
xenc	XML Encryption

6.3.3 Request Security Token (RST/SCT)

The Request Security Token message implements the abstract OpenSecureChannel request message defined in Part 4. The syntax of this message is defined by WS Trust. The structure of the message is described in detail in WS Secure Conversation.

This message shall have the following tokens:

1) A wsse:BinarySecurityToken containing the Client's Public Key. The public key is sent in a DER encoded X509v3 certificate.

2) An encrypted wsse:SecurityToken containing ClientNonce used to derive keys. This token shall be encrypted with the *AsymmetricKeyWrapAlgorithm* and the public key associated with the Server's Application Instance Certificate.

3) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm*. The signature element shall then be signed using the *AsymmetricSignatureAlgorithm* with the Client's Private Key. The wsc:DerivedKeyToken shall also specify a Nonce.

4) A wsc:DerivedKeyToken which is used to encrypt the body of the message using the *SymmetricEncryptionAlgorithm*.

This message shall have the wsa:Action, wsa:MessageId, wsa:ReplyTo and wsa:To headers defined by WS Addressing. The message shall also have a wsu:Timestamp header defined by WS Security. These headers shall also be signed with the derived key used to sign the message body.

The signature shall be calculated before applying encryption and the signature shall be encrypted.

The mapping between the OpenSecureChannel request parameters and the elements of the RST/SCT message are shown in Table 26.

Table 26 – RST/SCT Mapping to an OpenSecureChannel Request

OpenSecureChannel Parameter	RST/SCT Element	Description
clientCertificate	wsse:BinarySecurityToken	Passed in the SOAP header.
requestType	wst:RequestType	Shall be “http://schemas.xmlsoap.org/ws/2005/02/trust/Issue” when creating a new SCT. Shall be “http://schemas.xmlsoap.org/ws/2005/02/trust/Renew” when renewing a SCT.
secureChannelId	wsse:SecurityTokenReference	Passed in the SOAP header when renewing an SCT.
securityMode securityPolicyUri	wst:SignatureAlgorithm wst:EncryptionAlgorithm wst:KeySize	These elements describe the *SecurityPolicy* requested by the client. These elements shall match the *SecurityPolicy* used by the endpoint that the client wishes to connect to. These elements are optional.
clientNonce	wst:Entropy	This contains the nonce specified by the client. The nonce is specified with the wst:BinarySecret element.
requestedLifetime	wst:Lifetime	The requested lifetime for the SCT. This element is optional.

6.3.4 Request Security Token Response (RSTR/SCT)

The Request Security Token Response message implements the abstract OpenSecureChannel response message defined in Part 4. The syntax of this message is defined by WS Trust. The use of the message is described in detail in WS Secure Conversation. This message not signed or encrypted with the asymmetric algorithms as described in the Part 4. The symmetric algorithms and a key provided in the request message are used instead.

This message shall have the following tokens:

1) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm.* This key is derived from the encrypted *SecurityToken* specified in the RST/SCT message. The wsc:DerivedKeyToken shall also specify a Nonce.

2) A wsc:DerivedKeyToken which is used to encrypt the body of the message using the SymmetricEncryptionAlgorithm. This key is derived from the encrypted *SecurityToken* specified in the RST/SCT message. The wsc:DerivedKeyToken shall also specify a Nonce.

This message shall have the wsa:Action and wsa:RelatesTo headers defined by WS Addressing. The message shall also have a wsu:Timestamp header defined by WS Security. These headers shall also be signed with the derived key used to sign the message body.

The signature shall be calculated before applying encryption and the signature shall be encrypted.

The mapping between the *OpenSecureChannel* response parameters and the elements of the RSTR/SCT message are shown Table 27.

Table 27 – RSTR/SCT Mapping to an OpenSecureChannel Response

OpenSecureChannel Parameter	RSTR/SCT Element	Description
---	wst:RequestedProofToken	This contains a wst:ComputedKey element which specifies the algorithm used to compute the shared secret key from the nonces provided by the client and the server.
---	wst:TokenType	Specifies the type of token issued.
securityToken	wst:RequestedSecurityToken	Specifies the new SCT (Security Context Token) or renewed SCT.
channelId	wsc:Identifier	An absolute URI which identifies the SCT.
tokenId	wsc:Instance	An identifier for a set of keys issued for a context. It shall be unique within the context.
createdAt	wsu:Created	This is optional element in the wsc:SecurityContextToken returned in the header.
revisedLifetime	wst:Lifetime	The revised lifetime for the SCT.
serverNonce	wst:Entropy	This contains the nonce specified by the server. The nonce is specified with the wst:BinarySecret element. The xenc:EncryptedData element is not used in OPC UA because the message body shall be encrypted.

The lifetime specifies the UTC expiration time for the security context token. The client shall renew the SCT before that time by sending the RST/SCT message again. The exact behaviour is described in Part 4 (Clause 5.4).

6.3.5 Using the SCT

Once the Client receives the RSTR/SCT message it can use the SCT to secure all other messages.

An identifier for the SCT used shall be passed as an wsc:SecurityContextToken in each request message. The response message shall reference the *SecurityContextToken* used in the request.

If encryption is used it shall be applied before the signature is calculated.

Any message secured with the *SecurityContextToken* shall have the following additional tokens:

1) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm*. This key is derived from the *SecurityContextToken*. The wsc:DerivedKeyToken shall also specify a Nonce.

2) A wsc:DerivedKeyToken which is used to encrypt the body of the message using the *SymmetricEncryptionAlgorithm*. This key is derived from the *SecurityContextToken*. The wsc:DerivedKeyToken shall also specify a Nonce.

This message shall have the wsa:Action and wsa:RelatesTo headers defined by WS Addressing. The message shall also have a wsu:Timestamp header defined by WS Security.

6.3.6 Cancelling Security Contexts

The Cancel message defined by WS Trust implements the abstract CloseSecureChannel request message defined in Part 4.

This message shall be secured with the SCT.

6.4 OPC UA Secure Conversation

6.4.1 Overview

OPC UA Secure Conversation (UASC) is a binary version of WS-Secure Conversation. It allows secure communication over transports that do not use SOAP or XML.

UASC is designed to operate with different *TransportProtocols* that may have limited buffer sizes. For this reason, OPC UA Secure Conversation will break OPC UA messages into several pieces (called '*MessageChunks*') that are smaller than the buffer size allowed by the *TransportProtocol*. UASC requires a *TransportProtocol* buffer size that is at least 8192 bytes.

All security is applied to individual *MessageChunks* and not the entire OPC UA message. A *Stack* that implements UASC is responsible for verifying the security on each *MessageChunk* received and reconstructing the original OPC UA message.

All *MessageChunks* will have a 4-byte sequence assigned to them. These sequence numbers are used to detect and prevent replay attacks.

UASC requires a *TransportProtocol* that will preserve the order of *MessageChunks*, however, a UASC implementation does not necessarily process the *Messages* in the order that they were received.

6.4.2 MessageChunk Structure

Figure 13 shows the structure of a *MessageChunk* and how security is applied to the message.

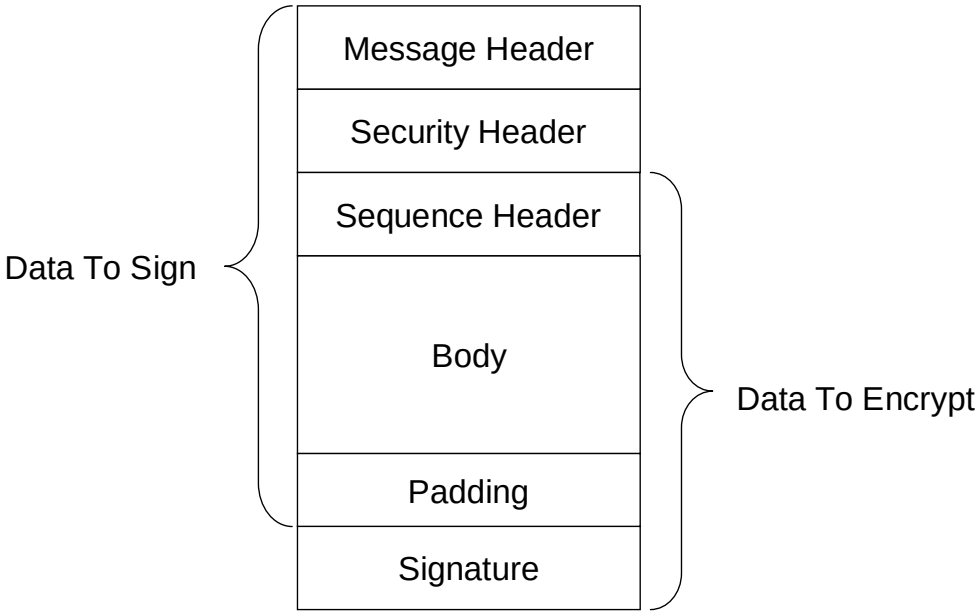

Figure 13 – OPC UA Secure Conversation MessageChunk

Every *MessageChunk* has a message header with the fields defined in Table 28.

