IEC 62541-6

Edition 2.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 6: Mappings

Architecture unifiée OPC – Partie 6: Correspondances

IEC 62541-6:2015-03(en-fr)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

IEC 62541-6

Edition 2.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 6: Mappings

Architecture unifiée OPC – Partie 6: Correspondances

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

ICS 25.040.40; 35.100

ISBN 978-2-8322-2373-4

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 6: Mappings

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 62541-6 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2011. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

a) Some applications need to operation in environments with no access to cryptography libraries. To support this a new HTTPS transport has been defined in 7.3;
b) The padding byte is not long enough to handle asymmetric key sizes larger than 2048 bits. Added an additional padding byte to 6.7.2 to handle this case.
c) Fixed errors in SOAP action URIs defined in 7.2.2;

d) Needed a standard way to serialize nodes in an address space. Added the UANodeSet schema defined in Annex F;

The text of this standard is based on the following documents:

CDV	Report on voting
65E/377/CDV	65E/405/RVC

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC UNIFIED ARCHITECTURE –

Part 6: Mappings

1 Scope

This part of IEC 62541 specifies the OPC Unified Architecture (OPC UA) mapping between the security model described in IEC TR 62541-2, the abstract service definitions, described in IEC 62541-4, the data structures defined in IEC 62541-5 and the physical network protocols that can be used to implement the OPC UA specification.

2 Normative references

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

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC TR 62541-2, *OPC Unified Architecture – Part 2: Security Model*

IEC 62541-3, *OPC Unified Architecture – Part 3: Address Space Model*

IEC 62541-4, *OPC Unified Architecture – Part 4: Services*

IEC 62541-5, *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-7, *OPC Unified Architecture – Part 7: Profiles*

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 5246 – The TLS Protocol Version 1.2
http://tools.ietf.org/html/rfc5246.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

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

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

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

RFC 4514: RFC 4514 – LDAP: String Representation of Distinguished Names

http://www.ietf.org/rfc/rfc4514.txt

NTP: RFC 1305 – Network Time Protocol (Version 3)

http://www.ietf.org/rfc/rfc1305.txt

Kerberos: WS Security Kerberos Token Profile 1.1

http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-KerberosTokenProfile.pdf

3 Terms, definitions, abbreviations and symbols

3.1 Terms and definitions

For the purposes of this document the terms and definitions given in IEC TR 62541-1, IEC TR 62541-2 and IEC 62541-3 as well as the following apply.

3.1.1
DataEncoding
a way to serialize OPC UA *Messages* and data structures

3.1.2
Mapping
specifies how to implement an OPC UA feature with a specific technology

Note 1 to entry: For example, the OPC UA Binary Encoding is a *Mapping* that specifies how to serialize OPC UA data structures as sequences of bytes.

3.1.3
Security Protocol
ensures the integrity and privacy of UA *Messages* that are exchanged between OPC UA applications

3.1.4
Stack Profile
a combination of *DataEncodings*, *SecurityProtocol* and *TransportProtocol Mappings*

Note 1 to entry: OPC UA applications implement one or more *StackProfiles* and can only communicate with OPC UA applications that support a *StackProfile* that they support.

3.1.5
Transport Protocol
a way to exchange serialized OPC UA *Messages* between OPC UA applications

3.2 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
HTTPS	Secure Hypertext Transfer Protocol
IPSec	Internet Protocol Security
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	OPC UA Secure Conversation
WS-*	XML Web Services Specifications
WSS	WS Security
WS-SC	WS Secure Conversation
XML	Extensible Markup Language

4 Overview

Other parts of this series of standards 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 IEC TR 62541-1 through to IEC 62541-5 because important implementation details have been left out.

This standard 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 standard defines the *DataEncodings* in Clause 5, the *SecurityProtocols* in Clause 6 and the *TransportProtocols* in 6.7.6. The *StackProfiles* are defined in IEC 62541-7.

All communication between OPC UA *Applications* is based on the exchange of *Messages*. The parameters contained in the *Messages* are defined in IEC 62541-4; however, their format is specified by the *DataEncoding* and *TransportProtocol*. For this reason, each *Message* defined in IEC 62541-4 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 implement 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 an unsigned integer which means that 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 standard.

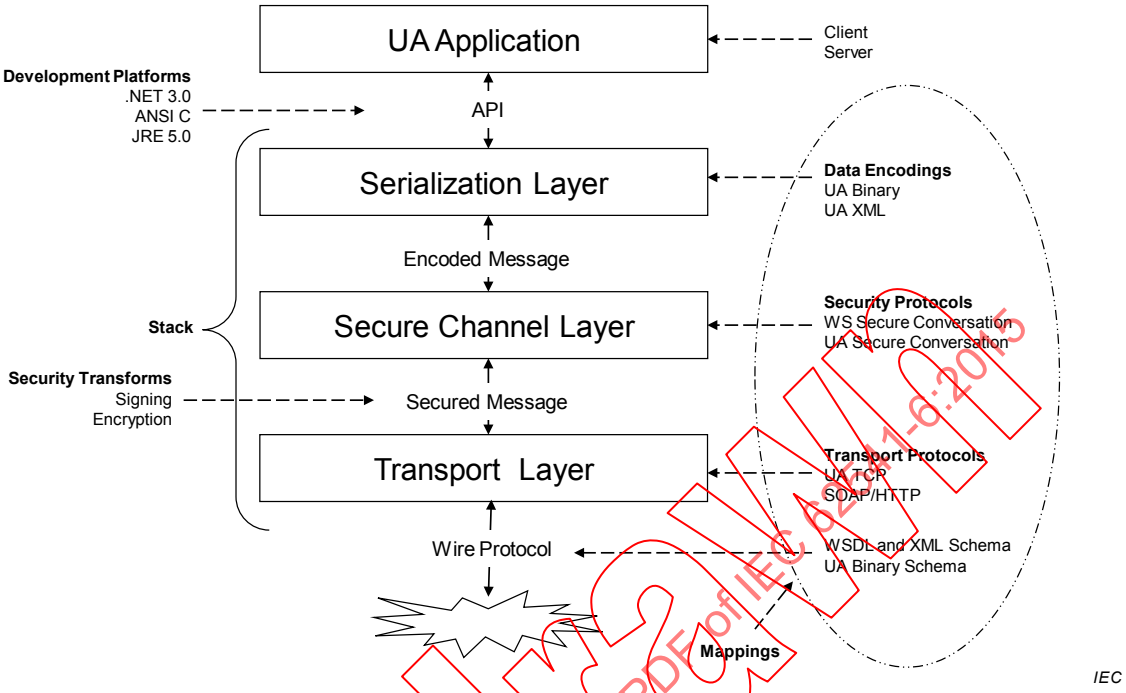

Figure 1 – The OPC UA Stack Overview

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 standard 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 256.
4	Int16	An integer value between −32 768 and 32 767.
5	UInt16	An integer value between 0 and 65 535.
6	Int32	An integer value between −2 147 483 648 and 2 147 483 647.
7	UInt32	An integer value between 0 and 429 4967 295.
8	Int64	An integer value between −9 223 372 036 854 775 808 and 9 223 372 036 854 775 807
9	UInt64	An integer value between 0 and 18 446 744 073 709 551 615.
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 IEC 62541-3 and IEC 62541-4. However, the *ExtensionObject* and *Variant* types are defined in this standard. In addition, this standard defines a representation for the *Guid* type defined in IEC 62541-3.

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 zeros. A typical *Guid* value would look like this when formatted as a string:

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

5.1.4 ByteString

A *ByteString* is structurally the same as a one dimensional array of *Byte*. It is represented as a distinct built-in data type because it allows encoders to optimize the transmission of the value. However, some *DevelopmentPlatforms* will not be able to preserve the distinction between a *ByteString* and a one dimensional array of *Byte*.

If a decoder for *DevelopmentPlatform* cannot preserve the distinction it shall convert all one dimensional arrays of *Byte* to *ByteStrings*.

Each element in a one dimensional array of *ByteString* can have a different length which means is structurally different from a two dimensional array of *Byte* where the length of each dimension is the same. This means decoders shall preserve the distinction between two or more dimension arrays of *Byte* and one or more dimension arrays of *ByteString*.

If a *DevelopmentPlatform* does not support unsigned integers then it will have to represent *ByteStrings* as arrays of *SByte*. In this case, the requirements for *Byte* would then apply to *SByte*.

5.1.5 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 DataType*. The *DataEncodings* available for any given *Complex Data* type are represented as a *DataTypeEncoding Object* in the *Server AddressSpace*. The *NodeId* for the *DataTypeEncoding Object* is the identifier stored in the *ExtensionObject*. IEC 62541-3 describes how *DataTypeEncoding Nodes* are related to other *Nodes* of the *AddressSpace*.

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*.

5.1.6 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*. Some *Development* Platforms may not be able to preserve the distinction between a null for a *DataType* and a null for a *Variant*. Therefore *Applications* shall not rely on this distinction.

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

DataValue and DiagnosticInfo types only have meaning when returned in a response message with an associated *StatusCode*. As a result, *Variants* cannot contain instances of *DataValue or 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 DataEncoding* 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 DataEncoding* 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 DataEncoding* 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 Data* 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.

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

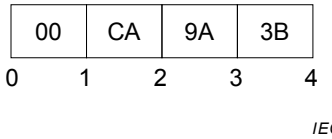

Figure 2 – Encoding Integers in a binary stream

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).

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 an IEEE quiet-NAN.

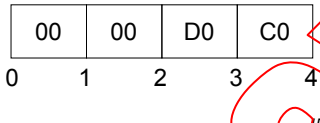

Figure 3 – Encoding Floating Points in a binary stream

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 "水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).

Not all *DevelopmentPlatforms* will be able to represent the full range of dates and times that can be represented with this *DataEncoding*. 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 *DevelopmentPlatform*. These rules are:

a) A date/time value is encoded as 0 if either
   1) The value is equal to or earlier than 1601-01-01 12:00AM.
   2) The value is the earliest date that can be represented with the *DevelopmentPlatform*'s encoding.

b) A date/time is encoded as the maximum value for an *Int64* if either

   1) The value is equal to or greater than 9999-01-01 11:59:59PM,

   2) The value is the latest date that can be represented with the *DevelopmentPlatform*'s encoding.

c) A date/time is decoded as the earliest time that can be represented on the platform if either

   1) The encoded value is 0,

   2) The encoded value represents a time earlier than the earliest time that can be represented with the *DevelopmentPlatform*'s encoding.

d) A date/time is decoded as the latest time that can be represented on the platform if either

   1) The encoded value is the maximum value for an *Int64*,

   2) The encoded value represents a time later than the latest time that can be represented with the *DevelopmentPlatform*'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 *DevelopmentPlatform*.

5.2.2.6 Guid

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

Figure 5 illustrates how the *Guid* "72962B91-FA75-4ae6-8D28-B404DC7DAF63" should be encoded in a byte stream.

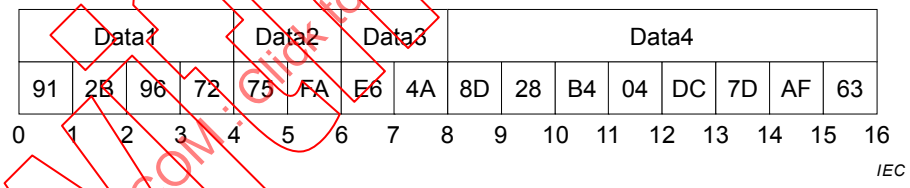

Figure 5 – Encoding Guids in a binary stream

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 UTF8 string and then encoded as *ByteString.*

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

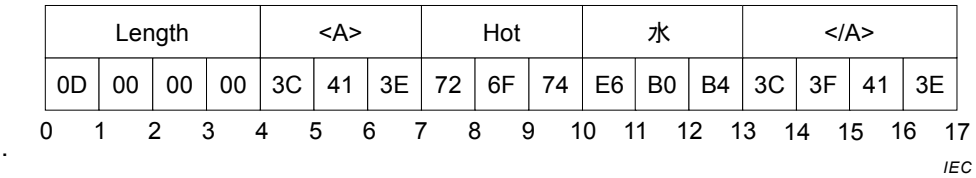

Figure 6 – Encoding XmlElements in a binary stream

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 *DataEncoding* 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 *DataEncoding* formats are shown in Table 5. The tables that follow describe the structure of each possible format (they exclude the byte which indicates the format).

Table 5 – NodeId DataEncoding 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 5.2.2.10.
ServerIndex Flag	0x40	See discussion of *ExpandedNodeId* in 5.2.2.10.

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

Table 6 – Standard NodeId Binary DataEncoding

Name	Data Type	Description
Namespace	UInt16	The *NamespaceIndex*.
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水” is shown in Figure 7.

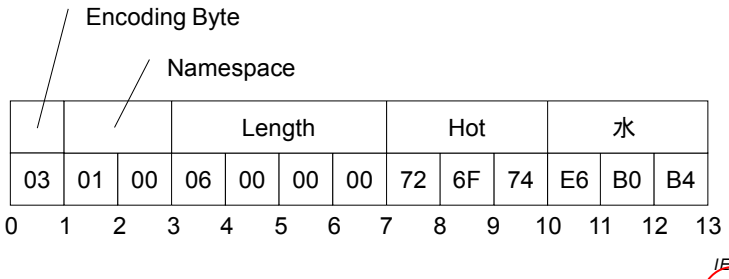

Figure 7 – A String NodeId

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

Table 7 – Two Byte NodeId Binary DataEncoding

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.

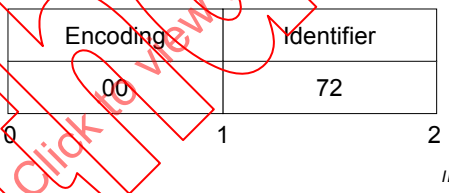

Figure 8 – A Two Byte NodeId

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

Table 8 – Four Byte NodeId Binary DataEncoding

Name	Data Type	Description
Namespace	Byte	The *Namespace* shall be in the range 0 to 255.
Identifier	UInt16	The *Identifier* Type is ‘Numeric’. The *Identifier* shall be an integer in the range 0 to 65 535.

An example of a Four Byte *NodeId* with Namespace = 5 and Identifier = 1 025 is shown in Figure 9.

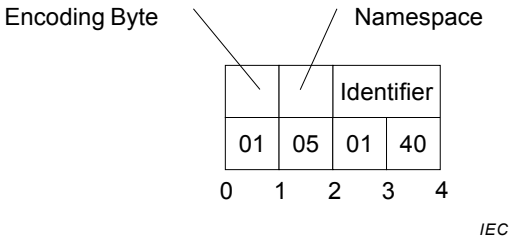

Figure 9 – A Four Byte NodeId

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 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 DataEncoding

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 IEC 62541-4. 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 IEC 62541-4, 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 corresponding string in the string table is encoded. An index of −1 indicates that there is no value for the string.

Table 10 – DiagnosticInfo Binary DataEncoding

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 IEC 62541-3.

Table 11 – QualifiedName Binary DataEncoding

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 IEC 62541-3.

Table 12 – LocalizedText Binary DataEncoding

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 to 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 or treat it as opaque data and skip over it.

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

Table 13 – Extension Object Binary DataEncoding

Name	Data Type	Description
TypeId	NodeId	The identifier for the *DataTypeEncoding* node in the *Server's AddressSpace*. *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 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 be specified if the body is encoded.
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

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

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

Table 14 – Variant Binary DataEncoding

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. For example an array with dimensions [2,2,2] is written in this order: [0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], [1,1,1]
Value	*	The value encoded according to its built-in 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 a *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 types and their identifiers that can be encoded in a *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 DataEncoding

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 picosecond 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 picosecond 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 10 picoseconds and the *Timestamp* field value which only has a 100 ns resolution. The *Picoseconds* fields shall contain values less than 10 000. The decoder shall treat values greater than or equal to 10 000 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 built-in 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 *TypeId* field contains the *DataTypeEncoding* identifier for the *Message*. The *Length* field is omitted since the *Messages* are defined by this series of OPC UA standards.

Each OPC UA *Service* described in IEC 62541-4 has a request and response *Message*. The *DataTypeEncoding* IDs assigned to each *Service* are given in A.3.

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 subtypes 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 *DevelopmentPlatform* have special meaning and shall not be literally encoded in XML.

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

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

The earliest date/time value on a *DevelopmentPlatform* is equivalent to a null date/time value.

5.3.1.7 Guid

A *Guid* is encoded using the string representation defined in 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 (see Base64).

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 an *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 an 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 *NamespaceIndex* 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 *IdentifierType*. 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 *IdentifierType*. 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
ns=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 an xs:string with the syntax:

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

The possible fields are shown in Table 20.

Table 20 – Components of ExpandedNodeId

Field	Data Type	Description
<serverindex>	UInt32	The *ServerIndex* formatted as a base 10 number. If the *ServerIndex* is 0 then the entire 'svr=0;' clause shall be omitted.
<namespaceindex>	UInt16	The *NamespaceIndex* formatted as a base 10 number. If the *NamespaceIndex* is 0 then the entire 'ns=0;' clause shall be omitted. The *NamespaceIndex* shall not be present if the URI is present.
<uri>	String	The *NamespaceUri* 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 *NamespaceUri* is null or empty then 'nsu=;' clause shall be omitted.
<type>	Enum	A flag that specifies the *IdentifierType.* This field is described in Table 19.
<value>	*	The *Identifier* encoded as string. This field is described in Table 19.