Table 28 – OPC UA Secure Conversation Message Header

Name	Data Type	Description
MessageType	Byte[3]	A three byte ASCII code that identifies the message type. The following values are defined at this time: MSG A message secured with the keys associated with a channel. OPN OpenSecureChannel message. CLO CloseSecureChannel message.
IsFinal	Byte	A one byte ASCII code that indicates whether the MessageChunk is the final chunk in a message. The following values are defined at this time: C An intermediate chunk. F The final chunk. A The final chunk (used when an error occurred and the message is aborted).
MessageSize	UInt32	The length of the MessageChunk, in bytes. This value includes size of the message header.
SecureChannelId	UInt32	A unique identifier for the SecureChannel assigned by the server. If a Server receives a SecureChannelId which it does not recognize it shall return an appropriate transport layer error.

The message header is followed by a security header which specifies what cryptography operations have been applied to the message. There are two versions of the security header which depend on the type of security applied to the Message. The security header used for asymmetric algorithms is defined in Table 29. Asymmetric algorithms are used to secure the *OpenSecureChannel* messages. PKCS #1 defines a set asymmetric algorithms that may be used by UASC implementations. The AsymmetricKeyWrapAlgorithm element of the *SecurityPolicy* structure defined in Table 22 is not used by UASC implementations.

Table 29 – Asymmetric Algorithm Security Header

Name	Data Type	Description
SecurityPolicyUriLength	Int32	The length of the SecurityPolicyUri in bytes.This value shall not exceed 255 bytes.
SecurityPolicyUri	Byte[*]	The URI of the security policy used to secure the message. This field is encoded as a UTF8 string without a null terminator.
SenderCertificateLength	Int32	The length of the SenderCertificate in bytes. This value shall not exceed MaxCertificateSize bytes.
SenderCertificate	Byte[*]	The X509v3 certificate assigned to the sending application instance. This is a DER encoded blob. The structure of an X509 certificate is defined in X509. The DER format for a certificate is defined in X690 This indicates what private key was used to sign the MessageChunk. The Stack shall close the channel and report an error to the application if the *SenderCertificate* is too large for the buffer size supported by the transport layer. This field shall be null if the message is not signed.
ReceiverCertificateThumbprintLength	Int32	The length of the ReceiverCertificateThumbprint in bytes. The length of this field is always 20 bytes.
ReceiverCertificateThumbprint	Byte[*]	The thumbprint of the X509v3 certificate assigned to the receiving application instance. The thumbprint is the SHA1 digest of the DER encoded form of the certificate. This indicates what public key was used to encrypt the MessageChunk. This field shall be null if the message is not encrypted.

The receiver shall close the communication channel if any of the fields in the security header have invalid lengths.

The *SenderCertificate* shall be small enough to fit into a single *MessageChunk* and leave room for at least one byte of body information. The maximum size for the *SenderCertificate* can be calculated with this formula:

```
MaxCertificateSize =
   MessageChunkSize –
   12 -                        // Header size
   4 -                         // SecurityPolicyUriLength
   SecurityPolicyUri -         // UTF-8 encoded string
   4 -                         // SenderCertificateLength
   4 -                         //
ReceiverCertificateThumbprintLength
   20 -                        // ReceiverCertificateThumbprint
   8 -                         // SequenceHeader size
   1 -                         // Minimum body size
   1 -                         // PaddingSize if present
   Padding -                   // Padding if present
   AsymmetricSignatureSize     // If present
```

The *MessageChunkSize* depends on the transport protocol but shall be at least 8196 bytes, The *AsymmetricSignatureSize* depends on the number of bits in the public key for the *SenderCertificate*. The *Int32FieldLength* is the length of an encoded Int32 value and it is always 4 bytes.

The security header used for symmetric algorithms defined in Table 30. Symmetric algorithms are used to secure all messages other than the *OpenSecureChannel* messages. FIPS 197 define symmetric encryption algorithms that UASC implementations may use. FIPS 180-2 and HMAC define some symmetric signature algorithms.

Table 30 – Symmetric Algorithm Security Header

Name	Data Type	Description
TokenId	UInt32	A unique identifier for the *SecureChannel* token used to secure the message. This identifier is returned by the server in an OpenSecureChannel response message. If a Server receives a TokenId which it does not recognize it shall return an appropriate transport layer error.

The security header is always followed by the sequence header which is defined in Table 31. The sequence header ensures that the first encrypted block of every message sent over a channel will start with different data.

Table 31 – Sequence Header

Name	Data Type	Description
SequenceNumber	UInt32	A monotonically increasing sequence number assigned by the sender to each MessageChunk sent over the SecureChannel.
RequestId	UInt32	An identifier assigned by the client to OPC UA request Message. All *MessageChunks* for the request and the associated response use the same identifier.

SequenceNumbers may not be reused for any *TokenId*. The token lifetime should be short enough to ensure that this never happens, however, if it does the receiver should treat it as a transport error and force a reconnect.

The *SequenceNumber* shall also monotonically increase for all messages and shall not wrap around until it is greater than 4294966271 (UInt32.MaxValue – 1024). The first number after the wrap around shall be less than 1024. Note that this requirement means that *SequenceNumbers* do not reset when a new *TokenId* is issued. The *SequenceNumber* shall be incremented by exactly one for each *MessageChunk* sent unless the communication channel was interrupted and re-established. Gaps are permitted between the *SequenceNumber* for the last *MessageChunk* received before the interruption and the

SequenceNumber for first *MessageChunk* received after communication was re-established. Note that the first *MessageChunk* after a network interruption is always an *OpenSecureChannel* request or response.

The sequence header is followed by the message body which is encoded with the OPC UA Binary encoding as described in Section 5.2.6. The body may be split across multiple *MessageChunks*.

Each *MessageChunk* also has a footer with the fields defined in Table 32.

Table 32 – OPC UA Secure Conversation Message Footer

Name	Data Type	Description
PaddingSize	Byte	The number of padding bytes (not including the byte for the PaddingSize).
Padding	Byte[*]	Padding added to the end of the message to ensure length of the data to encrypt is an integer multiple of the encryption block size. The value of each byte of the padding is equal to PaddingSize.
Signature	Byte[*]	The signature for the *MessageChunk*. The signature includes the all headers, all message data, the PaddingSize and the Padding.

The formula to calculate the amount of padding depends on the amount of data that needs to be sent (called BytesToWrite). The sender shall first calculate the maximum amount of space available in the *MessageChunk* (called MaxBodySize) using the following formula:

```
MaxBodySize = PlainTextBlockSize * Floor((MessageChunkSize –
      HeaderSize – SignatureSize - 1)/CipherTextBlockSize) –
            SequenceHeaderSize;
```

Where the *HeaderSize* includes the *MessageHeader* and the *SecurityHeader*. The *SequenceHeaderSize* is always 8 bytes.

During encryption a block with a size equal to *PlainTextBlockSize* is processed to produce a block with size equal to *CipherTextBlockSize*. These values depend on the encryption algorithm and may be the same.

The UA message can fit into a single chunk if *BytesToWrite* is less than or equal to the *MaxBodySize*. In this case the *PaddingSize* is calculated with this formula:

```
PaddingSize = PlainTextBlockSize –
      ((BytesToWrite + SignatureSize + 1) % PlainTextBlockSize);
```

If the *BytesToWrite* is greater than *MaxBodySize* the sender shall write *MaxBodySize* bytes with a PaddingSize of 0. The remaining *BytesToWrite* – *MaxBodySize* bytes shall be sent in subsequent *MessageChunks.*

The *PaddingSize* and *Padding* fields are not present if the *MessageChunk* is not encrypted.

The Signature field is not present if the *MessageChunk* is not signed.

6.4.3 MessageChunks and Error Handling

Message chunks are sent as they are encoded. Message chunks belonging to the same Message shall be sent sequentially. If an error occurs creating a chunk then the sender shall send a final chunk to the receiver that tells the receiver that an error occurred and that it should discard the previous chunks. The sender indicates that the chunk contains an error by setting the IsFinal flag to ‘A’ (for Abort). Table 33 specifies the contents of the message abort chunk.

Table 33 – OPC UA Secure Conversation Message Abort Body

Name	Data Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 40.
Reason	String	A more verbose description of the error. This string shall not be more than 4096 characters. A client shall ignore strings that are longer than this.