The XML schema for an *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 encoded as an *xs:unsignedInt* with the following XML schema:

```
<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 an *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 an *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 an *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 an *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 an 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 an *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 3,141 5 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 IEC 62541-4 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

Array 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 Message SecurityProtocols

6.1 Security handshake

All *SecurityProtocols* shall implement the *OpenSecureChannel* and *CloseSecureChannel* services defined in IEC 62541-4. 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*.

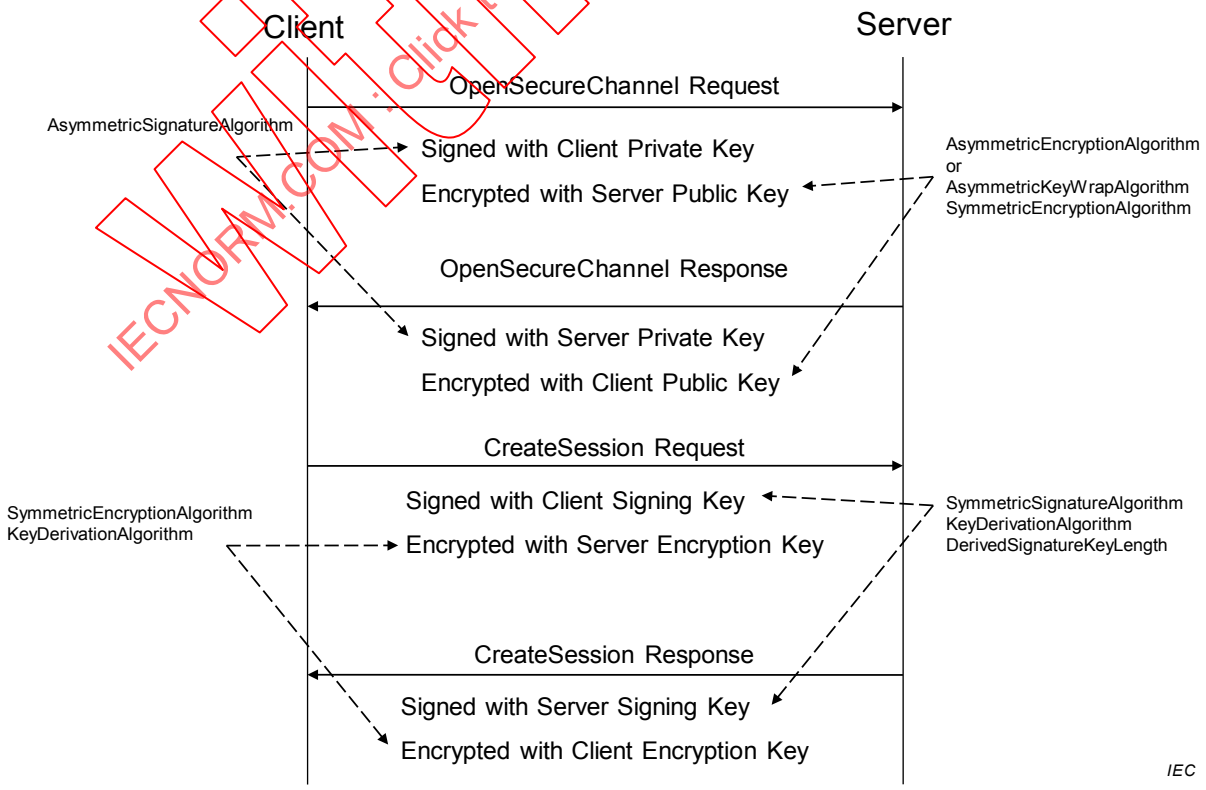

Figure 10 – Security handshake

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*. IEC 62541-7 defines standard *SecurityPolicies* as parts of the standard *Profiles* which OPC UA applications are expected to support. IEC 62541-7 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.
MinAsymmetricKeyLength	The minimum length for an asymmetric key.
MaxAsymmetricKeyLength	The maximum length for an asymmetric key.
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 is implied by the *SymmetricEncryptionAlgorithm*. The length of the keys used for creating *Symmetric Signatures* depends 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 *ApplicationInstance Certificate Data Type*. Subclause 6.2.2 describes how to create an X509 *Certificate* that can be used as an *ApplicationInstance Certificate*.

The *ServerSoftwareCertificates* and *ClientSoftwareCertificates* parameters in the abstract *CreateSession* and *ActivateSession Services* are instances of the *SignedSoftwareCertificate Data Type*. Subclause 6.2.3 describes how to create an X509 *Certificate* that can be used as a *SignedSoftwareCertificate*.

6.2.2 Application Instance Certificate

An *ApplicationInstanceCertificate* is a *ByteString* containing the DER encoded form (see X690) 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 also provides a mapping from the RFC 3280 terms to the terms used in the abstract definition of an *ApplicationInstanceCertificate* defined in IEC 62541-4.

Table 23 – ApplicationInstanceCertificate

Name	Part 4 Parameter Name	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.
authorityKeyIdentifier		Provides more information about the key used to sign the *Certificate*. It shall be specified for *Certificates* signed by a CA. It should be specified for self-signed *Certificates*.

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 be 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. It 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. Subclause 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 Time synchronization

All *Security Protocols* require that system clocks on communicating machines be reasonably synchronized in order to check the expiry times for *Certificates* or *Messages*. The amount of clock skew that can be tolerated depends on the system security requirements and *Applications* shall allow administrators to configure the acceptable clock skew when verifying times. A suitable default value is 5 minutes.

The Network Time Protocol (NTP) provides a standard way to synchronize a machine clock with a time server on the network. Systems running on a machine with a full featured operating system like Windows or Linux will already support NTP or an equivalent. Devices running embedded operating systems should support NTP.

If a device operating system cannot practically support NTP then an OPC UA *Application* can use the *Timestamps* in the *ResponseHeader* (see IEC 62541-4) to synchronize its clock. In this scenario the OPC UA *Application* will have to know the URL for a *Discovery Server* on a machine known to have the correct time. The OPC UA *Application* or a separate background utility would call the *FindServers Service* and set its clock to the time specified in the *ResponseHeader*. This process will need to be repeated periodically because clocks can drift over time.

6.4 UTC and International Atomic Time (TAI)

All times in OPC UA are in UTC, however, UTC can include discontinuities due to leap seconds or repeating seconds added to deal with variations in the earth's orbit and rotation. *Servers* that have access to source for International Atomic Time (TAI) may choose to use this instead of UTC. That said, *Clients* must always be prepared to deal with discontinuities due to the UTC or simply because the system clock is adjusted on the *Server* machine.

6.5 Issued User Identity Tokens – Kerberos

Kerberos *UserIdentityTokens* can be passed to the *Server* using the *IssuedIdentityToken*. The body of the token is an XML element that contains the WS-Security token as defined in the Kerberos Token Profile (Kerberos) specification.

Servers that support Kerberos authentication shall provide a *UserTokenPolicy* which specifies what version of the Kerberos Token Profile is being used, the Kerberos Realm and the Kerberos Principal Name for the *Server*. The Realm and Principal name are combined together with a simple syntax and placed in the *issuerEndpointUri* as shown in Table 25.

Table 25 – Kerberos UserTokenPolicy

Name	Decription
tokenType	ISSUEDTOKEN_3
issuedTypeType	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1
issuerEndpointUri	A string with the form \\<realm>\<server principal name> where <realm> is the Kerberos realm name (e.g. Windows Domain); <server principal name> is the Kerberos principal name for the OPC UA Server.

The interface between the *Client* and *Server* applications and the Kerberos Authentication Service is application specific. The realm is the DomainName when using a Windows Domain controller as the Kerberos provider.

6.6 WS Secure Conversation

6.6.1 Overview

Any *Message* sent via SOAP may be secured with the WS Secure Conversation . 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 in 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 IEC 62541-7. 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. IEC 62541-7 may define *Profiles* that require support for future versions of these specifications.

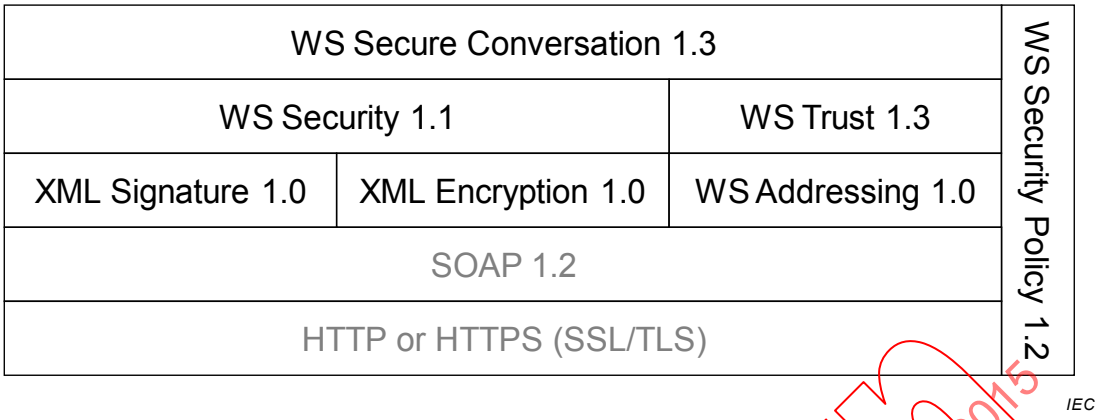

Figure 11 – Relevant XML Web Services specifications

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

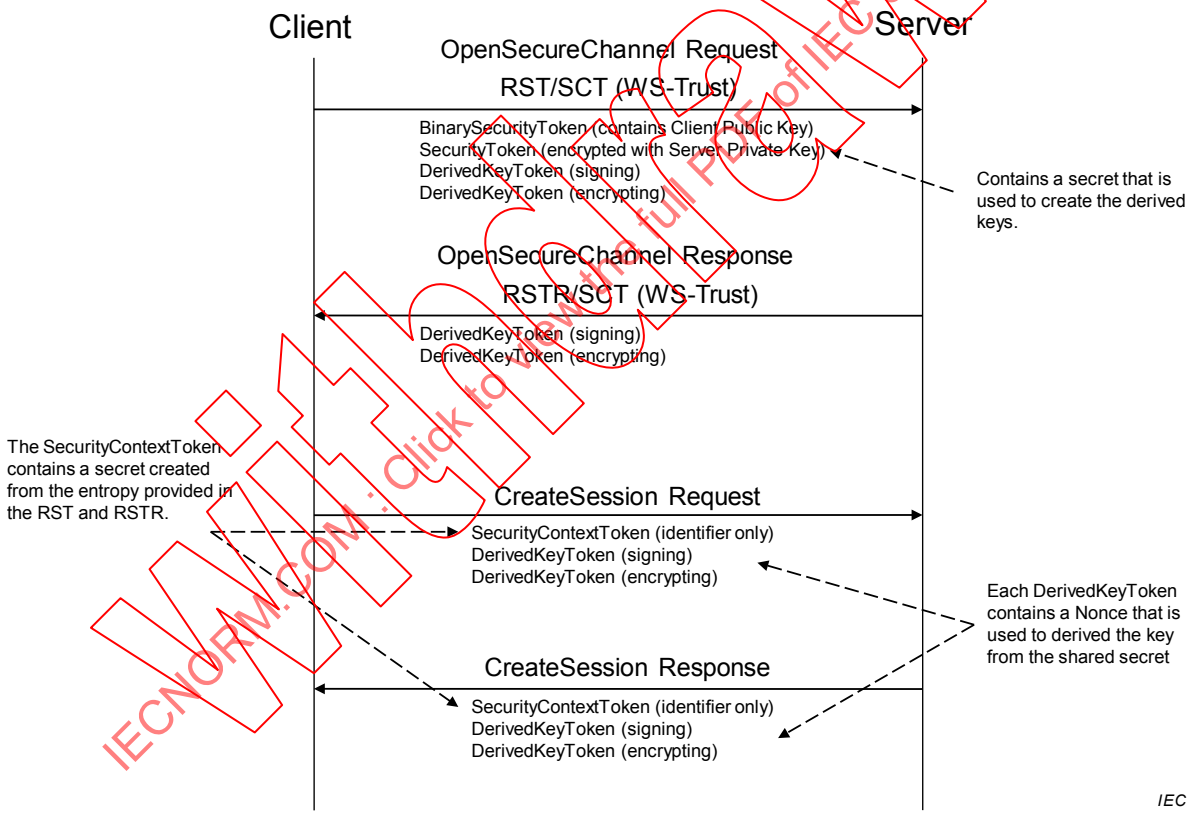

Figure 12 – The WS Secure Conversation handshake

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 subclauses 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.

6.6.2 Notation

SOAP *Messages* use XML elements defined in a number of different specifications. This document uses the prefixes in Table 26 to identify the specification that defines an XML element.

Table 26 – 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

6.6.3 Request Security Token (RST/SCT)

The Request Security Token *Messages* implements the abstract *OpenSecureChannel* request *Message* defined in IEC 62541-4. 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:

a) A wsse:BinarySecurityToken containing the *Client's Public Key*. The *Public Key* is sent in a DER encoded X509v3 *Certificate*.

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

c) 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*.

d) 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 27.

Table 27 – 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.

6.6.4 Request Security Token Response (RSTR/SCT)

The Request Security Token Response *Message* implements the abstract *OpenSecureChannel* response *Message* defined in IEC 62541-4. 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 IEC 62541-4. The symmetric algorithms and a key provided in the request *Message* are used instead.

This *Message* shall have the following tokens:

a) 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*.

b) 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 28.

Table 28 – 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 *SecurityToken* 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 IEC 62541-4.

6.6.5 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:

a) 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*.

b) 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.

6.6.6 Cancelling Security contexts

The *Cancel Message* defined by WS Trust implements the abstract *CloseSecureChannel* request *Message* defined in IEC 62541-4.

This *Message* shall be secured with the SCT.

6.7 OPC UA Secure Conversation

6.7.1 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 8196 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.

6.7.2 MessageChunk structure

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

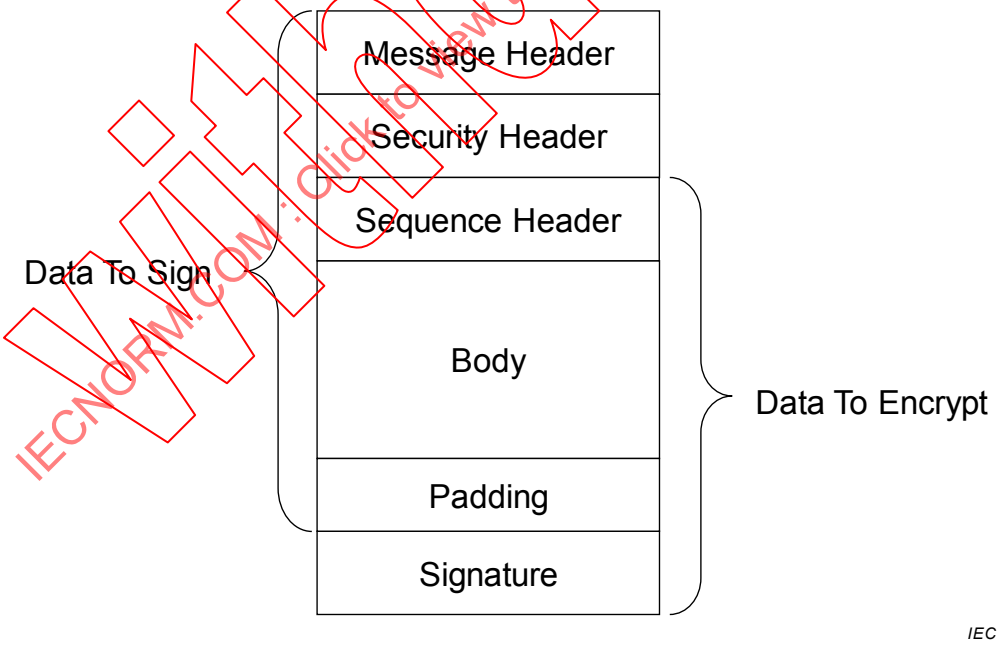

Figure 13 – OPC UA Secure Conversation MessageChunk

Every *MessageChunk* has a *Message* header with the fields defined in Table 29.

Table 29 – 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. When a *Server* starts the first *SecureChannelId* used should be a value that is likely to be unique after each restart. This ensures that a *Server* restart does not cause previously connected *Clients* to accidently 'reuse' *SecureChannels* that did not belong to them.

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 30. Asymmetric algorithms are used to secure the *OpenSecureChannel Messages*. PKCS #1defines a set of 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 30 – 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. If the *Certificate* is signed by a CA the DER encoded CA *Certificate* may be appended after the Certificate in the byte array. If the CA *Certificate* is also signed by another CA this process is repeated until the entire Certificate chain is in the buffer or if *MaxSenderCertificateSize* limit is reached (the process stops after the last whole *Certificate* that can be added without exceeding the *MaxSenderCertificateSize* limit). Receivers can extract the *Certificates* from the byte array by using the *Certificate* size contained in DER header (see X509). Receivers that do not handle *Certificate* chains shall ignore the extra bytes.
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*, including any chains, 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:

```
MaxSenderCertificateSize =
   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
   ExtraPadding –              // ExtraPadding 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 31. Symmetric algorithms are used to secure all *Messages* other than the *OpenSecureChannel Messages*. FIPS 197define symmetric encryption algorithms that UASC implementations may use. FIPS 180-2 and HMAC define some symmetric signature algorithms.

Table 31 – Symmetric algorithm Security header

Name	Data Type	Description
TokenId	UInt32	A unique identifier for the *SecureChannel SecurityToken* 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 32. The sequence header ensures that the first encrypted block of every *Message* sent over a channel will start with different data.

Table 32 – 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 *SecurityToken* 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 4 294 966 271 (UInt32.MaxValue – 1 024). The first number after the wrap around shall be less than 1 024. 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 reestablished. 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 5.2.6. The body may be split across multiple *MessageChunks*.

Each *MessageChunk* also has a footer with the fields defined in Table 33.

Table 33 – 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.
ExtraPaddingSize	Byte	The most significant byte of a two byte integer used to specify the padding size when the key used to encrypt the message chunk is larger than 2048 bits. This field is omitted if the key length is less than or equal to 2048 bits.
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;
```

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 OPC 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.

6.7.3 MessageChunks and error handling

MessageChunks are sent as they are encoded. *MessageChunks* belonging to the same *Message* shall be sent sequentially. If an error occurs creating a *MessageChunk* then the sender shall send a final *MessageChunk* to the receiver that tells the receiver that an error occurred and that it should discard the previous chunks. The sender indicates that the

MessageChunk contains an error by setting the IsFinal flag to 'A' (for Abort). Table 34 specifies the contents of the *Message* abort *MessageChunk*.

Table 34 – 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 41.
Reason	String	A more verbose description of the error. This string shall not be more than 4 096 characters. A *Client* shall ignore strings that are longer than this.

The receiver shall check the security on the abort *MessageChunk* 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*.

6.7.4 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 6.7.2) instead of the body of the *Message*. For this reason, the *OpenSecureChannel Service* is not the same as the one specified in IEC 62541-4. Table 35 lists the parameters that appear in the body of the *Message*.

Table 35 – 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 IEC 62541-4 and are added to the *Message* to allow backward compatibility if OPC UA-SecureConversation 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 OPC UA-*SecureConversation* is used with the OPC UA-TCP protocol (see 7.1) then the version numbers specified in the *OpenSecureChannel Messages* shall be the same as the version numbers specified 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 IEC 62541-4 if there are any errors with the parameters specified by the *Client*.

The *RevisedLifetime* tells the *Client* when it shall renew the *SecurityToken* by sending another *OpenSecureChannel* request. The *Client* shall continue to accept the old *SecurityToken* until it receives the *OpenSecureChannel* response. The *Server* has to accept requests secured with the old *SecurityToken* until that *SecurityToken* expires or until it receives a *Message* from the *Client* secured with the new *SecurityToken*. 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 *SecurityToken* 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 it shall return a *ServiceFault* instead of a *OpenSecureChannel* response. The *ServiceFault Message* is described in IEC 62541-4.

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

6.7.5 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 36.

Table 36 – 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.
```

6.7.6 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 an 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 *SecurityToken* 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 check that the Certificate is trusted first and return *Bad_CertificateUntrusted* on error. The receiver shall then verify the *SenderCertificate* using the rules defined in IEC 62541-4. 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 *SecurityToken* 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 secured error responses 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 TCP transport end-point if communication failures cause temporary TCP session 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 37.

Table 37 – 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 29. 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 38.

Table 38 – 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 *MessageChunk* that the sender can receive. This value shall be greater than 8 192 bytes.
SendBufferSize	UInt32	The largest *MessageChunk* that the sender will send. This value shall be greater than 8 192 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 6.7.3. 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 6.7.3. 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 4 096 bytes. *Servers* shall return a Bad_TcpUrlRejected error and close the connection if the length exceeds 4 096 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 (also known as the 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 39.

Table 39 – 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 *MessageChunk* 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 8 192 bytes.
SendBufferSize	UInt32	The largest *MessageChunk* 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 8 192 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 6.7.3. 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 6.7.3. A value of zero indicates that the *Server* has no limit.

The *Error Message* has the additional fields shown in Table 40.

Table 40 – 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 41.
Reason	String	A more verbose description of the error. This string shall not be more than 4 096 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 6.7) relate to the OPC UA TCP *Messages*.

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

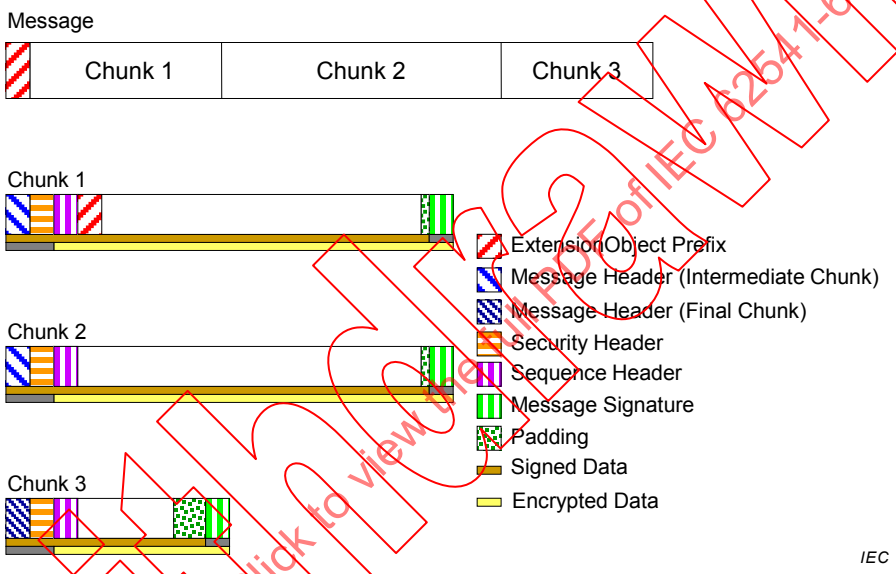

Figure 14 – OPC UA TCP Message structure

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.

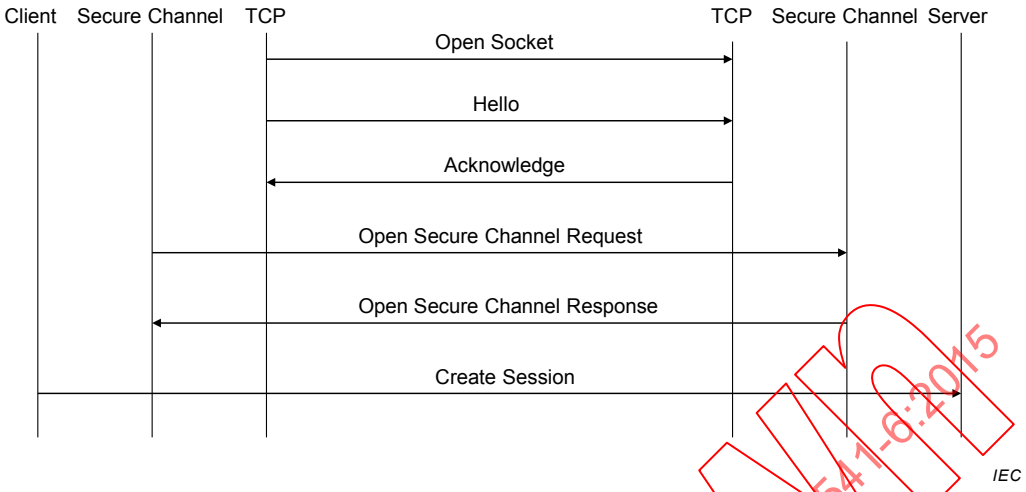

Figure 15 – Establishing a OPC UA TCP connection

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 an OPC UA TCP connection is shown in Figure 16.

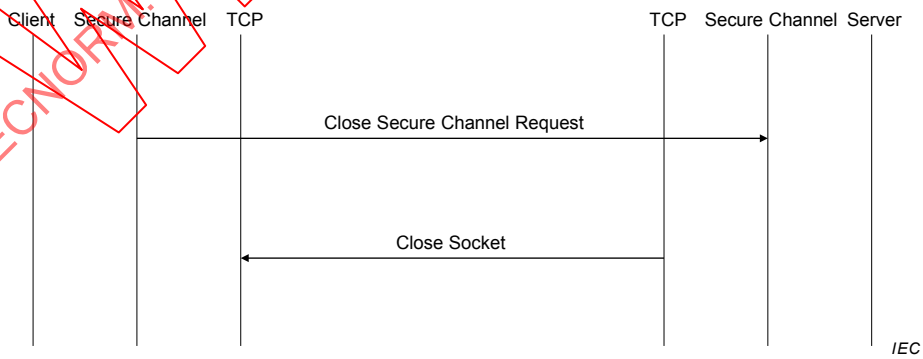

Figure 16 – Closing a OPC UA TCP connection

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 7.1.6.

The possible OPC UA TCP errors are defined in Table 41.

Table 41 – 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 *SecurityToken* 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 an error recovery state whenever the socket breaks or if the *Server* returns an OPC UA TCP Error *Message* as defined in Table 40. 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 6.7.4) 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, close the *SecureChannel* and returns an *Bad_TcpSecureChannelClosed* error to the *Application*.

The sequence of *Messages* when recovering an OPC UA TCP connection is shown in Figure 17.

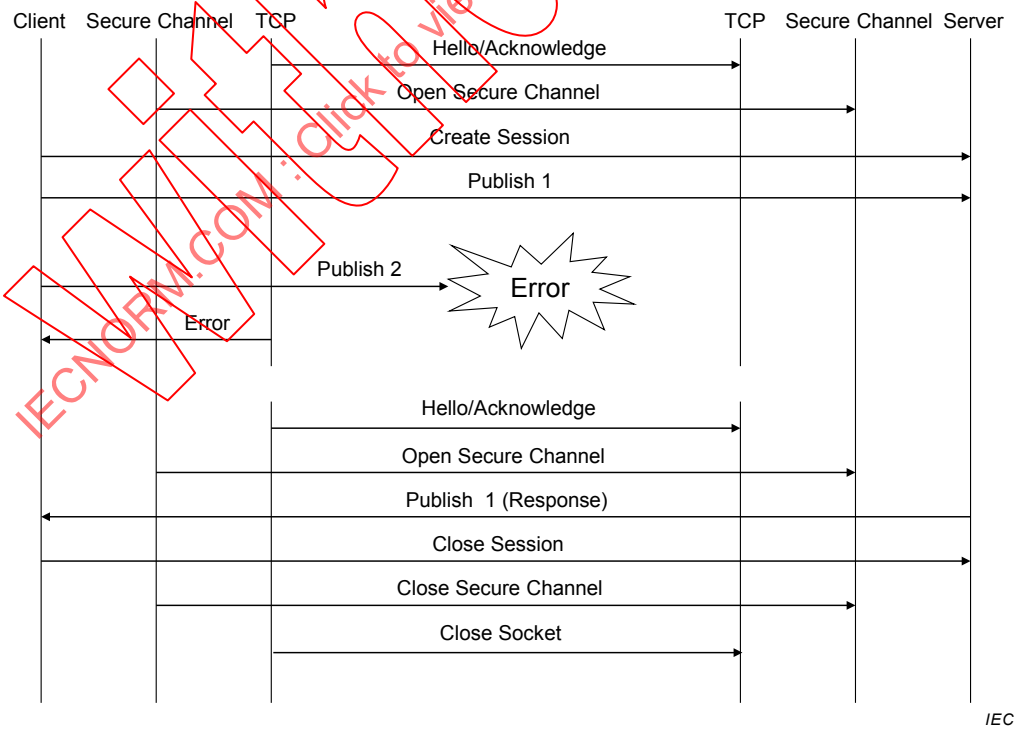

Figure 17 – Recovering an OPC UA TCP connection

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 5 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.

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 IEC 62541-4.

WS Addressing defines standard headers used to route SOAP *Messages* through intermediaries. Implementations shall support the WS-Addressing headers listed Table 42.

Table 42 – 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/2008/02/Services.wsdl/<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/2008/02/Services.wsdl/<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 Base64 data 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 OPC UA Binary encoded request *Message* is:

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

<wsdl:message name="InvokeServiceMessage">
  <wsdl:part name="input" element="s0:InvokeServiceRequest"/>
</wsdl:message>
```

The SOAP Action associated with an OPC UA Binary encoded request *Message* always has the form:

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/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="output" element="s0:InvokeServiceResponse"/>
</wsdl:message>
```

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

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/ InvokeServiceResponse
```

7.3 HTTPS

7.3.1 Overview

HTTPS refers HTTP *Messages* exchanged over a SSL/TLS connection. The syntax of the HTTP *Messages* does not change and the only difference is a TLS connection is created instead of a TCP/IP connection. This implies this that profiles which use this transport can also be used with HTTP when security is not a concern.

HTTPS is a protocol that provides transport security. This means all bytes are secured as they are sent without considering the *Message* boundaries. Transport security can only work for point to point communication and does not allow untrusted intermediaries or proxy servers to handle traffic.

The *SecurityPolicy* shall be specified, however, it only affects the algorithms used for signing the *Nonces* during the *CreateSession*/*ActivateSession* handshake. A *SecurityPolicy* of *None* indicates that the *Nonces* do not need to be signed. The *SecurityMode* is set to *Sign* unless the *SecurityPolicy* is *None*; in this case the *SecurityMode* shall be set to *None*. If a UserIdentityToken is to be encrypted it shall be explicitly specified in the UserTokenPolicy.

An HTTP Header called 'OPCUA-SecurityPolicy' is used by the *Client* to tell the *Server* what *SecurityPolicy* it is using if there are multiple choices available. The value of the header is the URI for the *SecurityPolicy*. If the *Client* omits the header then the *Server* shall assume a *SecurityPolicy* of *None*.

All HTTPS communications via a URL shall be treated as a single *SecureChannel* that is shared by multiple *Clients*. *Stacks* shall provide a unique identifier for the *SecureChannel* which allows *Applications* correlate a request with a *SecureChannel*. This means that *Sessions* can only be considered secure if the *AuthenticationToken* (see IEC 62541-4) is long (>20 bytes) and HTTPS encryption is enabled.

The crypography algorithms used by HTTPS have no relationship to the *EndpointDescription SecurityPolicy* and are determined by the policies set for HTTPS and are outside the scope of OPC UA.

Figure 18 illustrates a few scenarios where the HTTPS transport could be used.

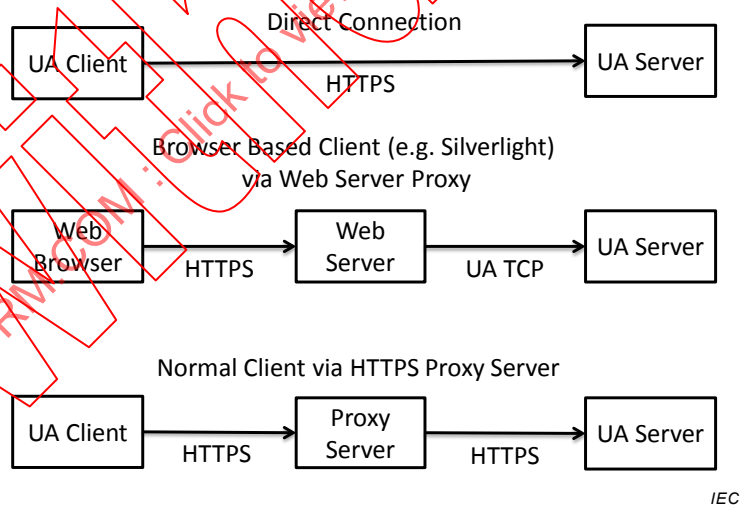

Figure 18 – Scenarios for the HTTPS Transport

In some scenarios, HTTPS communication will rely on an intermediary which is not trusted by the applications. If this is the case then the HTTPS transport cannot be used to ensure security and the applications will have to establish a secure tunnel like a VPN before attempting any OPC UA related communication.

Applications which support the HTTPS transport shall support HTTP 1.1 and SSL/TLS 1.0.

Some HTTPS implementations require that all *Servers* have a *Certificate* with a Common Name (CN) that matches the DNS name of the *Server* machine. This means that a *Server* with multiple DNS names will need multiple HTTPS certificates. If multiple *Servers* are on the

same machine they may share HTTPS certificates. This means that *ApplicationCertificates* are not the same as HTTPS *Certificates*. *Applications* which use the HTTPS transport and require *Application* authentication shall check *Application Certificates* during the CreateSession/ActivateSession handshake.

HTTPS *Certificates* can be automatically generated; however, this will cause problems for *Client*s operating inside a restricted environment such as a web browser. Therefore, HTTPS certificates should be issued by an authority which is accepted by all web browsers which need to access the *Server*. The set of *Certificate* authorities accepted by the web browsers is determined by the organization that manages the *Client* machines. *Client* applications that are not running inside a web may use the trust list that is used for *Application Certificates*.

HTTPS connections have an unpredictable lifetime. Therefore, *Servers* must rely on the *AuthenticationToken* passed in the *RequestHeader* to determine the identity of the *Client*. This means the *AuthenticationToken* shall be a randomly generated value with at least 32 bytes of data and HTTPS with signing and encryption shall always be used.

HTTPS allows *Clients* to have certificates; however, they are not required by the HTTPS transport. A *Server* shall allow *Clients* to connect without an HTTPS *Certificate*.

HTTP 1.1 supports *Message* chunking where the Content-Length header in the request response is set to “chunked” and each chunk is prefixed by its size in bytes. All applications that support the HTTPS transport shall supporting HTTP chunking.

7.3.2 XML Encoding

This *TransportProfile* implements the OPC UA *Services* using a SOAP request-response message pattern over an HTTPS connection.

The body of the HTTP *Messages* shall be a SOAP 1.2 *Message* (see SOAP Part 1). WS-Addressing headers are optional.The contents of the SOAP body and the SOAP action are the same as specified in 7.2.2 and 7.2.3.

All requests shall be HTTP POST requests. The Content-type shall be "application/soap+xml" and the charset and action parameters shall be specified. The charset parameter shall be "utf-8" and the action parameter shall be the URI for the SOAP action.

An example HTTP request header is:

```
POST /UA/SampleServer HTTP/1.1
Content-Type: application/soap+xml; charset="utf-8";
    action="http://opcfoundation.org/UA/2008/02/Services.wsdl/Read"
Content-Length: nnnn
```

The action parameter appears on the same line as the Content-Type declaration.

An example request *Message* (see 7.2.3):

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Body>
    <ReadRequest xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
    …
    </ReadRequest>
  </s:Body>
</s:Envelope>
```

An example HTTP response header is:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8";
```

```
    action="http://opcfoundation.org/UA/2008/02/Services.wsdl/ReadResponse"
Content-Length: nnnn
```

The action parameter appears on the same line as the Content-Type declaration.

An example response *Message*:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Body>
    <ReadResponse xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
    …
    </ReadResponse>
  </s:Body>
</s:Envelope>
```

7.3.3 OPC UA Binary Encoding

This *TransportProfile* implements the OPC UA *Services* using an OPC UA Binary encoded *Messages* exchanged over an HTTPS connection.

Applications which support the HTTPS *Profile* shall support HTTP 1.1.

The body of the HTTP *Messages* shall be OPC UA Binary encoded blob. The Content-type shall be "application/octet-stream".