The receiver shall check the security on the abort chunk before processing it. If everything is ok then the receiver shall ignore the message but shall not close the *SecureChannel*. The client shall report the error back to the application as *StatusCode* for the request. If the client is the sender then it shall report the error without waiting for a response from the server.

6.4.4 Establishing a SecureChannel

Most messages require a *SecureChannel* to be established. A client does this by sending an *OpenSecureChannel* request to the server. The server shall validate the message and the *ClientCertificate* and return an *OpenSecureChannel* response. Some of the parameters defined for the *OpenSecureChannel* service are specified in the security header (see Clause 6.4.2) instead of the body of the message. For this reason, the *OpenSecureChannel* service is not the same as the one specified in the Part 4. Table 34 lists the parameters that appear in the body of the message.

Table 34 – OPC UA Secure Conversation OpenSecureChannel Service

Name	Data Type
Request	
RequestHeader	RequestHeader
ClientProtocolVersion	UInt32
RequestType	SecurityTokenRequestType
SecurityMode	MessageSecurityMode
ClientNonce	ByteString
RequestedLifetime	Int32
Response	
ResponseHeader	ResponseHeader
ServerProtocolVersion	UInt32
SecurityToken	ChannelSecurityToken
SecureChannelId	UInt32
TokenId	UInt32
CreatedAt	DateTime
RevisedLifetime	Int32
ServerNonce	ByteString

The *ClientProtocolVersion* and *ServerProtocolVersion* parameters are not defined in Part 4 and are added to the message to allow backward compatibility if the OPC UA-SecureConversation protocol needs to be updated in the future. Receivers always accept numbers greater than the latest version that they support. The receiver with the higher version number is expected to ensure backward compatibility.

If the OPC UA-SecureConversation protocol is used with the OPC UA-TCP protocol (see Clause 7.1) then the version numbers specified in the *OpenSecureChannel* messages shall be the same as the version numbers specifed in the OPC UA-TCP protocol *Hello/Acknowledge* messages. The receiver shall close the channel and report a *Bad_ProtocolVersionUnsupported* error if there is a mismatch.

The server shall return an error response as described in Clause 5.3 of Part 4 if there are any errors with the parameters specified by the client.

The *RevisedLifetime* tells the client when it shall renew the token by sending another *OpenSecureChannel* request. The client shall continue to accept the old token until it receives

the *OpenSecureChannel* response. The server has to accept requests secured with the old token until that token expires or until it receives a message from the Client secured with the new token. The Server shall reject renew requests if the *SenderCertificate* is not the same as the one used to create the *SecureChannel* or if there is a problem decrypting or verifying the signature. The Client shall abandon the *SecureChannel* if the *Certificate* used to sign the response is not the same as the *Certificate* used to encrypt the request.

The *OpenSecureChannel* messages are not signed or encrypted if the *SecurityMode* is *None*. The nonces are ignored and should be set to null. The *SecureChannelId* and the *TokenId* are still assigned but no security is applied to messages exchanged via the channel. The token shall still be renewed before the *RevisedLifetime* expires. Receivers shall still ignore invalid or expired TokenIds.

If the communication channel breaks the Server shall maintain the secure channel long enough to allow the client to reconnect. The *ReviseLifetime* parameter also tells the client how long the Server will wait. If the Client cannot reconnect within that period it shall assume the SecureChannel has been closed.

The *AuthenticationToken* in the *RequestHeader* shall be set to null.

If an error occurs after the Server has verified message security if shall returned a *ServiceFault* instead of a *OpenSecureChannel* response. The *ServiceFault* message is described in Part 4 Section 7.24.

If the *SecurityMode* is not *None* then the Server shall verify that a *SenderCertificate* and a *ReceiverCertificateThumbprint* were specified in the *SecurityHeader*.

6.4.5 Deriving Keys

Once the *SecureChannel* is established the messages are signed and encrypted with keys derived from the nonces exchanged in the *OpenSecureChannel* call. These keys are derived by passing the nonces to a pseudo-random function which produces a sequence of bytes from a set of inputs. A pseudo-random function is represented by the following function declaration:

```
Byte[] PRF(Byte[] secret, Byte[] seed, Int32 length, Int32
offset)
```

Where *length* is the number of bytes to return and *offset* is a number of bytes from the beginning of the sequence.

The lengths of the keys that need to be generated depend on the *SecurityPolicy* used for the channel. The following information is specified by the *SecurityPolicy*:

a) SigningKeyLength (from the DerivedSignatureKeyLength);

b) EncryptingKeyLength (implied by the SymmetricEncryptionAlgorithm);

c) EncryptingBlockSize (implied by the SymmetricEncryptionAlgorithm);

The parameters passed to the pseudo random function are specified in Table 35.

Table 35 – Cryptography Key Generation Parameters

Key	Secret	Seed	Length	Offset
ClientSigningKey	ServerNonce	ClientNonce	SigningKeyLength	0
ClientEncryptingKey	ServerNonce	ClientNonce	EncryptingKeyLength	SigningKeyLength
ClientInitializationVector	ServerNonce	ClientNonce	EncryptingBlockSize	SigningKeyLength+ EncryptingKeyLength
ServerSigningKey	ClientNonce	ServerNonce	SigningKeyLength	0
ServerEncryptingKey	ClientNonce	ServerNonce	EncryptingKeyLength	SigningKeyLength
ServerInitializationVector	ClientNonce	ServerNonce	EncryptingBlockSize	SigningKeyLength+ EncryptingKeyLength

The client keys are used to secure messages sent by the client. The server keys are used to secure messages sent by the server.

The SSL/TLS specification defines a pseudo random function called P_SHA1 which is used for some *SecurityProfiles*. The P_SHA1 algorithm is defined as follows:

```
P_SHA1(secret, seed) = HMAC_SHA1(secret, A(1) + seed) +
                       HMAC_SHA1(secret, A(2) + seed) +
                       HMAC_SHA1 (secret, A(3) + seed) + ...
Where A(n) is defined as:
  A(0) = seed
  A(n) = HMAC_SHA1(secret, A(n-1))
+ indicates that the results are appended to previous results.
```

6.4.6 Verifying Message Security

The contents of the *MessageChunk* shall not be interpreted until the message is decrypted and the signature and sequence number verified.

If error occurs during message verification the receiver shall close the communication channel. If the receiver is the Server it shall also send a transport error message before closing the channel. Once the channel is closed the Client shall attempt to re-open the channel and request a new token by sending an *OpenSecureChannel* request. The mechanism for sending transport errors to the Client depends on the communication channel.

The receiver shall first check the *SecureChannelId*. This value may be 0 if the message is an *OpenSecureChannel* request. For other messages it shall report a Bad_SecureChannelUnknown error if the *SecureChannelId* is not recognized. If the message is an *OpenSecureChannel* request and the *SecureChannelId* is not 0 then the *SenderCertificate* shall be the same as the *SenderCertificate* used to create the channel.

If the message is secured with asymmetric algorithms then the receiver shall verify that it supports the requested *SecurityPolicy*. If the message is the response sent to the Client then the *SecurityPolicy* shall be the same as the one specified in the request. In the Server the *SecurityPolicy* shall be the same as the one used to originally create the *SecureChannel*. The receiver shall then verify the *SenderCertificate* using the rules defined in Part 4 Section 6.1.4. The receiver shall report the appropriate error if *Certificate* validation fails. The receiver shall verify the *ReceiverCertificateThumbprint* and report a Bad_CertificateUnknown error if it does not recognize it.

If the message is secured with symmetric algorithms then a Bad_SecureChannelTokenUnknown error shall be reported if the *TokenId* refers to a token that has expired or is not recognized.

If decryption or signature validation fails then a Bad_SecurityChecksFailed error is reported. If an implementation allows multiple *SecurityModes* to be used the receiver shall also verify that the message was secured properly as required by the *SecurityMode* specified in the *OpenSecureChannel* request.

After the security validation is complete the receiver shall verify the *RequestId* and the *SequenceNumber*. If these checks fail a Bad_SecurityChecksFailed error is reported. The *RequestId* only needs to be verified by the *Client* since only the *Client* knows if it is valid or not.

At this point the *SecureChannel* knows it is dealing with an authenticated message that was not tampered with or resent. This means the *SecureChannel* can return a secured error response if any further problems are encountered.

Stacks that implement UASC shall have a mechanism to log errors when invalid messages are discarded. This mechanism is intended for developers, systems integrators and administrators to debug network system configuration issues and to detect attacks on the network.

7 Transport Protocols

7.1 OPC UA TCP

7.1.1 Overview

OPC UA TCP is a simple TCP based protocol that establishes a full duplex channel between a client and server. This protocol has two key features that differentiate it from HTTP. First, this protocol allows responses to be returned in any order. Second, this protocol allows responses

to be returned on a different socket if communication failures causes temporary socket interruption.

The OPC UA TCP protocol is designed to work with the *SecureChannel* implemented by a layer higher in the stack. For this reason, the OPC UA TCP protocol defines its interactions with the *SecureChannel* in addition to the wire protocol.

7.1.2 Message Structure

Every OPC UA TCP message has a header with the fields defined in Table 36.

Table 36 – OPC UA TCP Message Header

Name	Type	Description
MessageType	Byte[3]	A three byte ASCII code that identifies the message type. The following values are defined at this time: HEL a Hello message. ACK an Acknowledge message. ERR an Error message. The *SecureChannel* layer defines additional values which the OPC UA TCP layer shall accept.
Reserved	Byte[1]	Ignored. shall be set to the ASCII codes for 'F' if the MessageType is one of the values supported by the OPC UA TCP protocol.
MessageSize	UInt32	The length of the message, in bytes. This value includes the 8 bytes for the message header.

The layout of the OPC UA TCP message header is intentionally identical to the first 8 bytes of the OPC UA Secure Conversation message header defined in Table 28. This allows the OPC UA TCP layer to extract the *SecureChannel* messages from the incoming stream even if it does not understand their contents.

The OPC UA TCP layer shall verify the *MessageType* and make sure the *MessageSize* is less than the negotiated ReceiveBufferSize before passing any message onto the *SecureChannel* layer.

The Hello message has the additional fields shown in Table 37.

Table 37 – OPC UA TCP Hello Message

Name	Data Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the *Client*. The *Server* may reject the *Client* by returning *Bad_ProtocolVersionUnsupported*. If the *Server* accepts the connection is responsible for ensuring that it returns messages that conform to this version of the protocol. The *Server* shall always accept versions greater than what it supports.
ReceiveBufferSize	UInt32	The largest message that the sender can receive. This value shall be greater than 8192 bytes.
SendBufferSize	UInt32	The largest message that the sender will send. This value shall be greater than 8192 bytes.
MaxMessageSize	UInt32	The maximum size for any response message. The server shall abort the message with a *Bad_ResponseTooLarge StatusCode* if a response message exceeds this value. The mechanism for aborting messages is described fully in Clause 6.4.3. The message size is calculated using the unencrypted message body. A value of zero indicates that the Client has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any response message. The server shall abort the message with a *Bad_ResponseTooLarge StatusCode* if a response message exceeds this value. The mechanism for aborting messages is described fully in Clause 6.4.3. A value of zero indicates that the Client has no limit.
EndpointUrl	String	The URL of the Endpoint which the Client wished to connect to. The encoded value shall be less than 4096 bytes. Servers shall return a Bad_TcpUrlRejected error and close the connection if the length exceeds 4096 or if it does not recognize the resource identified by the URL.

The *EndpointUrl* parameter is used to allow multiple servers to share the same port on a machine. The process listening (a.k.a. proxy) on the port would connect to the Server identified by the *EndpointUrl* and would forward all messages to the Server via this socket. If one socket closes then the proxy shall close the other socket.

The Acknowledge message has the additional fields shown in Table 38.

Table 38 – OPC UA TCP Acknowledge Message

Name	Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the *Server*. If the *Client* accepts the connection is responsible for ensuring that it sends messages that conform to this version of the protocol. The *Client* shall always accept versions greater than what it supports.
ReceiveBufferSize	UInt32	The largest message that the sender can receive. This value shall not be larger than what the Client requested in the Hello message. This value shall be greater than 8192 bytes.
SendBufferSize	UInt32	The largest message that the sender will send. This value shall not be larger than what the Client requested in the Hello message. This value shall be greater than 8192 bytes.
MaxMessageSize	UInt32	The maximum size for any request message. The client shall abort the message with a *Bad_RequestTooLarge StatusCode* if a request message exceeds this value. The mechanism for aborting messages is described fully in Clause 6.4.3. The message size is calculated using the unencrypted message body. A value of zero indicates that the *Server* has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any request message. The client shall abort the message with a *Bad_RequestTooLarge StatusCode* if a request message exceeds this value. The mechanism for aborting messages is described fully in Clause 6.4.3. A value of zero indicates that the *Server* has no limit.

The Error message has the additional fields shown in Table 39.

Table 39 – OPC UA TCP Error Message

Name	Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 40.
Reason	String	A more verbose description of the error. This string shall not be more than 4096 characters. A client shall ignore strings that are longer than this.

Figure 14 illustrates the structure of a message placed on the wire. This includes also illustrates how the message elements defined by the OPC UA Binary Encoding mapping (see 5.2) and the OPC UA Secure Conversation mapping (see 6.4) relate to the OPC UA TCP messages.

The socket is always closed gracefully by the Server after it sends an Error message.

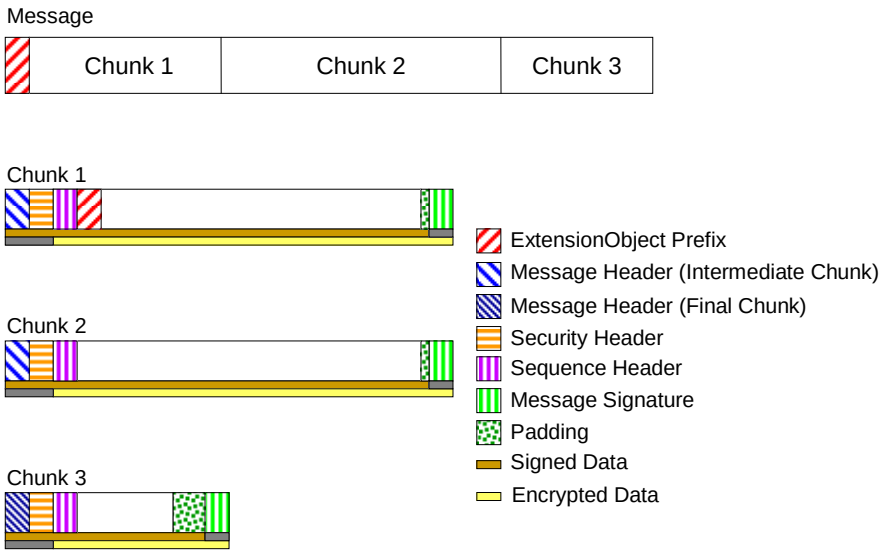

Figure 14 – OPC UA TCP Message Structure

7.1.3 Establishing a Connection

Connections are always initiated by the client which creates the socket before it sends the first *OpenSecureChannel* request. After creating the socket the first message sent shall be a *Hello* which specifies the buffer sizes that the client supports. The server shall respond with an Acknowledge message which completes the buffer negotiation. The negotiated buffer size shall be reported to the *SecureChannel* layer. The negotiated *SendBufferSize* specifies the size of the *MessageChunks* to use for messages sent over the connection.

The *Hello/Acknowledge* messages may only be sent once. If they are received again the receiver shall report an error and close the socket. *Servers* shall close any socket after a period of time if it does not receive a *Hello* message. This period of time shall be configurable and have a default value which does not exceed two minutes.

The client sends the *OpenSecureChannel* request once it receives the *Acknowledge* back from the server. If the server accepts the new channel it shall associate the socket with the *SecureChannelId*. The server uses this association to determine which socket to use when it has to send a response to the client. The client does the same when it receives the *OpenSecureChannel* response.

The sequence of messages when establishing a OPC UA TCP connection are shown in Figure 15.

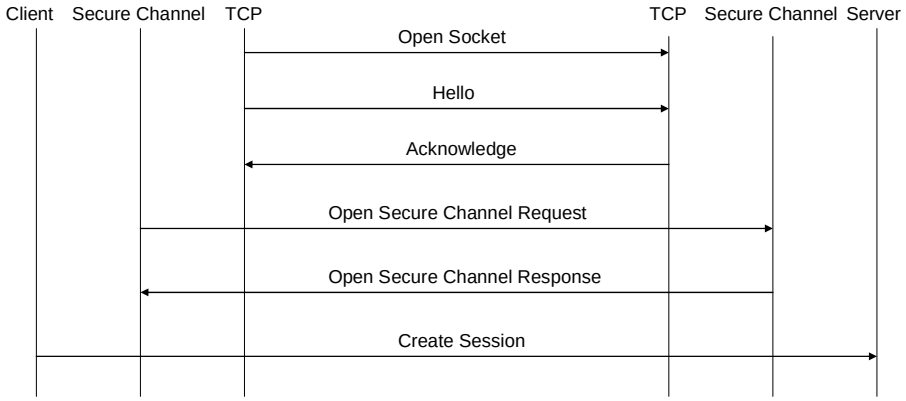

Figure 15 – Establishing a OPC UA TCP Connection

The *Server* application does not do any processing while the *SecureChannel* is negotiated, however, the *Server* application shall to provide the *Stack* with the list of trusted *Certificates*. The *Stack* shall provide notifications to the *Server* application whenever it receives an *OpenSecureChannel* request. These notifications shall include the *OpenSecureChannel* or *Error* response returned to the Client.