An example HTTP request header is:

```
POST /UA/SampleServer HTTP/1.1
Content-Type: application/octet-stream;
Content-Length: nnnn
```

An example HTTP response header is:

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream;
Content-Length: nnnn
```

The *Message* body is the request or response structure encoded as an *ExtensionObject* in OPC UA Binary.

7.4 Well known addresses

The *Local Discovery Server* (LDS) is an OPC UA *Server* that implements the *Discovery Service Set* defined in IEC 62541-4. If an LDS is installed on a machine it shall use one or more of the well-known addresses defined in Table 43.

Table 43 – 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.
OPC UA TCP	opc.tcp://localhost:4840/UADiscovery	
OPC UA HTTPS	https:// localhost:4843/UADiscovery	

OPC 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). OPC UA *Servers* shall allow administrators to configure the address to use for registration.

Each OPC UA *Server Application* implements the *Discovery Service Set*. If the OPC 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 IEC 62541-3. 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 D.

Annex A
(normative)

Constants

A.1 Attribute Ids

Table A.1 – 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 annex 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>, <Code>, <Description>
```

Where the *SymbolName* is the literal name for the error code that appears in the specification and the *Code* is the hexadecimal value for the *StatusCode* (see IEC 62541-4). The severity associated with a particular code is specified by the prefix (*Good*, *Uncertain* or *Bad*).

The CSV released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/StatusCode.csv

NOTE The latest CSV that is compatible with this version of the standard can be found here:
http://www.opcfoundation.org/UA/schemas/StatusCode.csv

A.3 Numeric Node Ids

This annex 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, IEC 62541-5 defines the *ServerType ObjectType Node* which has the *NamespaceArray Property*. The *SymbolName* for the NamespaceArray *InstanceDeclaration* within the *ServerType* declaration is: *ServerType_NamespaceArray*. IEC 62541-5 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 released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/NodeIds.csv

NOTE The latest CSV that is compatible with this version of the standard can be found here:
http://www.opcfoundation.org/UA/schemas/NodeIds.csv

Annex B
(normative)

OPC UA Nodeset

The OPC UA NodeSet includes the complete Information Model defined in this standard. It follows the XML Information Model schema syntax defined in Annex F and can thus be read and processed by a computer program.

The Information Model Schema released with this version of the standard can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.NodeSet2.xml

NOTE The latest Information Model schema that is compatible with this version of the standard can be found here:

http://www.opcfoundation.org/UA/schemas/Opc.Ua.NodeSet2.xml

Annex C
(normative)

Type declarations for the OPC UA native Mapping

This Annex defines the OPC UA Binary encoding for all *DataTypes* and *Messages* defined in this standard. The schema used to describe the type is defined in IEC 62541-3.

The OPC UA Binary Schema released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Types.bsd.xml

NOTE The latest file that is compatible with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/Opc.Ua.Types.bsd.xml

Annex D
(normative)

WSDL for the XML Mapping

D.1 XML Schema

This annex defines the XML Schema for all DataTypes and *Messages* defined in this series of OPC UA standards.

The XML Schema released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Types.xsd

NOTE The latest file that is compatible with this version of the standards can be found here:
http://www.opcfoundation.org/UA/2008/02/Types.xsd

D.2 WDSL Port Types

This annex defines the WSDL Operations and Port Types for all Services defined in IEC 62541-4.

The WSDL released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Services.wsdl

NOTE The latest file that is compatible with this version of the standards can be found here:
http://opcfoundation.org/UA/2008/02/Services.wsdl

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

D.3 WSDL Bindings

This annex defines the WSDL Bindings for all Services defined in IEC 62541-4.

The WSDL released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Endpoints.wsdl

NOTE The latest file that is compatible with this version of the standards can be found here:
http://opcfoundation.org/UA/2008/02/Endpoints.wsdl

This WSDL imports the WSDL defined in D.2.

Annex E
(normative)

Security settings management

E.1 Overview

All OPC UA applications shall support security; however, this requirement means that Administrators need to configure the security settings for the OPC UA *Application*. This appendix describes an XML Schema which can be used to read and update the security settings for a OPC UA *Application*. All OPC UA applications may support configuration by importing/exporting documents that conform to the schema (called the *SecuredApplication* schema) defined in this Annex.

The XML Schema released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/SecuredApplication.xsd

NOTE The latest file that is compatible with this version of this specification can be found here:
http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd

The *SecuredApplication* schema can be supported in two ways:

1) Providing an XML configuration file that can be edited directly;
2) Providing a import/export utility that can be run as required;

If the *Application* supports direct editing of an XML configuration file then that file shall have exactly one element with the local name 'SecuredApplication' and URI equal to the *SecuredApplication* schema URI. A third party configuration utility shall be able to parse the XML file, read and update the 'SecuredApplication' element. The administrator shall ensure that only authorized administrators can update this file. The following is an example of a configuration that can be directly edited:

```
<s1:SampleConfiguration xmlns:s1="http://acme.com/UA/Sample/Configuration.xsd">
  <ApplicationName>ACME UA Server</ApplicationName>
  <ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>

  <!-- any number of application specific elements -->

  <SecuredApplication xmlns="http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd">
    <ApplicationName>ACME UA Server</ApplicationName>
    <ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>
    <ApplicationType>Server_0</ApplicationType>
    <ApplicationCertificate>
      <StoreType>Windows</StoreType>
      <StorePath>LocalMachine\My</StorePath>
      <SubjectName>ACME UA Server</SubjectName>
    </ApplicationCertificate>
  </SecuredApplication>

  <!-- any number of application specific elements -->

  <DisableHiResClock>true</DisableHiResClock>
</s1:SampleConfiguration>
```

If an *Application* provides an import/export utility then the import/export file shall be a document that conforms to the *SecuredApplication* schema. The administrator shall ensure that only authorized administrators can run the utility. The following is an example of a file used by an import/export utility:

```
<?xml version="1.0" encoding="utf-8" ?>
<SecuredApplication xmlns="http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd">
  <ApplicationName>ACME UA Server</ApplicationName>
```

```
  <ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>
  <ApplicationType>Server_0</ApplicationType>
  <ConfigurationMode>urn:acme.com:ACME Configuration Tool</ConfigurationMode>
  <LastExportTime>2011-03-04T13:34:12Z</LastExportTime>
  <ExecutableFile>%ProgramFiles%\ACME\Bin\ACME UA Server.exe</ExecutableFile>
  <ApplicationCertificate>
    <StoreType>Windows</StoreType>
    <StorePath>LocalMachine\My</StorePath>
    <SubjectName>ACME UA Server</SubjectName>
  </ApplicationCertificate>
  <TrustedCertificateStore>
    <StoreType>Windows</StoreType>
    <StorePath>LocalMachine\UA Applications</StorePath>
    <!-- Offline CRL Checks by Default -->
    <ValidationOptions>16</ValidationOptions>
  </TrustedCertificateStore>
  <TrustedCertificates>
    <Certificates>
      <CertificateIdentifier>
        <SubjectName>CN=MyFactory CA</SubjectName>
        <!--  Online CRL Check for this CA -->
        <ValidationOptions>32</ValidationOptions>
      </CertificateIdentifier>
    </Certificates>
  </TrustedCertificates>
  <RejectedCertificatesStore>
    <StoreType>Directory</StoreType>
    <StorePath>%CommonApplicationData%\OPC Foundation\RejectedCertificates</StorePath>
  </RejectedCertificatesStore>
</SecuredApplication>
```

E.2 SecuredApplication

The *SecuredApplication* element specifies the security settings for an *Application*. The elements contained in a SecuredApplication are described in Table E.1.

When an 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.

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 physical stores. Each *Application* needs to decide whether it uses shared physical stores which the administrator can control directly by changing the location or private stores that can only be accessed via the import/export utility. If the *Application* uses private stores then the contents of these private stores shall be copied to the export file during export. If the import file references shared physical stores then the import/export utility shall copy the contents of those stores to the private stores.

The import/export utility shall not export private keys. If the administrator wishes to assign a new public-private key to the *Application* the administrator shall place the private in a store where it can be accessed by the import/export utility. The import/export utility is then responsible for ensuring it is securely moved to a location where the *Application* can access it.

Table E.1 – SecuredApplication

Element	Type	Description
ApplicationName	String	A human readable name for the *Application*. Applications shall allow this value to be read or changed.
ApplicationUri	String	A globally unique identifier for the instance of the *Application*. Applications shall allow this value to be read or changed.
ApplicationType	ApplicationType	The type of *Application*. May be one of • Server_0; • Client_1; • ClientAndServer_2; • DiscoveryServer_3; *Application* shall provide this value. Applications do not allow this value to be changed.
ProductName	String	A name for the product. *Application* shall provide this value. Applications do not allow this value to be changed.
ConfigurationMode	String	Indicates how the *Application* should be configured. An empty or missing value indicates that the configuration file can be edited directly. The location of the configuration file shall be provided in this case. Any other value is a URI that identifies the configuration utility. The vendor documentation shall explain how to use this utility. *Application* shall provide this value. Applications do not allow this value to be changed.
LastExportTime	UtcTime	When the configuration was exported by the import/export utility. It may be omitted if Applications allow direct editing of the security configuration.
ConfigurationFile	String	The full path to a configuration file used by the *Application*. Applications do nor provide this value if a import/export utility is used. 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 an executable file for 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	CertificateIdentifier	The identifier for the *ApplicationInstance Certificate*. Applications shall allow this value to be read or changed. This identifier may reference a *Certificate* store that contains the private key. If the private key is not accessible to outside applications this value shall contain the X509 *Certificate* for the *Application*. If the configuration utility assigns a new private key this value shall reference the store where the private key is placed. The import/export utility may delete this private key if it moves it to a secure location 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.

Element	Type	Description
TrustedCertificateStore	CertificateStore Identifier	The location of the CertificateStore containing the Certificates of Applications or *Certificate* Authorities (CAs) 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 separately from the *Application*. Applications that support shared physical stores 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* may trust Certificates in this store even if they cannot be verified back to a trusted root. Administrators shall place any CA certificates used to verify the signature in the UntrustedIssuerStore or the UntrustedIssuerList. This will allow applications to properly verify the signatures. The *Application* shall check the revocation status of the Certificates in this store if the *Certificate* was issued by a CA. The *Application* shall look for the offline *Certificate* Recovation List (CRL) for a CA in the store where it found the CA *Certificate*. The location of an online CRL for CA shall be specified with the CRLDistributionPoints (OID= 2.5.29.31) X509 *Certificate* extension. The ValidationOptions parameter is used to specify which revocation list should be used for CAs in this store.
TrustedCertificates	CertificateList	A list of Certificates for Applications for CAs that can be trusted. Applications shall allow this value to be read or changed. The value is an explicit list of Certificates which is private to the *Application*. It is used when the *Application* does not support shared physical *Certificate* stores or when Administrators need to specify ValidationOptions for individual Certificates. If the TrustedCertificateStore and the TrustedCertificates parameters are both specified then the *Application* shall use the TrustedCertificateStore for checking trust relationships. The TrustedCertificates parameter is only used to lookup ValidationOptions for individual Certificates. It may also be used to provide CRLs for CA certificates. If the TrustedCertificateStore is not specified then TrustedCertificates parameter shall contain the complete X509 *Certificate* for each entry.
IssuerStore	CertificateStore Identifier	The location of the CertificateStore containing CA Certificates which are not trusted but are needed to check signatures on Certificates. Applications shall allow this value to be read or changed. This value shall be a reference to a physical store which can be managed separately from the *Application*. Applications that support shared physical stores shall check this store for changes whenever they validate a *Certificate*. This store may also contain CRLs for the CAs.
IssuerCertificates	CertificateList	A list of Certificates for CAs which are not trusted but are needed to check signatures on Certificates. Applications shall allow this value to be read or changed. The value is an explicit list of Certificates which is private to the *Application*. It is used when the *Application* does not support shared physical *Certificate* stores or when Administrators need to specify ValidationOptions for individual Certificates. If the IssuerStore and the IssuerCertificates parameters are both specified then the *Application* shall use the IssuerStore for checking signatures. The IssuerCertificates parameter is only used to lookup ValidationOptions for individual Certificates. It may also be used to provide CRLs for CA certificates.
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 E.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).

Element	Type	Description
BaseAddresses	String[]	A list of URLs for the *Endpoint*s supported by a *Server*. Applications shall allow these values to be read or changed. If a *Server* does not support the scheme for a URL it shall ignore it. This list can have multiple entries for the same URL scheme. The first entry for a scheme is the base URL. The rest are assumed to be DNS aliases that point to the first URL. It is the responsibility of the Administrator to configure the network to route these aliases correctly.
SecurityProfileUris	SecurityProfile[] ProfileUri String Enabled Boolean	A list of security profiles supported by a *Server*. *Applications* shall allow these values to be read or changed. *Applications* shall allow the Enabled flag to be changed for each *SecurityProfile* that it supports. If the Enabled flag is false the *Server* shall not allow connections using the *SecurityProfile*. If a *Server* does not support a SecurityProfile it shall ignore it.
Extensions	xs:any	A list of vendor defined Extensions attached to the security settings. Applications shall ignore Extensions that they do not recognize. Applications that update a file containing Extensions shall not delete or modify extensions that they do not recognize.

E.3 CertificateIdentifier

The *CertificateIdentifier* element describes an 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 E.2.

Table E.2 – CertificateIdentifier

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the *Certificate*. Predefined values are "Windows" and "Directory". If not specified the RawData element shall be specified.
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType. If not specified the RawData element shall be specified.
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. Import utilities shall reject configurations containing invalid Certificates. This field shall not be specified if the StoreType and StorePath are specified.
ValidationOptions	Int32	The options to use when validating the *Certificate*.The possible options are described in E.6.
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 an 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.

If a "Directory" store contains a 'certs' subdirectory then it is presumed to be a structured store with the subdirectories described in Table E.3.

Table E.3 – Structured directory store

Subdirectory	Description
certs	Contains the DER encoded X509 Certificates. The files shall have a .der file extension.
private	Contains the private keys. The format of the file may be *Application* specific. PEM encoded files should have a .pem extension. PKCS#12 encoded files should have a .pfx extension. The root file name shall be the same as the corresponding public key file in the certs directory.
crl	Contains the DER encoded CRL for any CA Certificates found in the certs or ca directories. The files shall have a .crl file extension.

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 *Certificate* 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.

E.4 CertificateStoreIdentifier

The *CertificateStoreIdentifier* element describes a physical store containing X509 Certificates. The elements contained in a *CertificateStoreIdentifier* are described in Table E.4.

Table E.4 – CertificateStoreIdentfier

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the *Certificate*. Predefined values are "Windows" and "Directory".
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType. See E.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 E.6.

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.

E.5 CertificateList

The *CertificateList* element is a list of *Certificates*. The elements contained in a *CertificateList* are described in Table E.5.

Table E.5 – CertificateList

Element	Type	Description
Certficates	CertificateIdentifier[]	The list of Certificates contained in the Trust List
ValidationOptions	Int32	The options to use when validating the Certificates contained in the store. These options only apply to *Certificates* that have *ValidationOptions* with the *UseDefaultOptions* bit set. The possible options are described in E.6.

E.6 CertificateValidationOptions

The *CertificateValidationOptions* control the process used to validate a *Certificate*. Any *Certificate* can have validation options associated. If none are specified the *ValidationOptions* for the store or list containing the *Certificate* are used. The possible options are shown in Table E.6.

Table E.6 – CertificateValidationOptions

Field	Bit	Description
SuppressCertificateExpired	0	Ignore errors related to the validity time of the *Certificate* or its issuers.
SuppressHostNameInvalid	1	Ignore mismatches between the host name or *Application* uri.
SuppressRevocationStatusUnknown	2	Ignore errors if the issuer's revocation list cannot be found.
CheckRevocationStatusOnline	3	Check the revocation status online. If set the validator will look for the URL of the CRL Distribution Point in the *Certificate* and use the OCSP (Online Certificate Status Protocol) to determine if the *Certificate* has been revoked. If the CRL Distribution Point is not reachable then the validator will look for offline CRLs if the *CheckRevocationStatusOffine* bit is set. Otherwise, validation fails. This option is specified for Issuer *Certificates* and used when validating Certificates issued by that Issuer.
CheckRevocationStatusOffine	4	Check the revocation status offline. If set the validator will look a CRL in the *Certificate Store* where the CA *Certificate* was found. Valididation fails if a CRL is not found. This option is specified for Issuer *Certificates* and used when validating *Certificates* issued by that *Issuer*.
UseDefaultOptions	5	If set the *CertificateValidationOptions* from the *CertificateList* shall be used. If a *Certificate* does not belong to a *CertificateList* then the default is 0 for all bits.

Annex F
(normative)

Information Model XML Schema

F.1 Overview

Information Model developers define standard *AddressSpaces* which are implemented by many *Servers*. There is a need for a standard syntax that Information Model developers can use to formally define their models in a form that can be read by a computer program. This Annex defines an XML-based schema for this purpose.

The XML Schema released with this version of the standards can be found here:

http://www.opcfoundation.org/UA/schemas/1.02/UANodeSet.xsd

NOTE The latest file that is compatible with this version of the standards can be found here:
http://opcfoundation.org/UA/2011/03/UANodeSet.xsd

The schema document is the formal definition. The description in this Annex only discusses details of the semantics that cannot be captured in the schema document. Types which are self-describing are not discussed.

This schema can also be used to serialize (i.e. import or export) an arbitrary set of *Nodes* in the *Server Address Space*. This serialized form can be used to save *Server* state for use by the *Server* later or to exchange with other applications (e.g. to support offline configuration by a Client).

F.2 UANodeSet

The *UANodeSet* is the root of the document. It defines a set of *Nodes*, their *Attributes* and *References*. *References* to *Nodes* outside of the document are allowed.

The structure of a UANodeSet is shown in Table F.1.

Table F.1 – UANodeSet

Element	Type	Description
NamespaceUris	UriTable	A list of *NamespaceUris* used in the *UANodeSet*.
ServerUris	UriTable	A list of *ServerUris* used in the *UANodeSet*.
Aliases	AliasTable	A list of *Aliases* used in the *UANodeSet*.
Extensions	xs:any	An element containing any vendor defined extensions to the *UANodeSet*.
<choice>	UAObject UAVariable UAMethod UAView UAObjectType UAVariableType UADataType UAReferenceType	The *Nodes* in the *UANodeSet*.

The *NamespaceUri* is a list of URIs for namespaces used in the *UANodeSet*. The *NamespaceIndexes* used in *NodeId, ExpandedNodeIds* and *QualifiedNames* identify an element in this list. The first index is always 1 (0 is always the OPC UA namespace).

The *ServerUris* is a list of URIs for *Servers* referenced in the *UANodeSet*. The *ServerIndex* in *ExpandedNodeIds* identifies an element in this list. The first index is always 1 (0 is always the current *Server*).

The *Aliases* are a list of string substitutions for *NodeIds*. *Aliases* can be used to make the file more readable by allowing a string like 'HasProperty' in place of a numeric NodeId (i=46). *Aliases* are optional.

The *Extensions* are free form XML data that can be used to attach vendor defined data to the *UANodeSet*.

F.3 UANode

A *UANode* is an abstract base type for all *Nodes*. It defines the base set of *Attributes* and the *References*. There are subtypes for each *NodeClass* defined in IEC 62541-4. Each of these subtypes defines XML elements and attributes for the OPC UA *Attributes* specific to the *NodeClass*. The fields in the *UANode* type are defined in Table F.2.

Table F.2 – UANode

Element	Type	Description
NodeId	NodeId	A *NodeId* serialized as a *String*. The syntax of the serialized *String* is defined in 5.3.1.10.
BrowseName	QualifiedName	A *QualifiedName* serialized as a *String* with the form: <namespace index>:<name> Where the *NamespaceIndex* refers to the *NamespaceUris* table.
SymbolicName	String	A symbolic name for the *Node* that can be used as a class/field name in autogenerated code. It should only be specified if the *BrowseName* cannot be used for this purpose. This field does not appear in the *AddressSpace* and is intended for use by design tools. Only letters, digits or the underscore ('_') are permitted.
WriteMask	WriteMask	The value of the *WriteMask* Attribute.
UserWriteMask	WriteMask	The value of the *UserWriteMask* Attribute.
DisplayName	LocalizedText[]	A list of *DisplayNames* for the *Node* in different locales. There shall be only one entry per locale.
Description	LocalizedText[]	The list of the *Descriptions* for the *Node* in different locales. There shall be only one entry per locale.
References	Reference[]	The list of *References* for the *Node*.
Extensions	xs:any	An element containing any vendor defined extensions to the *UANode*.

The *Extensions* are free form XML data that can be used to attach vendor defined data to the *UANode*.

F.4 Reference

The *Reference* type specifies a *Reference* for a *Node*. The *Reference* can be forward or inverse. Only one direction for each *Reference* needs to be in a *UANodeSet*. The other direction shall be added automatically during any import operation. The fields in the *Reference* type are defined in Table F.3.

Table F.3 – Reference

Element	Type	Description
NodeId	NodeId	The *NodeId* of the target of the *Reference* serialized as a *String*. The syntax of the serialized *String* is defined in 5.3.1.11 (*ExpandedNodeId*). This value can be replaced by an *Alias*.
ReferenceType	NodeId	The NodeId of the *ReferenceType* serialized as a *String*. The syntax of the serialized *String* is defined in 5.3.1.10 (*NodeId*). This value can be replaced by an *Alias*.
IsForward	Boolean	If TRUE the *Reference* is a forward reference.

F.5 UAType

A *UAType* is a subtype of the *UANode* defined in F.3. It is the base type for the types defined in Table F.4.

Table F.4 – UANodeSet Type Nodes

Subtype	Description
UAObjectType	Defines an *ObjectType Node* as described in IEC 62541-3.
UAVariableType	Defines a *VariableType Node* as described in IEC 62541-3.
UADataType	Defines a *DataType Node* as described in IEC 62541-3.
UAReferenceType	Defines a *ReferenceType Node* as described in IEC 62541-3.

F.6 UAInstance

A *UAInstance* is a subtype of the *UANode* defined in F.3. It is the base type for the types defined in Table F.5. The fields in the *UAInstance* type are defined in Table F.6.

Table F.5 – UANodeSet Instance Nodes

Subtype	Description
UAObject	Defines an *Object Node* as described in IEC 62541-3.
UAVariable	Defines a *Variable Node* as described in IEC 62541-3.
UAMethod	Defines a *Method Node* as described in IEC 62541-3.
UAView	Defines a *View Node* as described in IEC 62541-3.

Table F.6 – UAInstance

Element	Type	Description
All of the fields from the *UANode* type described in F.3.		
ParentNodeId	NodeId	The *NodeId* of the *Node* that is the parent of the *Node* within the information model. This field is used to indicate that a tight coupling exists between the *Node* and its parent (e.g. when the parent is deleted the child is deleted as well). This information does not appear in the *AddressSpace* and is intended for use by design tools.

F.7 UAVariable

A *UAVariable* is a subtype of the *UAInstance* defined in F.6. It represents a Variable Node. The fields in the *UAVariable* type are defined in Table F.7.

Table F.7 – UAVariable

Element	Type	Description
All of the fields from the *UAInstance* type described in 0.		
Value	Variant	The Value of the Node encoding using the UA XML wire encoding.
Translation	TranslationType[]	A list of translations for the Value if the Value is a LocalizedText or a structure containing LocalizedTexts. This field may be omitted. If the Value is an array the number of elements in this array shall match the number of elements in the Value. Extra elements are ignored. If the Value is a scalar then there is one element in this array. If the Value is a structure then the each element contains translations for one or more fields identified by a name. See the TranslationType for more information.
DataType	NodeId	The data type of the value.
ValueRank	ValueRank	The value rank.
ArrayDimensions	ArrayDimensions	The number of dimensions in an array value.
AccessLevel	AccessLevel	The access level.
UserAccessLevel	AccessLevel	The access level for the current user.
MinimumSamplingInterval	Duration	The minimum sampling interval.
Historizing	Boolean	Whether history is being archived.

F.8 UAMethod

A *UAMethod* is a subtype of the *UAInstance* defined in 0. It represents a Method Node. The fields in the *UAMethod* type are defined in Table F.8.

Table F.8 – UAMethod

Element	Type	Description
All of the fields from the *UAInstance* type described in 0.		
MethodDeclarationId	NodeId	May be specified for *Method Nodes* that are a target of a *HasComponent* reference from a single *Object Node*. It is the *NodeId* of the *UAMethod* with the same *BrowseName* contained in the *TypeDefinition* associated with the *Object Node*. If the *TypeDefinition* overrides a *Method* inherited from a base *ObjectType* then this attribute shall reference the *Method Node* in the subtype.

F.9 TranslationType

A *TranslationType* contains additional translations for *LocalizedTexts* used in the *Value* of a *Variable*. The fields in the *TranslationType* are defined in Table F.9. If multiple *Arguments* existed there would be a Translation element for each *Argument*.

The type can have two forms depending on whether the *Value* is a *LocalizedText* or a *Structure* containing *LocalizedTexts*. If it is a *LocalizedText* is contains a simple list of translations. If it is a *Structure* it contains a list of fields which each contain a list of translations. Each field is identified by a Name which is unique within the structure. The mapping between the Name and the *Structure* requires an understanding of the *Structure* encoding. If the *Structure* field is encoded as a *LocalizedText* with UA XML then the name is the unqualified path to the XML element where names in the path are separated by '/'. For example, a structure with a nested structure containing a LocalizedText could have a path like "Server/ApplicationName".

The following example illustrates how translations for the Description field in the *Argument Structure* are represented in XML:

```
<Value>
  <ListOfExtensionObject xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
```

```
    <ExtensionObject>
      <TypeId>
        <Identifier>i=297</Identifier>
      </TypeId>
      <Body>
        <Argument>
          <Name>ConfigData</Name>
          <DataType>
            <Identifier>i=15</Identifier>
          </DataType>
          <ValueRank>-1</ValueRank>
          <ArrayDimensions />
          <Description>
            <Text>[English Translation for Description]</Text>
          </Description>
        </Argument>
      </Body>
    </ExtensionObject>
  </ListOfExtensionObject>
</Value>
<Translation>
  <Field Name="Description">
    <Text Locale="de-DE">[German Translation for Description]</Text>
    <Text Locale="fr-FR">[French Translation for Description]</Text>
  </Field>
</Translation>
```

If multiple Arguments existed there would be a Translation element for each Argument.

Table F.9 – TranslationType

Element	Type	Description
Text	LocalizedText[]	An array of translations for the Value. It only appears if the *Value* is a *LocalizedText* or an array of *LocalizedText*.
Field	StructureTranslationType[]	An array of structure fields which have translations. It only appears if the Value is a *Structure* or an array of *Structures*.
Name	String	The name of the field. This uniquely identifies the field within the structure. The exact mapping depends on the encoding of the structure.
Text	LocalizedText[]	An array of translations for the structure field.

F.10 UADataType

A *UADataType* is a subtype of the *UAType* defined in 0. It defines a *DataType Node*. The fields in the *UADataType* type are defined in Table F.10.

Table F.10 – UADataType

Element	Type	Description
All of the fields from the *UANode* type described in F.3.		
Definition	DataTypeDefinition	An abstract definition of the data type that can be used by design tools to create code that can serialize the data type in XML and/or Binary forms. It does not appear in the *AddressSpace*. This is only used to define subtypes of the *Structure* or *Enumeration DataTypes*.

F.11 DataTypeDefinition

A *DataTypeDefinition* defines an abstract representation of a *UADataType* that can be used by design tools to automatically create serialization code. The fields in the *DataTypeDefinition* type are defined in Table F.11.

Table F.11 – DataTypeDefinition

Element	Type	Description
Name	QualifiedName	A unique name for the data type. This field is only specified for nested *DataTypeDefinitions*. The *BrowseName* of the *DataType Node* is used otherwise.
SymbolicName	String	A symbolic name for the data type that can be used as a class/structure name in autogenerated code. It should only be specified if the *Name* cannot be used for this purpose. Only letters, digits or the underscore ('_') are permitted. This field is only specified for nested *DataTypeDefinitions*. The *SymbolicName* of the *DataType Node* is used otherwise.
BaseType	QualifiedName	The name of any base type. Note that the BaseType can refer to types defined in other files. The NamespaceUri associated with the Name should indicate where to look for the BaseType definition. This field is only specified for nested *DataTypeDefinitions*. The *HasSubtype Reference* of the *DataType Node* is used otherwise.
Fields	DataTypeField[]	The list of fields that make up the data type. This definition assumes the structure has a sequential layout. For enumerations the fields are simply a list of values. Unions are not supported.

F.12 DataTypeField

A *DataTypeField* defines an abstract representation of a field within a *UADataType* that can be used by design tools to automatically create serialization code. The fields in the *DataTypeField* type are defined in Table F.12.

Table F.12 – DataTypeField

Element	Type	Description
Name	String	A name for the field that is unique within the *DataTypeDefinition*.
SymbolicName	String	A symbolic name for the field that can be used in autogenerated code. It should only be specified if the *Name* cannot be used for this purpose. Only letters, digits or the underscore ('_') are permitted.
DataType	NodeId	The *NodeId* of the *DataType* for the field. This *NodeId* can refer to another *Node* with its own *DataTypeDefinition*. This field is not specified for subtypes of *Enumeration*.
ValueRank	Int32	The value rank for the field. It shall be *Scalar* (-1) or a fixed rank *Array* (>=1). This field is not specified for subtypes of *Enumeration*.
Description	LocalizedText[]	A description for the field in multiple locales.
Definition	DataTypeDefinition	The field is a structure with a layout specified by the definition. This field is optional. This field allows designers to create nested structures without defining a new *DataType Node* for each structure. This field is not specified for subtypes of *Enumeration*.
Value	Int32	The value associated with the field. This field is only specified for subtypes of *Enumeration*.

F.13 Variant

The *Variant* type specifies the value for a *Variable* or *VariableType Node*. This type is the same as the type defined in 5.3.1.17. As a result, the functions used to serialize *Variants* during *Service* calls can be used to serialize *Variant* in this file syntax.

Variants can contain *NodeIds*, *ExpandedNodeIds* and *QualifiedNames* which must be modified so the *NamespaceIndexes* and *ServerIndexes* reference the *NamespaceUri and ServerUri* tables in the *UANodeSet*.

Variants can also contain *ExtensionObjects* which contain and *EncodingId* and a *Structure* with fields which could be are *NodeIds*, *ExpandedNodeIds* or *QualifiedNames*. The *NamespaceIndexes* and *ServerIndexes* in these fields shall also reference the tables in the *UANodeSet*.

F.14 Example (Informative)

An example of the *UANodeSet* can be found below.

This example defines the *Nodes* for an *InformationModel* with the URI of "http://sample.com/Instances". This example references *Nodes* defined in the base OPC UA *InformationModel* and an *InformationModel* with the URI "http://sample.com/Types".

The XML namespaces declared at the top include the URIs for the *Namespaces* referenced in the document because the document includes *Complex Data*. Documents without *Complex Data* would not have these declarations.

```
<UANodeSet
xmlns:s1="http://sample.com/Instances"
xmlns:s0="http://sample.com/Types"
xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
```

The *NamespaceUris* table includes all *Namespaces* referenced in the document except for the base OPC UA *InformationModel*. A *NamespaceIndex* of 1 refers to the URI "http://sample.com/Instances".

```
  <NamespaceUris>
    <Uri>http://sample.com/Instances</Uri>
    <Uri>http://sample.com/Types</Uri>
  </NamespaceUris>
```

The *Aliases* table is provided to enhance readability. There are no rules for what is included. A useful guideline would include standard *ReferenceTypes* and *DataTypes* if they are referenced in the document.

```
  <Aliases>
    <Alias Alias="HasComponent">i=47</Alias>
    <Alias Alias="HasProperty">i=46</Alias>
    <Alias Alias="HasSubtype">i=45</Alias>
    <Alias Alias="HasTypeDefinition">i=40</Alias>
  </Aliases>
```

The *BicycleType* is a *DataType Node* that inherits from a *DataType* defined in another *InformationModel* (ns=2;i=314). It is assumed that any *Application* importing this file will already know about the referenced *InformationModel*. A *Server* could map the references onto another OPC UA *Server* by adding a *ServerIndex* to *TargetNode NodeIds*. The structure of the *DataType* is defined by the *Definition* element. This information can be used by code generators to automatically create serializers for the *DataType*.

```
<UADataType NodeId="ns=1;i=365" BrowseName="1:BicycleType">
  <DisplayName>BicycleType</DisplayName>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=314</Reference>
  </References>
```

```
  <Definition Name="BicycleType" BaseType="0:1:BicycleType">
    <Field Name="NoOfGears" DataType="UInt32" />
    <Field Name="ManufacterName" DataType="QualifiedName" />
  </Definition>
</UADataType>
```

This *Node* is an instance of an *Object TypeDefinition Node* defined in another *InformationModel* (ns=2;i=341). It has a single *Property* which is declared later in the document.

```
<UAObject NodeId="ns=1;i=375" BrowseName="1:DriverOfTheMonth" ParentNodeId="ns=1;i=281">
  <DisplayName>DriverOfTheMonth</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=376</Reference>
    <Reference ReferenceType="HasTypeDefinition">ns=2;i=341</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=281</Reference>
  </References>
</UAObject>
```

This *Node* is an instance of a *Variable TypeDefinition Node* defined in base OPC UA *InformationModel* (i=68). The *DataType* is the base type for the *BicycleType DataType*. The *AccessLevels* declare the *Variable* as *Readable* and *Writeable*. The ParentNodeId indicates that this Node is tightly coupled with the Parent (DriverOfTheMonth) and will be deleted if the Parent is deleted.

```
<UAVariable NodeId="ns=1;i=376" BrowseName="2:PrimaryVehicle"
  ParentNodeId="ns=1;i=375" DataType="ns=2;i=314" AccessLevel="3" UserAccessLevel="3">
  <DisplayName>PrimaryVehicle</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=375</Reference>
  </References>
```

This *Value* is an instance of a *BicycleType DataType*. It is wrapped in an *ExtensionObject* which declares that the value is serialized using the *Default XML DataTypeEncoding* for the *DataType*. The *Value* could be serialized using the *Default Binary DataTypeEncoding* but that would result in a document that cannot be edited by hand. No matter which *DataTypeEncoding* is used, the *NamespaceIndex* used in the *ManufacterName* field refers to the *NamespaceUris* table in this document. The *Application* is responsible for changing whatever value it needs to be when the document is loaded by an *Application*.

```
    <Value>
      <ExtensionObject xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
        <TypeId>
          <Identifier>ns=1;i=366</Identifier>
        </TypeId>
        <Body>
          <s1:BicycleType>
            <s0:Make>Trek</s0:Make>
            <s0:Model>Compact</s0:Model>
            <s1:NoOfGears>10</s1:NoOfGears>
              <s1:ManufacterName>
                <uax:NamespaceIndex>1</uax:NamespaceIndex>
                <uax:Name>Hello</uax:Name>
            </s1:ManufacterName>
          </s1:BicycleType>
        </Body>
      </ExtensionObject>
    </Value>
  </UAVariable>
```

These are the *DataTypeEncoding Nodes* for the *BicyleType DataType*.

```
  <UAObject NodeId="ns=1;i=366" BrowseName="Default XML">
    <DisplayName>Default XML</DisplayName>
    <References>
      <Reference ReferenceType="HasEncoding" IsForward="false">ns=1;i=365</Reference>
      <Reference ReferenceType="HasDescription">ns=1;i=367</Reference>
      <Reference ReferenceType="HasTypeDefinition">i=76</Reference>
    </References>
  </UAObject>
```

```
<UAObject NodeId="ns=1;i=370" BrowseName="Default Binary">
  <DisplayName>Default Binary</DisplayName>
  <References>
    <Reference ReferenceType="HasEncoding" IsForward="false">ns=1;i=365</Reference>
    <Reference ReferenceType="HasDescription">ns=1;i=371</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=76</Reference>
  </References>
</UAObject>
```

This is the *DataTypeDescription Node* for the *Default XML DataTypeEncoding* of the *BicyleType DataType*. The *Value* is one of the built-in types.

```
<UAVariable NodeId="ns=1;i=367" BrowseName="1:BicycleType" DataType="String">
  <DisplayName>BicycleType</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">i=69</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=341</Reference>
  </References>
  <Value>
    <uax:String>//xs:element[@name='BicycleType']</uax:String>
  </Value>
</UAVariable>
```

This is the *DataTypeDescription Node* for the *DataTypeDescription* declared above. The XML Schema document is a UTF-8 document stored as a base64 string. This allows clients to read the schema for the

```
<UAVariable NodeId="ns=1;i=341" BrowseName="1:Quickstarts.DataTypes.Instances"
DataType="ByteString">
  <DisplayName>Quickstarts.DataTypes.Instances</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=343</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=367</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">i=92</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=72</Reference>
  </References>
  <Value>
    <uax:ByteString>PHhz...W1hPg==</uax:ByteString>
  </Value>
</UAVariable>
```

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIÉE OPC –

Partie 6: Correspondances

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. À cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La norme internationale IEC 62541-6 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Cette deuxième édition annule et remplace la première édition parue en 2011. Cette édition constitue une révision technique.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) Certaines applications ont besoin de fonctionner dans des environnements ne disposant pas d'accès à des bibliothèques de cryptographie. Pour pallier ce problème, un nouveau protocole de transport HTTPS a été défini en 7.3;

b) La longueur de l'octet de remplissage n'est pas suffisante pour gérer les tailles des clés asymétriques de longueur supérieure à 2048 bits. Ajout d'un octet de remplissage supplémentaire en 6.7.2 pour gérer ce cas.

c) Définition des erreurs fixes dans les URI d'action SOAP en 7.2.2;

d) Nécessité d'une méthode normalisée permettant de sérialiser les nœuds dans un espace d'adresses. Ajout du schéma de l'Ensemble de Nœuds UA (UANodeSet) défini à l'Annexe F.