7.1.4 Closing a Connection

The client closes the connection by sending a *CloseSecureChannel* request and closing the socket gracefully. When the server receives this message it shall release all resources allocated for the channel. The server does not send a *CloseSecureChannel* response.

If security verification fails for the *CloseSecureChannel* message then the Server shall report the error and close the socket. The Server shall allow the Client to attempt to reconnect.

The sequence of messages when closing a OPC UA TCP connection are shown in Figure 16.

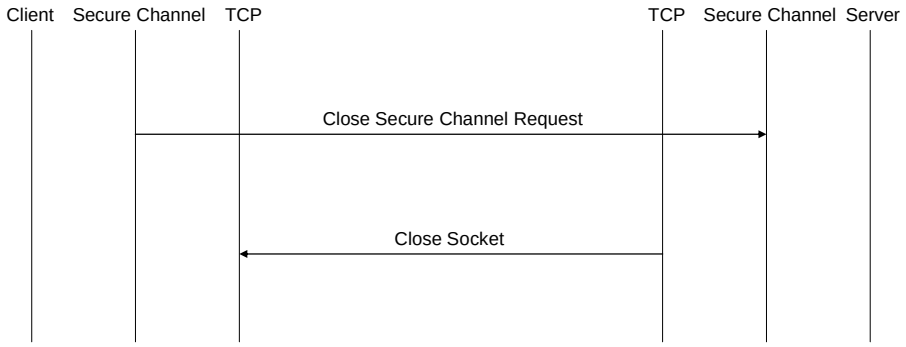

Figure 16 – Closing a OPC UA TCP Connection

The *Server* application does not do any processing when the *SecureChannel* is closed, however, the *Stack* shall provide notifications to the *Server* application whenever a *CloseSecureChannel* request is received or when the *Stack* cleans up an abandoned *SecureChannel*.

7.1.5 Error Handling

When a fatal error occurs the server shall send an *Error* message to the client and close the socket. When a client encounters one of these errors, it shall also close the socket but does not send an Error message. After the socket is closed a client shall try to reconnect automatically using the mechanisms described in Clause 7.1.6.

The possible OPC UA TCP errors are defined in Table 40.

Table 40 – OPC UA TCP Error Codes

Name	Description
TcpServerTooBusy	The server cannot process the request because it is too busy. It is up to the server to determine when it needs to return this message. A server can control the how frequently a client reconnects by waiting to return this error.
TcpMessageTypeInvalid	The type of the message specified in the header invalid. Each message starts with a 4 byte sequence of ASCII values that identifies the message type. The server returns this error if the message type is not accepted. Some of the message types are defined by the *SecureChannel* layer.
TcpSecureChannelUnknown	The SecureChannelId and/or TokenId are not currently in use. This error is reported by the SecureChannel layer.
TcpMessageTooLarge	The size of the message specified in the header is too large. The server returns this error if the message size exceeds its maximum buffer size or the receive buffer size negotiated during the Hello/Acknowledge exchange.
TcpTimeout	A timeout occurred while accessing a resource. It is up to the server to determine when a timeout occurs.
TcpNotEnoughResources	There are not enough resources to process the request. The server returns this error when it runs out of memory or encounters similar resource problems. A server can control the how frequently a client reconnects by waiting to return this error.
TcpInternalError	An internal error occurred. This should only be returned if an unexpected configuration or programming error occurs.
TcpUrlRejected	The Server does not recognize the EndpointUrl specified.
SecurityChecksFailed	The message was rejected because it could not be verified.
RequestInterrupted	The request could not be sent because of a network interruption.
RequestTimeout	Timeout occurred while processing the request.
SecureChannelClosed	The secure channel has been closed.
SecureChannelTokenUnknown	The token has expired or is not recognized.
CertificateUntrusted	The sender certificate is not trusted by the receiver.
CertificateTimeInvalid	The sender certificate has expired or is not yet valid.
CertificateIssuerTimeInvalid	The issuer for the sender certificate has expired or is not yet valid.
CertificateUseNotAllowed	The sender's certificate may not be used for establishing a secure channel.
CertificateIssuerUseNotAllowed	The issuer certificate may not be used as a Certificate Authority.
CertificateRevocationUnknown	Could not verify the revocation status of the sender's certificate.
CertificateIssuerRevocationUnknown	Could not verify the revocation status of the issuer certificate.
CertificateRevoked	The sender certificate has been revoked by the issuer.
IssuerCertificateRevoked	The issuer certificate has been revoked by its issuer.
CertificateUnknown	The receiver certificate thumbprint is not recognized by the receiver.

The numeric values for these error codes are defined in A.2.

7.1.6 Error Recovery

Once the *SecureChannel* has been established, the client shall go into a error recovery state whenever the socket breaks or if the server returns an OPC UA TCP Error message as defined in Table 39. While in this state the client periodically attempts to reconnect to the server. If the reconnect succeeds the client sends a Hello followed by an *OpenSecureChannel* request (see Clause 6.4.4) that re-authenticates the client and associates the new socket with the existing *SecureChannel*.

The client shall wait between reconnect attempts. The first reconnect shall happen immediately. After that the wait period should start as 1 second and increase gradually to a maximum of 2 minutes. One sequence would double the period each attempt until reaching

the maximum. In other words, the client would use the following wait periods: { 0, 1, 2, 4, 8, 16, 32, 64, 120, 120, ...}. The client shall keep attempting to reconnect until the *SecureChannel* is closed or after the period equal to the RevisedLifetime of the last SecurityToken elapses.

The stack in the server should not discard responses if there is no connection immediately available. It should wait and see if the client creates a new socket. It is up to the server stack implementation to decide how long it will wait and how many responses it is willing to hold onto.

The stack in the Client shall not fail requests that have already been sent and are waiting for a response when the socket is closed, however, these requests may timeout and report a Bad_TcpRequestTimeout error to the application. If the client sends a new request the stack shall either buffer the request or return a Bad_TcpRequestInterrupted error. The client can stop the reconnect process by closing the *SecureChannel*.

The Server may abandon the *SecureChannel* before a *Client* is able to reconnect. If this happens the *Client* will get a Bad_TcpSecureChannelUnknown error in response to the *OpenSecureChannel* request. The stack shall return this error to the application that can attempt to create a new *SecureChannel*.

The negotiated buffer sizes should never change when a connection is recovered, however, the buffer sizes are negotiated before the server knows whether the socket is being used for an existing *SecureChannel* or a new one. A client shall treat this as a fatal error, closes the SecureChannel and returns an *Bad_TcpSecureChannelClosed* error to the application.

The sequence of messages when recovering an OPC UA TCP connection are shown in Figure 17.

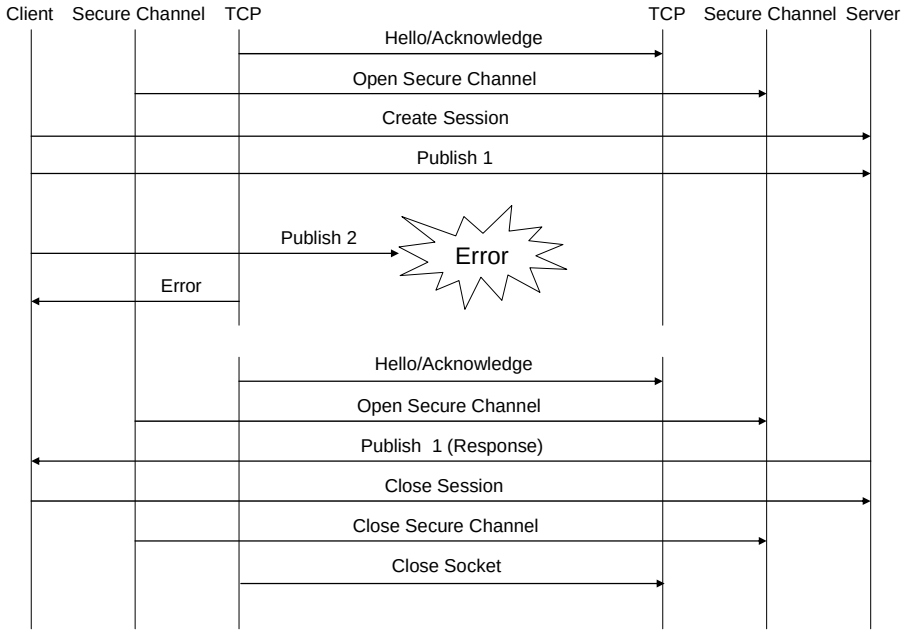

Figure 17 – Recovering an OPC UA TCP Connection

7.2 SOAP/HTTP

7.2.1 Overview

SOAP describes an XML based syntax for exchanging messages between applications. OPC UA messages are exchanged using SOAP by serializing the OPC UA messages using one of the supported encodings described in Clause 0 and inserting that encoded message into the body of the SOAP message.