Le texte de cette norme est issu des documents suivants:

CDV	Rapport de vote
65E/377/CDV	65E/405/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme.

Cette publication a été rédigée selon les Directives ISO/IEC, Partie 2.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture unifiée OPC*, peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "http://webstore.iec.ch" dans les données relatives à la publication recherchée. À cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IMPORTANT – Le logo *"colour inside"* qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

ARCHITECTURE UNIFIÉE OPC –

Partie 6: Correspondances

1 Domaine d'application

La présente partie de l'IEC 62541 spécifie les correspondances de l'architecture unifiée OPC (OPC UA) entre le modèle de sécurité décrit dans l'IEC TR 62541-2, les définitions de services abstraits décrites dans l'IEC 62541-4, les structures de données définies dans l'IEC 62541-5 et les protocoles de réseaux physiques qui peuvent être utilisés pour mettre en œuvre la spécification OPC UA.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and concepts* (disponible en anglais seulement)

IEC TR 62541-2, *OPC Unified Architecture – Part 2: Security model* (disponible en anglais seulement)

IEC 62541-3, *Architecture unifiée OPC – Partie 3: Modèle de l'espace d'adressage*

IEC 62541-4, *Architecture unifiée OPC – Partie 4: Services*

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'Information*

IEC 62541-7, *Architecture unifiée OPC – Partie 7: Profils*

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 5246 – *The TLS Protocol Version 1.2*

http://tools.ietf.org/html/rfc5246.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

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

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

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

RFC 4514: RFC 4514 – LDAP: String Representation of Distinguished Names

http://www.ietf.org/rfc/rfc4514.txt

NTP: RFC 1305 – Network Time Protocol (Version 3)

http://www.ietf.org/rfc/rfc1305.txt

Kerberos: WS Security Kerberos Token Profile 1.1

http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-KerberosTokenProfile.pdf

3 Termes, définitions, abréviations et symboles

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans l'IEC TR 62541-1, l'IEC TR 62541-2 et l'IEC 62541-3, ainsi que les suivants s'appliquent.

3.1.1
codage de données (DataEncoding)
méthode de sérialisation des *Messages* et des structures de données OPC UA

3.1.2
correspondance (Mapping)
spécification de la méthode de mise en œuvre d'une fonctionnalité OPC UA avec une technologie spécifique

Note 1 à l'article: Par exemple, le codage OPC UA Binaire est une *Correspondance* qui spécifie comment sérialiser les structures de données OPC UA en séquences d'octets.

3.1.3
protocole de sécurité (Security Protocol)
protocole garantissant l'intégrité et la confidentialité des *Messages* UA échangés entre des applications OPC UA

3.1.4
profil de pile (Stack Profile)
combinaison des *correspondances Codages de Données*, *Protocole de Sécurité* et *Protocole de Transport*

Note 1 à l'article: Les applications OPC UA mettent en œuvre un ou plusieurs *Profils de Piles* et peuvent communiquer uniquement avec des applications OPC UA qui prennent en charge le même *Profil de Pile* qu'elles.

3.1.5
protocole de transport (Transport Protocol)
méthode d'échange des *Messages* OPC UA sérialisés entre des applications OPC UA

3.2 Abréviations et symboles

API	Application Programming Interface (Interface de programme d'application)
ASN.1	Abstract Syntax Notation (Notation syntaxique abstraite #1) (utilisée dans la recommandation X690)
BP	WS-I Basic Profile (Version de profil de base WS-I)
BSP	WS-I Basic Security Profile (Profil de sécurité de base WS-I)
CSV	Comma Separated Value (Valeur séparée par des virgules) (format de fichier)
HTTP	Hypertext Transfer Protocol (Protocole de Transport hypertexte)
HTTPS	Secure Hypertext Transfer Protocol (Protocole de Transport hypertexte sécurisé)

IPSec	Internet Protocol Security (Sécurité de protocole Internet)
RST	Request Security Token (Demande de jeton de sécurité)
OID	Object Identifier (Identificateur d'objet) (utilisé avec ASN.1)
RSTR	(Request Security Token Response (Réponse à une demande de jeton de sécurité)
SCT	(Security Context Token (Jeton de contexte de sécurité)
SHA1	Secure Hash Algorithm (Algorithme de hachage sécurisé)
SOAP	Single Object Access Protocol (Protocole SOAP (protocole d'accès d'objet simple)
SSL	Secure Sockets Layer (Protocole SSL (Protocole de sécurisation) (défini en SSL/TLS)
TCP	Transmission Control Protocol (Protocole TCP (protocole de contrôle de transmission)
TLS	Transport Layer Security (Protocole TLS) (défini en SSL/TLS)
UTF8	Unicode Transformation Format) (Format de transformation Unicode) (8 bits) (défini en UTF8)
UA	Unified Architecture (Architecture unifiée)
UASC	OPC UA Secure Conversation (Conversation sécurisée OPC UA)
WS-*	XML Web Services (Spécifications des services Web XML)
WSS	WS Security (Sécurité WS)
WS-SC	WS Secure Conversation (Conversation sécurisée WS)
XML	eXtensible Markup Language (Langage de balisage extensible)

4 Vue d'ensemble

Les autres parties de cette série de normes sont rédigées de façon à être indépendantes de la technologie de mise en œuvre utilisée. Cette approche signifie qu'OPC UA est une spécification souple qui s'adaptera aux évolutions technologiques. Par ailleurs, cette approche signifie que la constitution d'une *Application* OPC UA sur la seule base des informations contenues dans les normes IEC TR 62541-1 à IEC 62541-5 est impossible du fait de l'omission de détails de mise en œuvre importants qui sont détaillés ci-après.

La présente norme définit des *Correspondances* entre les spécifications abstraites et les technologies qui peuvent être utilisées pour la mise en œuvre. Les *Correspondances* sont organisées en trois groupes: *Codages de Données, Protocoles de Sécurité* et *Protocoles de Transport.* Différentes *Correspondances* sont associées pour créer des *Profils de Piles*. Toutes les *Applications* OPC UA doivent mettre en œuvre au moins un *Profil de Pile* et peuvent communiquer uniquement avec d'autres *Applications* OPC UA qui mettent en œuvre le même *Profil de Pile*.

La présente norme définit dans l'Article 5 les *Codages de Données*, les *Protocoles de Sécurité* dans l'Article 6 et les *Protocoles de Transport* en 6.7.6. Les *Profils de Piles* sont définis dans l'IEC 62541-7.

Toutes les communications entre les *Applications* OPC UA sont basées sur l'échange de *Messages*. Les paramètres contenus dans les *Messages* sont définis dans l'IEC 62541-4; toutefois, leur format est spécifié par le *Codage de Données* et le *Protocole de Transport.* Ainsi, chaque *Message* défini dans l'IEC 62541-4 doit avoir une description normative qui spécifie exactement ce qui doit être mis sur le réseau de communication. Les descriptions normatives sont définies dans les annexes.

Une *Pile* est un regroupement de bibliothèques logicielles qui mettent en œuvre un ou plusieurs *Profils de Piles.* L'interface entre une *Application* OPC UA et la *Pile* est une interface API non normative qui masque les détails de mise en œuvre de la *Pile*. Une

interface API dépend d'une *Plate-forme de Développement (DevelopmentPlatform)* spécifique. Noter que du fait des limites de la *Plate-forme de Développement,* les types de données présentés dans l'interface API pour une *Plate-forme de Développement* peuvent ne pas correspondre aux types de données définis par la spécification. Par exemple, Java ne prend pas en charge un entier non signé, ce qui signifie qu'il est nécessaire que toute interface API Java mette en correspondance les entiers non signés dans un type d'entier signé.

La Figure 1 illustre la relation entre les différents concepts définis dans cette norme.

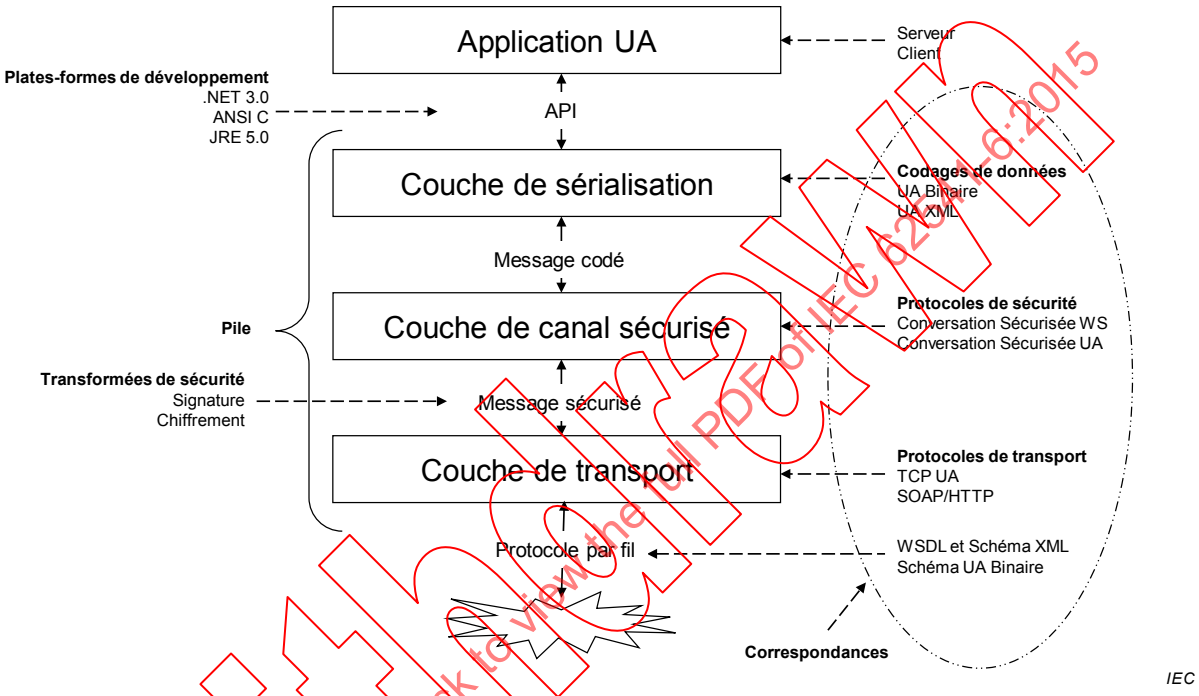

Figure 1 – Vue d'ensemble des piles OPC UA

Les couches décrites dans la présente spécification ne correspondent pas aux couches du modèle OSI 7 [X200]. Il convient de traiter chaque *Profil de Pile* OPC UA comme un protocole avec une Couche Application unique 7 bâtie sur un protocole existant (Couche 5, 6 ou 7), tel que TCP/IP, TLS ou HTTP. La couche *Canal Sécurisé* est toujours présente même si le *Mode de Sécurité* est *Aucun* (*None*). Dans ce type de situation, aucune sécurité n'est appliquée mais la mise en œuvre du *Protocole de Sécurité* doit maintenir un canal logique avec un Identificateur unique. Les utilisateurs et les administrateurs sont censés être conscients du fait qu'un *Canal Sécurisé* dont le *Mode de Sécurité* est réglé sur *Aucun* ne peut pas être fiable, à moins que l'*Application* ne fonctionne sur un réseau physiquement sécurisé ou qu'un protocole de bas niveau, tel que IPSec, ne soit utilisé.

5 Codage de données

5.1 Généralités

5.1.1 Vue d'ensemble

La présente norme définit deux types de codage de données: OPC UA Binaire et OPC UA XML. Elle décrit comment créer des *Messages* avec chacun de ces codages.

5.1.2 Types intégrés

Tous les *Codages de Données* OPC UA sont fondés sur les règles établies pour un ensemble normalisé de types intégrés. Ces types intégrés permettent alors de créer des structures, des matrices et des *Messages*. Les types intégrés sont décrits dans le Tableau 1.

Tableau 1 – Types de Données intégrés

ID (Identificateur)	Dénomination	Description
1	Boolean	Valeur logique binaire (vrai ou faux).
2	Sbyte	Valeur entière entre −128 et 127.
3	Byte	Valeur entière entre 0 et 256.
4	Int16	Valeur entière entre −32 768 et 32 767.
5	UInt16	Valeur entière entre 0 et 65 535.
6	Int32	Valeur entière entre −2 147 483 648 et 2 147 483 647.
7	UInt32	Valeur entière entre 0 et 429 4967 295.
8	Int64	Valeur entière entre −9 223 372 036 854 775 808 et 9 223 372 036 854 775 807.
9	UInt64	Valeur entière entre 0 et 18 446 744 073 709 551 615.
10	Float	Valeur à virgule flottante (32 bits) en simple précision IEEE.
11	Double	Valeur à virgule flottante (64 bits) en double précision IEEE.
12	String	Séquence de caractères Unicode.
13	DateTime	Instance dans le temps.
14	Guid	Valeur à 16 octets qui peut être utilisée comme identificateur globalement unique.
15	ByteString	Séquence d'octets.
16	XmlElement	Un élément XML.
17	NodeIdentifier	Identificateur d'un nœud dans l'espace d'adresses d'un Serveur OPC UA.
18	ExtendedNodeId	Identificateur de Nœud permettant de spécifier l'URI d'espace de nom en lieu et place d'un indice.
19	StatusCode	Identifiant numérique d'une erreur ou d'un état associé à une valeur ou une opération.
20	QualifiedName	Nom qualifié par un espace de nom.
21	LocalizedText	Texte lisible en clair avec un identificateur de lieu facultatif.
22	ExtensionObject	Structure contenant un type de données spécifique à l'application susceptible de ne pas être reconnu par le récepteur.
23	DataValue	Valeur de données avec code de statut et des horodatages associés.
24	Variant	Union de tous les types spécifiés ci-dessus.
25	DiagnosticInfo	Structure contenant une erreur détaillée et des informations de diagnostic associées à un Code de Statut

La plupart de ces types de données sont identiques aux types abstraits définis dans l'IEC 62541-3 et l'IEC 62541-4. Cette norme définit toutefois les types *Objet d'Extension* et *Variante*. De plus, la présente norme définit une représentation pour le type *Guid (Identificateur Globalement Unique)* défini dans l'IEC 62541-3.

5.1.3 Guid (Identificateur globalement Unique)

Identificateur globalement unique de 16 octets, dont la configuration est indiquée dans le Tableau 2.

Tableau 2 – Structure du Guid

Composant	Type de données
Data1	UInt32
Data2	UInt16
Data3	UInt16
Data4	Byte[8]

Les valeurs du *Guid* peuvent être représentées sous la forme de la chaîne suivante:

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

Lorsque Data1 a une largeur de 8 caractères, Data2 et Data3 ont une largeur de 4 caractères et chaque *Octet* dans Data4 a une largeur de 2 caractères. Chaque valeur est formatée sous forme de nombre hexadécimal rempli de zéros. Une valeur de *Guid* typique ressemble à ce qui suit lorsque son format est celui d'une chaîne:

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

5.1.4 Chaîne d'octets

La structure d'une *Chaîne d'Octets* est identique à celle d'une matrice unidimensionnelle d'*Octet.* Elle est représentée comme un type de données intégré distinct dans la mesure où elle permet aux codeurs d'optimiser la transmission de la valeur. Cependant, certaines *Plates-formes de Développement* ne sont pas capables de pérenniser la distinction entre une *Chaîne d'Octets* et une matrice unidimensionnelle d'*Octet*.

Si un décodeur applicable à la *Plate-forme de Développement* ne peut pas pérenniser cette distinction, il doit convertir toutes les matrices unidimensionnelles d'*Octet* en *Chaînes d'Octets*.

Chaque élément d'une matrice unidimensionnelle de *Chaîne* d'*Octets* peut avoir une longueur différente, ce qui signifie que sa structure est différente de celle d'une matrice bidimensionnelle d'*Octet* où la longueur de chaque dimension est identique. Cela signifie que les décodeurs doivent pérenniser la distinction entre deux matrices dimensionnelles ou plus d'*Octet* et une ou plusieurs matrices dimensionnelles de *Chaîne d'Octets*.