All OPC UA applications that support the SOAP/HTTP transport shall support SOAP 1.2 as described in SOAP Part 1.

All OPC UA messages are exchanged using the request-response message exchange pattern defined in SOAP Part 2 even if the OPC UA service does not specify any output parameters. In these cases, the server shall return an empty response message that tells the client that no errors occurred.

WS-I Basic Profile 1.1 defines best practices when using SOAP messages which will help ensure interoperability. All OPC UA implementations shall conform to this specification.

HTTP is the network communication protocol used to exchange SOAP messages. An OPC UA service request message is always sent by the client in the body of an HTTP POST request. The server returns an OPC UA response message in the body of the HTTP response. The HTTP binding for SOAP is described completely in SOAP Part 2.

HTTPS is a variant of HTTP that encrypts and/or signs HTTP messages using the SSL/TLS protocol. HTTPS provides a efficient way to encrypt data sent across the network when two applications can communicate directly without intermediaries.

OPC UA does not define any SOAP headers, however, SOAP messages containing OPC UA messages will include headers used by the other WS specifications in the stack.

SOAP faults are returned only for errors that occurred with in the SOAP stack. Errors that occur within in the application are returned as OPC UA error response messages as described in Clause 5.3 of Part 4.

WS Addressing defines standard headers used to route SOAP messages through intermediaries. Implementations shall support the WS-Addressing headers listed Table 41.

Table 41 – WS-Addressing Headers

Header	Request	Response
wsa:To	Required	Optional
wsa:From	Optional	Optional
wsa:ReplyTo	Required	Not Used
wsa:Action	Required	Required
wsa:MessageID	Required	Optional
wsa:RelatesTo	Not Used	Required

Note that WS-Addressing defines standard URIs to use in the ReplyTo and From headers when a client does not have an externally accessible endpoint. In these cases, the SOAP response is always returned to the client using the same communication channel that sent the request.

OPC UA servers shall ignore the wsa:FaultTo header if it is specified in a request.

7.2.2 XML Encoding

The OPC UA XML Encoding specifies a way to represent an OPC UA message as an XML element. This element is added to the SOAP message as the only child of the SOAP body element.

If an error occurs in the server while parsing the request body, the server may return a SOAP fault or it may return an OPC UA error response.

The SOAP Action associated with an XML encoded request message always has the form:

```
http://opcfoundation.org/UA/<service name>
```

Where <service name> is the name of the OPC UA service being invoked.

The SOAP Action associated with an XML encoded response message always has the form:

```
http://opcfoundation.org/UA/<service name>Response
```

7.2.3 OPC UA Binary Encoding

The OPC UA Binary Encoding specifies a way to represent an OPC UA message as a sequence of bytes. These bytes sequences shall be encoded in the SOAP body using the Base64data format.

The Base64 data format is a UTF-7 representation of binary data that is less efficient than raw binary data, however, many OPC UA applications that exchange messages using SOAP will find that encoding OPC UA messages in OPC UA Binary and then encoding the binary in Base64 is more efficient than encoding everything in XML.

The WSDL definition for a UA Binary encoded request message is:

```
<xs:element          name="InvokeService"          type="xs:base64Binary"
nillable="true" />

<wsdl:message name="InvokeServiceMessage">
  <wsdl:part name="InvokeService" element="tns:InvokeService" />
</wsdl:message>
```

The SOAP Action associated with a OPC UA Binary encoded request message always has the form:

```
http://opcfoundation.org/UA/InvokeService
```

The WSDL definition for an OPC UA Binary encoded response message is:

```
<xs:element name="InvokeServiceResponse" type="xs:base64Binary"
   nillable="true" />

<wsdl:message name="InvokeServiceResponseMessage">
  <wsdl:part                                  name="InvokeServiceResponse"
element="tns:InvokeServiceResponse" />
</wsdl:message>
```

The SOAP Action associated with an OPC UA Binary encoded response message always has the form:

http://opcfoundation.org/UA/InvokeServiceResponse

7.3 Well Known Addresses

The Local Discovery Server (LDS) is a UA Server that implements the Discovery Service Set defined in Part 4. If an LDS is installed on a machine it shall use one or more of the well-known addresses defined in Table 42.

Table 42 – Well Known Addresses for Local Discovery Servers

Transport Mapping	URL	Notes
SOAP/HTTP	http://localhost/UADiscovery	May require integration with a web server like IIS.
SOAP/HTTP	http://localhost:52601/UADiscovery	Alternate if it Port 80 cannot be used by the LDS.
UA TCP	opc.tcp://localhost:4840/UADiscovery	

UA Applications that make use of the LDS shall allow Administrators to change the well known addresses used within a system.

The endpoint used by servers to register with the LDS shall be the base address with the path "/registration" appended to it (e.g. http://localhost/UADiscovery/registration). UA Servers shall allow adminstrators to configure the address to use for registration.

Each UA Server application implements the Discovery Service Set. If the UA Server requires a different address for this endpoint it shall create the address by appending the path "/discovery" to its base address.

8 Normative Contracts

8.1 OPC Binary Schema

The normative contract for the OPC UA Binary encoded messages is an OPC Binary Schema. This file defines the structure of all types and messages. The syntax for an OPC Binary Type Schema is described in Appendix B of Part 3. This schema captures normative names for types and their fields as well the order the fields appear when encoded. The data type of each field is also captured.

8.2 XML Schema and WSDL

The normative contract for the OPC UA XML encoded messages is an XML Schema. This file defines the structure of all types and messages. This schema captures normative names for types and their fields as well the order the fields appear when encoded. The data type of each field is also captured.

The normative contract for message sent via the SOAP/HTTP *TransportProtocol* is a WSDL that includes XML Schema for the OPC UA XML encoded messages. It also defines the port types for OPC UA *Servers* and *DiscoveryServers*.

Links to the WSDL and XML Schema files can be found in Annex C.

Annex A. Constants

A.1 Attribute Ids

Table 43 – Identifiers Assigned to Attributes

Attribute	Identifier
NodeId	1
NodeClass	2
BrowseName	3
DisplayName	4
Description	5
WriteMask	6
UserWriteMask	7
IsAbstract	8
Symmetric	9
InverseName	10
ContainsNoLoops	11
EventNotifier	12
Value	13
DataType	14
ValueRank	15
ArrayDimensions	16
AccessLevel	17
UserAccessLevel	18
MinimumSamplingInterval	19
Historizing	20
Executable	21
UserExecutable	22

A.2 Status Codes

This appendix defines the numeric identifiers for all of the StatusCodes defined by the OPC UA Specification. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <SubCode>
```

Where the *SymbolName* is the literal name for the error code that appears in the specification and the *SubCode* is numeric value for the *SubCode* field within a *StatusCode* (See Clause 7.33 in Part 4). The severity associated with particular code is specified by the prefix (*Good*, *Uncertain* or *Bad*).

The CSV associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/StatusCodes.csv

The most recent set of StatusCodes can be found here:

http://www.opcfoundation.org/UAPart6/StatusCodes.csv

A.3 Numeric Node Ids

This appendix defines the numeric identifiers for all of the numeric *NodeIds* defined by the OPC UA Specification. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <Identifier>, <NodeClass>
```

Where the *SymbolName* is either the *BrowseName* of a *Type Node* or the *BrowsePath* for an *Instance Node* that appears in the specification and the *Identifier* is numeric value for the *NodeId*.

The *BrowsePath* for an instance *Node* is constructed by appending the *BrowseName* of the instance *Node* to *BrowseName* for the containing instance or type. A '_' character is used to separate each *BrowseName* in the path. For example, Part 5 defines the *ServerType ObjectType Node* which has the *NamespaceArray Property*. The *SymbolName* for the *NamespaceArray InstanceDeclaration* within the *ServerType* declaration is: *ServerType_NamespaceArray*. Part 5 also defines a standard instance of the *ServerType ObjectType* with the *BrowseName* 'Server'. The *BrowseName* for the *NamespaceArray Property* of the standard *Server Object* is: *Server_NamespaceArray*.

The *NamespaceUri* for all *NodeIds* defined here is http://opcfoundation.org/UA/

The CSV associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/NodeIds.csv

The most recent set of *NodeIds* can be found here:

http://www.opcfoundation.org/UAPart6/NodeIds.csv

Annex B. Type Declarations for the OPC UA Native Mapping

This appendix defines the OPC UA Binary encoding for all DataTypes and Messages defined in the specification. The schema used to describe the type is defined in Appendix C of Part 3.

The OPC UA Binary Schema associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/Types.bsd.xml

The most recent OPC UA Binary Schema can be found here:

http://www.opcfoundation.org/UAPart6/Types.bsd.xml

Annex C. WSDL for the XML Mapping

C.1 XML Schema

This appendix defines the XML Schema for all DataTypes and Messages defined in the specification.

The XML Schema associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/Types.xsd

The most recent XML Schema can be found here:

http://www.opcfoundation.org/UAPart6/Types.xsd

C.2 WDSL Port Types

This appendix defines the WSDL Operations and Port Types for all Services defined in Part 4.

The WSDL associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/PortTypes.wsdl

The most recent WSDL can be found here:

http://www.opcfoundation.org/UAPart6/PortTypes.wsdl

This WSDL imports the XML Schema defined in C.1.

C.3 WSDL Bindings

This appendix defines the WSDL Bindings for all Services defined in Part 4.

The WSDL associated with this version of the specification can be found here:

http://www.opcfoundation.org/UA/2008/02/Bindings.wsdl

The most recent WSDL can be found here:

http://www.opcfoundation.org/UAPart6/Bindings.wsdl

This WSDL imports the WSDL defined in C.2.

Annex D. Security Settings Management

D.1 Overview

All UA applications shall support security, however, this requirement means that Adminstrators need to configure the security settings for the UA application. This appendix describes an XML Schema which can be used to read and update the security settings for a UA application. All UA applications shall support configuration by importing/exporting documents that conform to the schema defined in this Annex.

D.2 SecuredApplication

The SecuredApplication element specifies the security settings for an Application. The elements contained in a SecuredApplication are described in Table 44.

When a instance of a SecuredApplication is imported into an Application the Application updates its configuration based on the information contained within it. If unrecoverable errors occur during import an Application shall not make any changes to its configuration and report the reason for the error.

When a instance of a SecuredApplication is exported from an Application the Application reads its configuration and stores it in the SecuredApplication element.

The mechanism used to import or export the configuration depends on the Application. Applications shall ensure that only authorized users are able to access this feature.

The SecuredApplication element may reference X509 Certificates which are contained in shared physical stores. The managment of X509 Certificates contained within these physical stores is outside the scope of this Annex. Applications are expected to use these physical stores and shall not make private copies of these shared stores when the configuration is imported.

Note that some elements are optional or read only. Administrators are expected to export the current configuration, update it and then import the changes back. During export, Applications shall omit or leave empty any elements which are not supported. During import, Applications shall raise an error if read only elements are change and Applications shall report an error if unsupported elements are added.

The ExportTime and Version fields are two exceptions to the above rule. These fields may be used by an application to determine if an imported configuration was based on current configuration. Applications may report an error if an older configuration is being imported.

Table 44 – SecuredApplication

Element	Type	Description
Name	String	A human readable name for the application. Applications shall allow this value to be read or changed.
Uri	String	A globally unique identifier for the instance of the application. Applications shall allow this value to be read or changed.
ProductName	String	A name for the product. Application shall provide this value. Applications do not allow this value to be changed.
ApplicationType	ApplicationType	The type of Application. May be one of Server_0, Client_1, ClientAndServer_2 or DiscoveryServer_3. Application shall provide this value. Applications do not allow this value to be changed.
ConfigurationFile	String	The full path to a configuration file used by the application. Applications may not provide this value. Applications do not allow this value to be changed. Permissions set on this file shall control who has rights to change the configuration of the Application.
ExecutableFile	String	The full path to a executable file used by the application. Applications may not provide this value. Applications do not allow this value to be changed. Permissions set on this file shall control who has rights to launch the Application.
ApplicationCertificate	CertificiateIdentifier	The identifier for the application certificate. Applications shall allow this value to be read or changed. After export this value shall be a reference to a CertificateStore which contains the private key associated with Certificate. The private key is never exported. A new certificate with a private key may be provided during an import. Applications are expected to save the private key in a CertificateStore accessible to the Application. Applications shall allow Administrators to enter the password required to access the private key during the import operation. The exact mechanism depends on the Application. Applications shall report an error if the ApplicationCertificate is not valid.
TrustedPeerStore	CertificateStore Identifier	The location of the CertificateStore containing the Certificates of Applications which can be trusted. Applications shall allow this value to be read or changed. This value shall be a reference to a physical store which can be managed seperately from the Application. Applications shall check this store for changes whenever they validate a Certificate. The Administrator is responsible for verifying the signature on all Certificates placed in this store. This means the Application shall trust Certificates in this store even if they cannot verified back to a trusted root. The Application shall check the revocation status of the Certificates in this store if the Certificate Issuer is found in the TrustedIssuerStore or the TrustedIssuerCertificates.
TrustedPeerCertificates	CertificateTrustList	A list of Certificates for Applications that can be trusted. Applications shall allow this value to be read or changed. The value is a explicit list of Certificates which is private to the Application. Each CertificateIdentifier may have ValidationOptions specified. The Application shall use these ValidationOptions when validating the Certificate. When validating certificates Applications shall check this list before checking the TrustedPeerStore. The Application shall check the revocation status of the Certificates in this list if the Certificate Issuer is found in the TrustedIssuerStore or the TrustedIssuerCertificates. Applications shall ignore an entries in the list which are invalid.
TrustedIssuerStore	CertificateStore Identifier	The location of the CertificateStore containing the Certificates of Issuers which can be trusted.

		Applications shall allow this value to be read or changed. This value shall be a reference to a physical store which can be managed seperately from the Application. Applications shall check this store for changes whenever they validate a Certificate. The Administrator is responsible for verifying the signature on all Certificates placed in this store. This means the Application shall trust Certificates in this store even if they cannot verified back to a trusted root. The Application shall check the revocation status of the Certificates in this store if the Certificate Issuer is found in the TrustedIssuerStore or the TrustedIssuerCertificates.
TrustedIssuerCertificates	CertificateTrustList	A list of Certificates for Issuers that can be trusted. Applications shall allow this value to be read or changed. The value is a explicit list of Certificates which is private to the Application. Each CertificateIdentifier may have ValidationOptions specified. The Application shall use these ValidationOptions when validating the Certificate or when validating Certificates issued by the Certificate. When validating certificates Applications shall check this list before checking the TrustedIssuerStore. The Application shall check the revocation status of the Certificates in this list if the Certificate Issuer is found in the TrustedIssuerStore or the TrustedIssuerCertificates.
RejectedCertificatesStore	CertificateStore Identifier	The location of the shared CertificateStore containing the Certificates of Applications which were rejected. Applications shall allow this value to be read or changed. Applications shall add the DER encoded Certificate into this store whenever it rejects a Certificate because it is untrusted or if it failed one of the validation rules which can be suppressed (see Clause D.6). Applications shall not add a Certificate to this store if it was rejected for a reason that cannot be suppressed (e.g. certificate revoked).
BaseAddresses	String[]	A list of base URLs for endpoints exposed by a Server Application. Application shall not provide this value if it cannot be changed. This value is ignored by Applications which are only Clients. A Server Application shall raise an error if provided a URL that it cannot used because it is not accessible or specifies a protocol not supported by the Application.
SecurityPolicies	ApplicationSecurityPolicy	A list of security policies permitted by a ServerApplication. Application shall not provide this value if it cannot be changed. A Server Application shall raise an error if provided a SecurityPolicy that it cannot used because it is not supported by the Application.
WelKnownDiscoveryUrls	String[]	A list of URLs for Local Discovery Servers used by a Client Application. Applications shall allow this value to be changed. The Application is expected to use the URLs in the order that they appear. The URL for a discovery server running on a specific host can be constructed by replacing the host name portion of the URL with name of the target host.
RegistrationEndpoint	EndpointDescription	The EndpointDescription for the Local Discovery Server registration endpoint. Applications shall allow this value to be changed. This value is ignored by Applications which are only Clients.
DiscoveryEndpoints	EndpointDescription[]	A list of EndpointDescriptions for global Discovery Servers. Applications shall allow this value to be changed. This value is ignored by Applications which are only Servers.
AccessRules	ApplicationAccessRule[]	A list of rules used to control access to an application. Application shall not provide this value if it cannot be changed. These rules are used to control access to the Application.
ExportTime	UtcTime	When the configuration was exported from the Application.
Version	String	An optional string identifier for the version of the Application configuration. This field is used to detect out of date configurations during import.