Si une *Plate-forme de Développement* ne prend pas en charge les entiers non signés, il faut alors qu'elle représente les *Chaînes d'Octets* sous forme de matrices de *SOctet*. Dans ce cas, les exigences concernant *Octet* s'appliqueraient alors à *SOctet*.

5.1.5 Objet d'Extension

Un *Objet* d'*Extension* contient tous types de *Données Complexes* qui ne peuvent pas être codées comme un des autres types de données intégrés. L'*Objet d'Extension* contient une valeur complexe sérialisée sous forme d'une séquence d'octets ou d'un élément XML. Il contient également un identificateur qui indique les données qu'il contient, ainsi que la méthode de codage utilisée.

Les types de *Données Complexes* sont représentés dans l'espace d'adresses du *Serveur* sous forme de sous-types du *Type de Données* de *Structure*. Les *Codages de Données* disponibles pour tout type de *Données Complexes* sont représentés comme un *Objet de Codage de Type de Données* dans *l'Espace d'adresses* du *Serveur*. L'*Identificateur de Nœud* de l'*Objet de Codage de Type de Données* est l'identificateur archivé dans l'*Objet d'Extension*. L'IEC 62541-3 décrit comment les *Nœuds Codage de Type de Données* sont liés aux autres *Nœuds* de l'*Espace d'adresses*.

Il convient que les ingénieurs qui implémentent des *Serveurs* utilisent des *Identificateurs de Nœuds* numériques qualifiés de l'espace de nom pour les *Objets de Codage de Type de Données* qu'ils définissent. Ceci permet de réduire au minimum la surcharge générée par le tassement des valeurs de *Données Complexes* dans les *Objets d'Extension.*

5.1.6 Variante

Une *Variante* est l'union de tous les types de données intégrés, y compris un *Objet d'Extension.* Les *Variantes* peuvent également contenir des matrices de ces types intégrés. Elles servent à mémoriser toute valeur ou tout paramètre avec un type de données de *Type de Données de Base* ou l'un de ses sous-types.

Les *Variantes* peuvent être vides. On décrit une *Variante* vide comme une variante ayant une valeur *Nulle*, et qu'il convient de traiter comme une colonne Nulle dans une base de données SQL. Une valeur *Nulle* dans une *Variante* peut ne pas être identique à une valeur *Nulle* pour les types de données qui prennent en charge les valeurs *Nulles*, telles que les *Chaînes*. Certaines *Plates-formes* de *Développement* peuvent ne pas être capables de pérenniser la distinction entre une valeur nulle pour un *Type de Données* et une valeur nulle pour une *Variante*. Par conséquent, les *Applications* ne doivent pas reposer sur cette distinction.

Les *Variantes* peuvent contenir des matrices de *Variantes*, mais ne peuvent pas contenir directement une autre *Variante*.

Les types *Valeur de Données* et *Information de Diagnostic* n'ont qu'une signification que lorsqu'ils sont renvoyés dans un message de réponse avec un *Code de Statut* associé. De ce fait, les *Variantes* ne peuvent pas contenir des instances de *Valeur de Données* ou d'*Information de Diagnostic*.

Les *Variables* avec un *Type de Données* de *Type de Données de Base* sont mises en correspondance avec une *Variante;* toutefois, les *Attributs Rang de Valeur* et *Dimensions de Matrice* imposent des restrictions concernant le contenu autorisé de la *Variante*. Par exemple, si le *Rang de Valeur* est *Scalaire,* alors la *Variante* ne peut contenir que des valeurs scalaires.

5.2 OPC UA Binaire

5.2.1 Généralités

Le *Codage* OPC UA de *Données* Binaires est un format de données développé afin de satisfaire aux exigences de performance des *Applications* OPC UA. Ce format est conçu principalement pour un codage et un décodage rapides. Toutefois, il a également été tenu compte de la taille des données codées mises sur le réseau.

Le *Codage* OPC UA de *Données Binaires* repose sur plusieurs types de données primitives avec des règles de codage clairement définies, qui peuvent être écrites ou lues suivant un ordre séquentiel à partir d'un flot de bits. Le codage d'une structure s'effectue par l'écriture de la forme codée de chaque champ suivant un ordre séquentiel. Lorsqu'un champ donné est également une structure, les valeurs de ses champs sont écrites suivant un ordre séquentiel, avant d'écrire le champ suivant dans la structure contenant. Tous les champs doivent être écrits dans la séquence de bits, même s'ils contiennent des valeurs nulles. Les codages applicables à chaque type de primitive spécifient comment coder soit une valeur nulle, soit une valeur par défaut pour le type concerné.

Le *Codage* OPC UA de *Données Binaires* ne comprend aucune information de nom de type ou de champ, dans la mesure où toutes les applications OPC UA sont supposées avoir une connaissance avancée des services et des structures qu'elles prennent en charge. Un *Objet d'Extension* qui fournit un identificateur et une taille pour la structure de *Données Complexes* qu'il représente constitue une exception. Ceci permet à un décodeur de sauter les types qu'il ne reconnaît pas.

5.2.2 Types intégrés

5.2.2.1 Booléen (Boolean)

Une valeur *Booléen* doit être codée comme un octet simple, où une valeur de 0 (zéro) correspond à Faux et toute valeur non nulle, correspond à Vrai.

Les codeurs doivent utiliser la valeur de 1 pour indiquer une valeur Vrai. Les décodeurs doivent en revanche traiter toute valeur non nulle comme Vrai.

5.2.2.2 Entier (Integer)

Tous les types d'entier doivent être codés comme valeurs «little endian» («petit-boutistes») où l'octet de poids faible apparaît en premier dans la séquence des bits.

La Figure 2 illustre la méthode qu'il convient d'utiliser pour coder la valeur 1 000 000 000 (Hex: 3B9ACA00) comme un entier à 32 bits dans la séquence binaire.

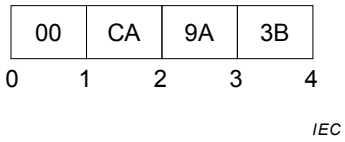

Figure 2 – Codage des entiers dans une séquence binaire

5.2.2.3 Virgule flottante (Float)

Toutes les valeurs à virgule flottante doivent être codées avec la représentation binaire IEEE-754 appropriée qui comporte trois composants de base: le signe, l'exposant et la fraction. Les plages de bits attribuées à chaque composant dépendent de la largeur du type. Le Tableau 3 donne la liste des plages de bits relatives aux types à virgule flottante pris en charge.

Tableau 3 – Types à virgule flottante pris en charge

Dénomination	Largeur (bits)	Fraction	Exposant	Signe
Virgule flottante	32	0-22	23-30	31
Double	64	0-51	52-62	63

De plus, l'ordre des octets dans la séquence des bits est important. Toutes les valeurs à virgule flottante doivent être codées avec l'octet de poids faible apparaissant en premier (c'est-à-dire «little endian»).

La Figure 3 illustre la méthode qu'il convient d'utiliser pour coder la valeur −6,5 (Hex: C0D00000) comme une *Virgule flottante*.

Le type à virgule flottante prend en charge l'infinité positive et négative et non pas un nombre (NaN). La spécification IEEE permet l'utilisation de plusieurs variantes NaN. Toutefois, les codeurs/décodeurs peuvent ne pas faire la distinction. Les codeurs doivent coder une valeur NaN comme une NAN «quiet» IEEE (000000000000F8FF) ou (0000C0FF). Les types non pris en charge éventuels, tels que les nombres non normalisés, doivent être également codés sous la forme d'une NAN «quiet» IEEE.

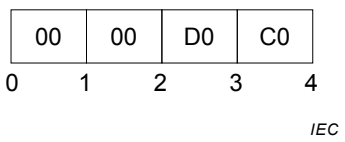

Figure 3 – Codage des virgules flottantes dans une séquence binaire

5.2.2.4 Chaîne (String)

Toutes les valeurs *Chaîne* sont codées sous forme de séquence de caractères UTF8 sans terminateur nul et précédée par la longueur en octets.

La longueur en octets est codée sous la forme *Int32*. Une valeur de -1 permet d'indiquer une chaîne «nulle».

La Figure 4 illustre la méthode qu'il convient d'utiliser pour le codage de la chaîne multilingue "水Boy" dans un train d'octets.

Figure 4 – Codage de chaînes dans une séquence binaire

5.2.2.5 Date&Heure (DateTime)

Une valeur *Date&Heure* doit être codée sous la forme d'un entier signé de 64 bits (voir 5.2.2.2) qui représente le nombre d'intervalles de 100 nanosecondes depuis le 1er janvier 1601 (UTC).

Les *Plates-formes de Développement* ne sont pas toutes capables de représenter la plage complète de dates et d'heures qui peut être représentée par ce *Codage de Données*. Par exemple, la structure time_t sous UNIX a uniquement une résolution de 1 seconde et ne peut représenter de dates antérieures à 1970. Pour cette raison, un certain nombre de règles doivent être appliquées lorsqu'il s'agit de valeurs date/heure au-delà de la plage dynamique d'une *Plate-forme de Développement*. Ces règles sont les suivantes:

a) Une valeur date/heure est codée 0 si, soit
   1) la valeur est égale ou antérieure à 1601-01-01 12:00AM,
   2) la valeur constitue la date la plus proche qui peut être représentée avec le codage de la *Plate-forme de Développement*.
b) Une valeur date/heure est codée comme la valeur maximale d'un *Int64* si, soit
   1) la valeur est égale ou supérieure à 9999-01-01 11:59:59PM,
   2) la valeur constitue la date la plus lointaine qui peut être représentée avec le codage de la *Plate-forme de Développement*.
c) Une valeur date/heure est décodée comme l'heure la plus proche qui peut être représentée sur la plate-forme si, soit
   1) la valeur codée est 0,
   2) la valeur codée représente une heure antérieure à l'heure la plus proche qui peut être représentée avec le codage de la *Plate-forme de Développement*.
d) Une valeur date/heure est décodée comme l'heure la plus éloignée qui peut être représentée sur la plate-forme si, soit
   1) la valeur codée est la valeur maximale d'un *Int64,*
   2) la valeur codée représente une heure ultérieure à l'heure la plus lointaine qui peut être représentée avec le codage de la *Plate-forme de Développement*.

Ces règles impliquent que les heures les plus proches et les plus lointaines qui peuvent être représentées sur une plate-forme donnée sont des valeurs de date/heure non valides, qu'il convient de traiter comme tel par les *Applications*.

Un décodeur doit tronquer la valeur s'il rencontre une valeur *Date&Heure* dont la résolution est supérieure à celle prise en charge sur la *Plate-forme de Développement*.

5.2.2.6 Guid (Identificateur globalement Unique)

Un *Guid* est codé dans une structure tel qu'indiqué dans le Tableau 2. Le codage des champs s'effectue suivant un ordre séquentiel selon le type de données propre au champ.

La Figure 5 illustre la méthode qu'il convient d'utiliser pour le codage du *Guid* 72962B91-FA75-4ae6-8D28-B404DC7DAF63" dans une séquence d'octets.

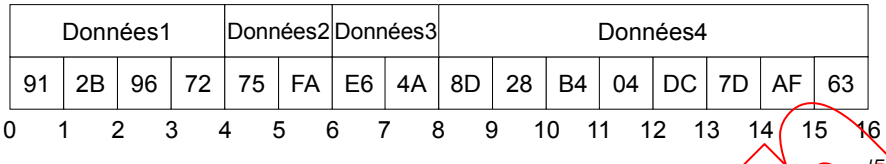

Figure 5 – Codage des Guid dans une séquence binaire

5.2.2.7 Chaîne d'Octets (ByteString)

Une *Chaîne d'Octets* est codée comme une séquence d'octets précédée de sa longueur en octets. La longueur est codée comme un entier signé de 32 bits tel que décrit ci-dessus.

Si la longueur de la chaîne d'octets est -1, la chaîne d'octets est alors «nulle».

5.2.2.8 Élément Xml (XmlElement)

Un *Élément Xml* est un fragment XML sérialisé sous forme d'une chaîne UTF8, puis codé sous forme de *Chaîne d'Octets*.

La Figure 6 illustre la méthode qu'il convient d'utiliser pour le codage de l'*Elément Xml* "<A>Hot水</A>" dans un train d'octets.

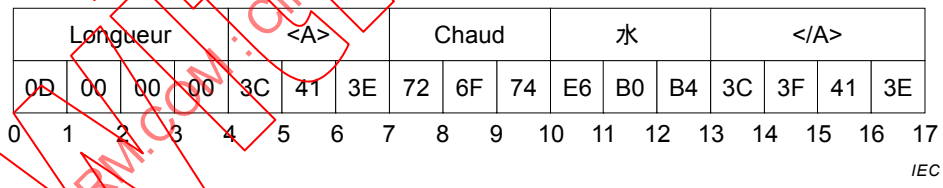

Figure 6 – Codage des Eléments Xml dans une séquence binaire

5.2.2.9 Identificateur de Nœud (NodeId)

Les composants d'un *Identificateur de Nœud* sont décrits dans le Tableau 4.

Tableau 4 – Composants d'un Identificateur de Nœud

Dénomination	Type de données	Description
Namespace	UInt16 (Entier Non Signé codé sur 16 bits)	Indice d'un URI d'espace de nom. Un indice de 0 est utilisé pour les *Identificateurs de Nœud* définis OPC UA.
IdentifierType	Enum	Le format et le type de données de l'identificateur. La valeur peut être l'une des valeurs suivantes NUMERIC - la valeur est un *Entier Non Signé* (*Uinteger);* STRING - la valeur est *Chaîne* GUID - la valeur est un *Guid*; OPAQUE - la valeur est une *Chaîne d'Octets;*
Value	*	Identificateur d'un nœud dans l'espace d'adresses d'un *Serveur* OPC UA.

Le *Codage de Données* d'un *Identificateur de Nœud* varie selon le contenu de l'instance. Ainsi, le premier octet de la forme codée indique le format du reste de l'*Identificateur de Nœud* codé. Les formats de *Codage de Données* possibles sont présentés dans le Tableau 5. Les Tableaux ci-dessous décrivent la structure de chaque format possible (ils excluent l'octet qui indique le format).

Tableau 5 – Valeurs de Codage de Données de l'Identificateur de Nœud

Dénomination	Valeur	Description
Two Byte	0x00	Valeur numérique ajustée à la représentation à deux octets.
Four Byte	0x01	Valeur numérique ajustée à la représentation à quatre octets.
Numeric	0x02	Valeur numérique non ajustée à la représentation à deux ou à quatre octets.
String	0x03	Valeur Chaîne.
Guid	0x04	Valeur Guid.
ByteString	0x05	Valeur opaque (Chaîne d'Octets).
NamespaceUri Flag	0x80	Voir explication de *Identificateur de Nœud Etendu* en 5.2.2.10.
ServerIndex Flag	0x40	Voir explication de *Identificateur de Nœud Etendu* en 5.2.2.10.

La structure du *Codage de Données* de l'*Identificateur de Nœud* normalisé est décrite dans le Tableau 6. Le *Codage de Données* normalisé est utilisé pour tous les formats non définis de manière explicite.

Tableau 6 – Codage de Données binaires normalisé d'Identificateur de Nœud

Dénomination	Type de données	Description
Namespace	UInt16	*Indice d'espace de nom.*
Identifier	*	Identificateur codé selon les règles suivantes: NUMERIC UInt32 STRING Chaîne GUID Guid OPAQUE Chaîne d'Octets

Un exemple d'*Identificateur de Nœud* de chaîne avec Espace de Nom = 1 et Identificateur = "Hot水" est illustré à la Figure 7.

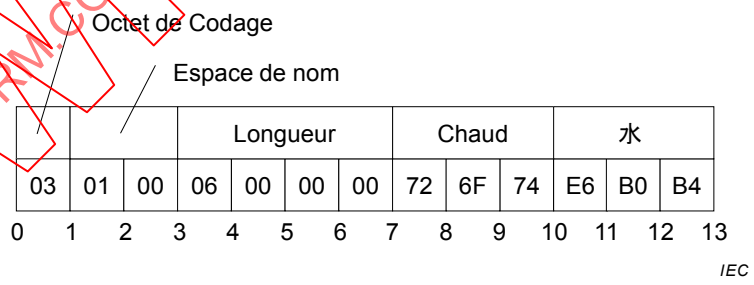

Figure 7 – Identificateur de Nœud de chaîne

La structure du *Codage de Données* de l'*Identificateur de Nœud* à deux octets est décrite dans le Tableau 7.

Tableau 7 – Codage de Données binaires de l'Identificateur de nœud à deux octets

Dénomination	Type de données	Description
Identifier	Octet	*L'espace de nom* est *l'espace de nom* OPC UA par défaut (c'est-à-dire 0). Le type d'*identificateur* est «Numérique». L'*identificateur* doit se situer dans la plage comprise entre 0 et 255.

Un exemple d'*Identificateur de Nœud* à deux octets avec Identificateur = 72 est illustré à la Figure 8.

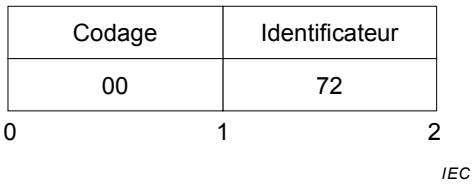

Figure 8 – Identificateur de Nœud à deux octets

La structure du *Codage de Données* de l'*Identificateur de Nœud* à quatre octets est décrite dans le Tableau 8.

Tableau 8 – Codage de Données binaires de l'Identificateur de Nœud à quatre octets

Dénomination	Type de données	Description
Namespace	Octet	*L'espace de nom* doit se situer dans la plage comprise entre 0 et 255.
Identifier	UInt16	Le type d'*identificateur* est «Numérique» L'*identificateur* doit être un entier situé dans la plage comprise entre 0 et 65 535.

Un exemple d'*Identificateur de Nœud* à quatre octets avec Espace de Nom = 5 et Identificateur = 1 025 est illustré à la Figure 9.