D.3 CertificateIdentfier

The CertificateIdentifier element describes a X509 Certificate. The Certificate can be provided explicitly within the element or the element can specify the location of the CertificateStore that

contains the Certificate. The elements contained in a CertificateIdentifier are described in Table 45.

Table 45 – CertificateIdentfier

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the Certificate. Predefined values are "Windows", "Directory" and "OpenSSL". If not specified the RawData and/or PrivateKey element shall be specified.
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType.
SubjectName	String	The SubjectName for the Certificate. The Common Name (CN) component of the SubjectName. The SubjectName represented as a string that complies with Section 3 of RFC 4514. Values that do not contain '=' characters are presumed to be the Common Name component.
Thumbprint	String	The SHA1 thumbprint for the Certificate formatted as a hexadecimal string. Case is not significant.
RawData	ByteString	The DER encoded Certificate. The CertificateIdentifier is invalid if the information in the DER certificate conflicts with the information specified in other fields.
PrivateKey	ByteString	The private key of the Certificate encoded as a PKCS #12 blob protected by a password. The CertificateIdentifier is invalid if the information in the PKCS #12 blob conflicts with the information specified in other fields.
ValidationOptions	Int32	The options to use when validating the certificate. The possible options are described in Clause D.6.
Issuer	CertificateIdentifier	The issuer for the Certificate. This field specifies the trust chain for the Certificate.
OfflineRevocationList	ByteString	A Certificate Revocation List (CRL) associated with an Issuer certificate. The format of a CRL is defined by RFC 3280. This field is only meaningful for Issuer Certificates.
OnlineRevocationList	String	A URL for a Online Revocation List associated with an Issuer certificate. This field is only meaningful for Issuer Certificates.

A "Windows" StoreType specifies a Windows certificate store.

The syntax of the StorePath has the form:

[\\HostName\]StoreLocation[\(ServiceName | UserSid)]\StoreName

where:

HostName - the name of the machine where the store resides.

StoreLocation - one of LocalMachine, CurrentUser, User or Service

ServiceName - the name of a Windows Service.

UserSid - the SID for a Windows user account.

StoreName - the name of the store (e.g. My, Root, Trust, CA, etc.).

Examples of Windows StorePaths are:

\\MYPC\LocalMachine\My

\CurrentUser\Trust

\\MYPC\Service\My UA Server\UA Applications

\User\S-1-5-25\Root

A "Directory" StoreType specifies a directory on disk which contains files with DER encoded Certificates. The name of the file is the SHA1 thumbprint for the Certificate. Only public keys may be placed in a "Directory" Store. The StorePath is an absolute file system path with a syntax that depends on the operating system.

An "OpenSSL" StoreType specifies the root directory of a OpenSSL compatible CertificateStore. The StorePath is an absolute file system path with a syntax that depends on the operating system.

The public keys for X509 Certificates are exchanged as DER encoded blobs as described in Clause 6.2. The private keys for X509 Certificates with private keys are exchanged as PKCS #12 encoded blobs which are protected by a password. When applications import configuration documents containing PKCS #12 encoded blobs they shall allow Administrators to provide a password.

Each certificate is uniquely identified by its Thumbprint. The SubjectName or the distinguished SubjectName may be used to identify a certificate to a human, however, they are not unique. The SubjectName may be specified in conjuction with the Thumbprint or the RawData. If there is an inconsistency between the information provided then the CertificateIdentifier is invalid. Invalid CertificateIdentifiers are handled differently depending on where they are used.

It is recommended that the SubjectName always be specified.

A certificate revocation list (CRL) contains a list of certificates issued by a CA that are no longer trusted. These lists should be checked before an application can trust a certicated issued by a trusted CA. The format of a CRL is defined by RFC 3280.

Offline CRLs are placed in a local certificate store with the Issuer certificate. Online CRLs may exist but the protocol depends on the system. An online CRL is identified by a URL.

D.4 CertificateStoreIdentifier

The CertificateStoreIdentifier element describes a physical store containing X509 Certificates. The elements contained in a CertificateIdentifier are described in Table 46.

Table 46 – CertificateStoreIdentfier

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the Certificate. Predefined values are "Windows", "Directory" and "OpenSSL".
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType. See Clause D.3 for a description of the syntax for different StoreTypes.
ValidationOptions	Int32	The options to use when validating the Certificates contained in the store. The possible options are described in **Error! Reference source not found.**.

All certificates are placed in a physical store which can be protected from unauthorized access. The implementation of a store can vary and will depend on the application, development tool or operating system. A certificate store may be shared by many applications on the same machine.

Each certificate store is identified by a StoreType and a StorePath. The same path on different machines identifies a different store.

D.5 CertificateTrust List

The CertificateTrustList element is a CertificateStore where the X509 Certificates are specified within the XML document. CertificateTrustList is a extends the CertificateStoreIdentifier defined in Clause D.4 and adds the elements described in Table 47.

Table 47 – CertificateTrustList

Element	Type	Description
TrustedCertficates	CertificateIdentifier[]	The list of Certificates contained in the Trust List

The StoreType and StorePath elements are ignored for CertificateTrustLists.

D.6 CertificateValidationOptions

The CertificateValidationOptions control the process used to validate a Certificate. Any Certificate can have validation options associated with it if it is placed in TrustedPeerCertificates or TrustedIssuerCertificates lists. The possible values are described in Table 48.

Table 48 – CertificateValidationOptions

Field	Value	Description
SuppressCertificateExpired	0x01	Ignore errors related to the validity time of the certificate or its issuers.
SuppressHostNameInvalid	0x02	Ignore mismatches between the host name or application uri.
SuppressUseNotAllowed	0x04	Ignore restrictions on the allowed uses for the certificate.
SuppressRevocationStatusUnknown	0x08	Ignore errors if the issuer's revocation list cannot be found.
CheckRevocationStatusOnline	0x10	Check the revocation status online. This option is specified for Issuer Certificates and used when validating Certificates issued by that Issuer.
CheckRevocationStatusOffine	0x20	Check the revocation status offline. This option is specified for Issuer Certificates and used when validating Certificates issued by that Issuer.
DoNotTrust	0x40	The certificate must not be trusted. If this option is applied to an Issuer Certificates then all Certificates issued by that Certificate must not be trusted.

D.7 ApplicationAccessRule

ApplicationAccessRule specifies the access rights to an Application granted to a user or group. The mechanisms used to enforce access rules depend on the operating system. The elements in an ApplicationAccessRule are described in Table 48.

Table 49 – ApplicationAccessRule

Field	Type	Description
RuleType	AccessControlType	The type of rule. Allow means the right is granted to the specified user or group. Deny means the right is denied to the specified user or group. A Deny rule always takes priority over an Allow rule if multiple rules apply to a single user.
Right	ApplicationAccessRight	The type of access granted or revoked. Run means the user or group can launch the application. Update means the user or group can update the application configuration. Configure means the user or group can change the access rights of other users or groups. The access rights are cumulative. That is, Update grants the permissiom to Run and Configure grants the permission to Run or Update.
IdentityName	String	The name of the user or group. The syntax of this filed depends on the operating system. On Windows it is a domain qualified user name or an account SID. Account SIDs shall be used for well known accounts such as 'Administrators'.

D.8 ApplicationSecurityPolicy

ApplicationSecurityPolicy specifies the security policies permitted by an Application. The elements in an ApplicationSecurityPolicy are described in Table 48.

Table 50 – ApplicationSecurityPolicy

Field	Type	Description
UrlScheme	String	The URL scheme that the policy applies to. If blank it applies to all types of endpoints.
Address	String	The address that the policy applies to. If this is a complete URL then the policy only applies to the specified URL. Non-URLs may be used to construct URLs from the BaseAddresses. A URL that does not start with one of the BaseAddresses is ignored.
SecurityPolicyUri	String	The URI of the SecurityPolicy to use. These URIs are defined in Part 7.
SecurityModes	Int32	A mask that specifies the security modes that are permitted. The following value are permitted: 0x1 - Sign 0x2 - SignAndEncrypt A value of 0 indicates the no security is used.
SecurityLevel	Byte	A relative measure of the security provided by the policy. A higher number means better security. A value of 0 indicates the policy is not recommended.

Client Applications may use these policies to restrict access to Servers that do not support one of the specified policies.

Server Applications may need to construct multiple endpoints for each ApplicationSecurityPolicy. The mechanisms used to construct the different endpoint URLs are specific to the Server.