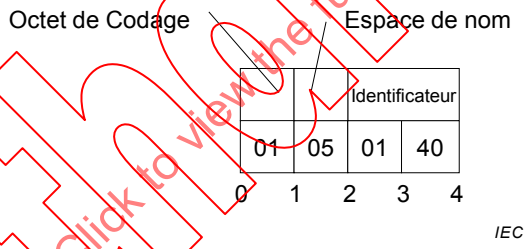

Figure 9 – Identificateur de Nœud à quatre octets

5.2.2.10 Identificateur de Nœud Etendu (ExpandedNodeId)

Un *Identificateur de Nœud Etendu* étend la structure de l'*Identificateur de Nœud* en permettant une spécification explicite de l'*Uri d'Espace de Nom* en lieu et place de l'utilisation de l'*Indice d'Espace de Nom.* L'*Uri d'Espace de Nom* est facultatif. Si ce dernier est spécifié, l'*Indice d'Espace de Nom* au sein de l'*Identificateur de Nœud* doit être ignoré.

Le codage de l'*Identificateur de Nœud Etendu* s'effectue tout d'abord par le codage d'un *Identificateur de Nœud* tel que décrit au 5.2.2.9, puis par le codage de l'*Uri d'Espace de Nom* sous forme d'une *Chaîne*.

Une instance d'un *Identificateur de Nœud Etendu* peut continuer à utiliser l'*Indice d'Espace de Nom* en lieu et place de l'*Uri d'Espace de Nom.* Dans ce cas, l'*Uri d'Espace de Nom* n'est pas codé dans la séquence des bits. La présence de l'*Uri d'Espace de Nom* dans la séquence des bits est indiquée par l'établissement du fanion *Uri d'Espace de Nom* dans l'octet du format de codage propre à l'*Identificateur de Nœud.*

En cas de présence de l'*Uri d'Espace de Nom,* le codeur doit alors coder l'*Indice d'Espace de Nom* sous la valeur 0 dans la séquence des bits lorsque la partie *Identificateur de Nœud* est codée. Pour être cohérent, l'*Indice d'Espace de Nom* non utilisé est inclus dans la séquence des bits.

Un *Identificateur de Nœud Etendu* peut également comporter un *Indice de Serveur* codé comme un *UInt32* après l'*Uri d'Espace de Nom.* L'indicateur *Indice de Serveur* dans l'octet de codage de l'*Identificateur de Nœud* indique la présence ou non de l'*Indice de Serveur* dans la séquence des bits. L'*Indice de Serveur* est omis s'il est égal à zéro.

La structure du codage de l'*Identificateur de Nœud Etendu* est décrite dans le Tableau 9.

Tableau 9 – Codage de Données Binaires de l'Identificateur de Nœud Etendu

Dénomination	Type de données	Description
NodeId	Identificateur de Nœud	Les fanions Uri d'Espace de Nom et Indice de Serveur dans le codage de l'Identificateur de Nœud indiquent si ces champs sont présents ou non dans la séquence de bits.
NamespaceUri	Chaîne	Non présent si nul ou vide.
ServerIndex	UInt32	Non présent si 0.

5.2.2.11 Code de Statut (StatusCode)

Un *Code de Statut* est codé comme un *UInt32*.

5.2.2.12 Information de diagnostic (DiagnosticInfo)

Une structure d'*Information de Diagnostic* est décrite dans l'IEC 62541-4. Elle spécifie un nombre de champs susceptibles d'être manquants. Pour cette raison, le codage utilise un masque de bits pour indiquer quels champs sont effectivement présents dans la forme codée.

Tel que décrit dans l'IEC 62541-4, les champs *Identificateur Symbolique*, *Uri d'Espace de Nom*, *Texte Localisé* et *Paramètres de Lieu* (*Locale*) constituent des indices dans une table de chaînes renvoyée dans l'en-tête de réponse. Seul l'indice de la chaîne correspondante dans le tableau de chaînes est codé. Un indice de -1 indique qu'il n'existe aucune valeur pour la chaîne.

Tableau 10 – Codage de Données Binaires de l'Information de Diagnostic

Dénomination	Type de données	Description
Encoding Mask	Octet	Masque de bits indiquant les champs présents dans le train de bits. Le masque comprend les bits suivants: 0x01 Identificateur Symbolique 0x02 Espace de Nom 0x04 Texte Localisé 0x08 Paramètres de lieu 0x10 Information complémentaire 0x20 Code de Statut Interne 0x40 Information de Diagnostic Interne
SymbolicId	Int32	Nom symbolique du code de statut.
NamespaceUri	Int32	Espace de nom qui qualifie l'identificateur symbolique.
LocalizedText	Int32	Résumé lisible en clair du code de statut.
Locale	Int32	Lieu utilisé pour le texte localisé.
Additional Info	Chaîne	Information de diagnostic spécifique à l'application détaillée.
InnerStatusCode	Code de Statut	Code de statut fourni par un système sous-jacent.
InnerDiagnosticInfo	Information de Diagnostic	Information de diagnostic associée au code de statut interne.

5.2.2.13 Nom Qualifié (QualifiedName)

Une structure de *Nom Qualifié* est codée comme indiqué dans le Tableau 11.

La structure abstraite de *Nom Qualifié* est définie dans l'IEC 62541-3.

Tableau 11 – Codage de Données Binaires de Nom Qualifié

Dénomination	Type de données	Description
NamespaceIndex	UInt16	Indice d'Espace de nom.
Name	Chaîne	Le nom.

5.2.2.14 Texte Localisé (LocalizedText)

Une structure de *Texte Localisé* contient deux champs susceptibles d'être manquants. Pour cette raison, le codage utilise un masque de bits pour indiquer quels champs sont effectivement présents dans la forme codée.

La structure abstraite de *Texte Localisé* est définie dans l'IEC 62541-3.

Tableau 12 – Codage de Données Binaires de Texte Localisé

Dénomination	Type de données	Description
EncodingMask	Octet	Masque de bits indiquant les champs présents dans le train de bits. Le masque comprend les bits suivants: 0x01 Paramètres de lieu 0x02 Texte
Locale	Chaîne	Paramètres de lieu. Omis signifie nul ou vide.
Text	Chaîne	Texte dans le paramètre de lieu spécifié. Omis signifie nul ou vide.

5.2.2.15 Objet d'extension (ExtensionObject)

Un *Objet d'Extension* est codé comme une séquence d'octets dont le préfixe est l'*Identificateur de Nœud* de son *Codage de Type de Données* et le nombre d'octets codés.

Un *Objet d'Extension* peut être codé par l'*Application*, ce qui signifie qu'il est transmis au codeur sous forme de *Chaîne d'Octets* ou d'*Élément Xml.* Dans ce cas, le codeur est capable d'écrire le nombre d'octets dans l'objet avant de coder les octets. Toutefois, il est possible qu'un *Objet d'Extension* sache comment coder/décoder lui-même, ce qui signifie que le codeur doit calculer le nombre d'octets avant de coder l'objet, ou il doit être capable d'effectuer une rétrorecherche dans la séquence de bits et d'actualiser la longueur après codage du corps de l'objet.

Lorsqu'un décodeur rencontre un *Objet d'Extension,* il doit vérifier s'il reconnaît l'identificateur de *Codage de Type de Données*. Si c'est le cas, il peut demander à la fonction appropriée de décoder le corps de l'objet. Si le décodeur ne reconnaît pas le type, il doit utiliser le *Masque de Codage* pour déterminer si le corps est une *Chaîne d'Octets* ou un *Élément Xml,* puis décoder le corps de l'objet ou le traiter comme une donnée opaque et l'ignorer.

La forme sérialisée d'un *Objet d'Extension* est présentée dans le Tableau 13.

Tableau 13 – Codage de Données Binaires de l'Objet d'Extension

Dénomination	Type de Données	Description
TypeId	Identificateur de Nœud	Identificateur du nœud de *codage de Type de Données* dans *l'Espace d'adresses* du *Serveur*. Les *Objets d'Extension* définis par la spécification OPC UA ont un identificateur de nœud numérique qui leur est attribué avec un *Indice d'Espace de Nom* de 0. Les identificateurs numériques sont définis en A.1.
Encoding	Octet	Énumération qui indique la méthode de codage du corps. Le paramètre peut avoir les valeurs suivantes: 0x00 Aucun corps codé. 0x01 Le corps est codé comme une chaîne d'Octets. 0x02 Le corps est codé comme un Élément Xml.
Length	Int32	Longueur du corps de l'objet. La longueur doit être spécifiée si le corps est codé.
Body	Octet[*]	Le corps de l'objet. Ce champ contient les octets bruts des corps de Chaîne d'Octets. Pour les corps d'Élément Xml, ce champ contient le XML codé sous forme de chaîne UTF-8 sans terminateur nul.

Les *Objets d'Extension* sont utilisés dans deux contextes: sous forme de valeurs contenues dans les structures de *Variante* ou sous forme de paramètres dans les *Messages* OPC UA.

5.2.2.16 Variante

Une *Variante* est une union des types intégrés.

La structure d'une *Variante* est présentée au Tableau 14.

Tableau 14 – Codage de Données Binaires de Variante

Dénomination	Type de Données	Description
EncodingMask	Octet	Type de données codé dans la séquence de bits. Les bits suivants sont attribués au masque: 0:5 Identificateur de Type Intégré (voir Tableau 1). 6 Vrai si le champ Dimensions de Matrice est codé. 7 Vrai si une matrice de valeurs est codée.
ArrayLength	Int32	Le nombre d'éléments dans la matrice. Ce champ est présent uniquement si le bit de matrice est établi dans le masque de codage. Les matrices multidimensionnelles sont codées sous forme d'une matrice unidimensionnelle et ce champ spécifie le nombre total d'éléments. Il est possible de reconstituer la matrice d'origine à partir des dimensions codées après le champ Valeur. Les dimensions de rang supérieur sont sérialisées en premier lieu. Par exemple, une matrice de dimensions [2,2,2] s'écrit dans l'ordre suivant: [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	*	Valeur codée selon son type de données intégré. Si le bit de matrice est établi dans le masque de codage, chaque élément dans la matrice est codé de manière séquentielle. Dans la mesure où de nombreux types ont un codage de longueur variable, chaque élément doit être décodé dans l'ordre. La valeur ne doit pas être une *Variante*, mais peut être une matrice de *Variantes*. De nombreuses plates-formes de mise en œuvre ne font pas la différence entre les Matrices unidimensionnelles d'*Octets* et les *Chaînes d'Octets*. Ainsi, les décodeurs sont autorisés à convertir automatiquement une Matrice d'*Octets* en une *Chaîne d'Octets*.
ArrayDimensions	Int32[]	La longueur de chaque dimension. Ce champ est présent uniquement si le fanion des dimensions de matrice est établi dans le masque de codage. Les dimensions de rang inférieur apparaissent en premier dans la matrice.

Les types et leurs identificateurs qui peuvent être codés dans une *Variante* sont présentés dans le Tableau 1.

5.2.2.17 Valeur de Données (DataValue)

Une *Valeur de Données* est toujours précédée d'un masque qui indique quels champs sont présents dans la séquence de bits.

Les champs d'une *Valeur de Donnée* sont décrits dans le Tableau 15.

Tableau 15 – Codage de Données Binaires de la Valeur de Données

Dénomination	Type de données	Description
Encoding Mask	Octet	Masque de bits indiquant quels champs sont présents dans la séquence de bits. Le masque a les bits suivants: 0x01 Faux si la valeur est *Nulle*. 0x02 Faux si le Code de Statut est *Correct*. 0x04 Faux si l'Horodatage Source est *Valeur Minimale Date&Heure*.. 0x08 Faux si l'Horodatage Serveur est *Valeur Minimale Date&Heure.* 0x10 Faux si les Picosecondes Source sont égales à 0. 0x20 Faux si les Picosecondes Serveur sont égales à 0.
Value	Variante	La valeur. Absente si le bit de Valeur dans le Masque de Codage est à l'état Faux.
Status	Code de Statut	L'état associé à la valeur. Absent si le bit de Code de Statut dans le Masque de Codage est à l'état Faux.
SourceTimestamp	Date&Heure	Horodatage Source associé à la valeur. Absent si le bit d'Horodatage Source dans le Masque de Codage est à l'état Faux.
SourcePicoseconds	UInt16	Nombre d'intervalles de 10 picosecondes pour l'Horodatage Source. Absent si le bit de Picosecondes de la Source dans le Masque de Codage est à l'état Faux. En l'absence de l'horodatage source, il n'est pas tenu compte des picosecondes.
ServerTimestamp	Date&Heure	Horodatage *Serveur* associé à la valeur. Absent si le bit d'Horodatage Serveur dans le Masque de Codage est à l'état Faux.
ServerPicoseconds	UInt16	Nombre d'intervalles de 10 picosecondes pour l'Horodatage Serveur. Absent si bit de Picosecondes du Serveur dans le Masque de Codage est à l'état Faux. En l'absence de l'horodatage *Serveur*, il n'est pas tenu compte des picosecondes.

Les champs "*Picosecondes*" mémorisent la différence entre un horodatage haute résolution avec une résolution de 10 picosecondes et la valeur de champ "*Horodatage"* ayant une résolution de 100 ns uniquement. Les champs "*Picosecondes*" doivent contenir des valeurs inférieures à 10 000. Le décodeur doit traiter les valeurs supérieures ou égales à 10 000 comme la valeur '9999'.

5.2.3 Énumérations

Les énumérations sont codées comme valeurs *Int32*.

5.2.4 Matrices

Les *Matrices* qui apparaissent à l'extérieur d'une *Variante* sont codées comme une séquence d'éléments précédée du nombre d'éléments codés comme une valeur Int32. Si une *Matrice* est nulle, sa longueur est alors codée −1. Une *Matrice* de longueur nulle étant différente d'une *Matrice* nulle, les codeurs et les décodeurs doivent donc conserver cette distinction.

Les matrices multidimensionnelles peuvent être codées uniquement au sein d'une *Variante*.

5.2.5 Structures

Les *structures* sont codées sous forme d'une séquence de champs dans leur ordre d'apparition dans la définition. Le codage de chaque champ est déterminé par le type intégré propre au champ.

Tous les champs spécifiés dans le type complexe doivent être codés.

Les *structures* n'ont pas de valeur *nulle*. Si un codeur est écrit dans un langage de programmation qui permet aux structures d'avoir des valeurs nulles, le codeur doit créer une nouvelle instance comportant des valeurs par défaut pour tous les champs et procéder à une sérialisation. Les codeurs ne doivent pas générer une erreur de codage dans ce type de situation.

L'exemple suivant est un exemple de structure utilisant la syntaxe C++:

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

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

Le champ Y est un pointeur dirigé vers une matrice dont la longueur est mémorisée dans NoOfY.

Une instance de *Type1* qui contient une matrice de deux instances de *Type2* serait codée comme une séquence de 37 octets. Si l'instance de *Type1* était codée dans un *Objet d'Extension,* elle aurait la forme de codage présentée dans le Tableau 16. L'*Identificateur de Type* (TypeId), le Codage (Encoding) et la Longueur (Length) sont des champs définis par l'*Objet d'Extension*. Le codage des instances de *Type2* n'inclut pas l'identificateur de type dans la mesure où il est défini de manière explicite dans le *Type1*.

Tableau 16 – Échantillon de structure codée binaire OPC UA

Champ	Octets	Valeur
Identificateur de Type	4	Identificateur du Type1
Codage	1	0x1 pour la Chaîne d'Octets
Longueur	4	28
X	4	Valeur du champ 'X'
NoOfY	4	2
Y.A	4	Valeur du champ 'Y[0].A'
Y.B	4	Valeur du champ 'Y[0].B'
Y.A	4	Valeur du champ 'Y[1].A'
Y.B	4	Valeur du champ 'Y[1].B'
Z	4	Valeur du champ 'Z'

5.2.6 Messages

Les *Messages* sont codés sous forme d'*Objets d'Extension*. Les paramètres inclus dans chaque *Message* sont sérialisés de la même manière que le sont les champs d'une *Structure*.

Le champ "*Identificateur de Type*" contient l'identificateur du *Codage de Type de Données* propre au *Message*. Le champ "*Longueur*" est omis étant donné que les *Messages* sont définis par cette série de normes OPC UA.

Chaque *Service* OPC UA décrit dans l'IEC 62541-4 a un *Message* de demande et de réponse. Les identificateurs du *Codage de Type de Données* attribués à chaque *Service* sont définis en A.3.

5.3 XML

5.3.1 Types intégrés

5.3.1.1 Généralités

La plupart des types intégrés sont codés en langage XML utilisant les formats définis dans la spécification de la XML Schema Part 2 Les restrictions ou utilisations spéciales sont traitées ci-dessous. Certains types intégrés comportent un schéma XML défini pour eux au moyen de la syntaxe explicitée dans la XML Schema Part 1.

Le préfixe *xs:* sert à désigner un symbole défini par la spécification du schéma XML.

5.3.1.2 Booléen

Une valeur Booléen est codée sous forme d'une valeur *xs:booléen* (*xs:boolean*).

5.3.1.3 Entier

Les valeurs entières sont codées avec l'un des sous-types du type *xs:décimal* (*xs:decimal)*. Les correspondances entre les types d'entier OPC UA et les types de données du schéma XML sont présentées dans le Tableau 17.

Tableau 17 – Correspondances des types de données XML pour des Entiers

Dénomination	Type XML
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 Virgule flottante

Les valeurs à virgule flottante sont codées avec l'un des types à virgule flottante XML. Les correspondances entre les types à virgule flottante OPC UA et les types de données du schéma XML sont présentées dans le Tableau 18.

Tableau 18 – Correspondances de types de données XML pour les virgules flottantes

Dénomination	Type XML
Float	xs:float
Double	xs:double

Le type à virgule flottante XML prend en charge la Chaîne "Infinité Positive" (INF), "Infinité négative"(-INF) et "Pas un nombre"(NaN)

5.3.1.5 Chaîne

Une valeur *Chaîne* est codée comme valeur *xs:chaîne* (*xs:string*).

5.3.1.6 Date&Heure

Une valeur *Date&Heure* est codée comme valeur *xs:date&Heure* (*xs:dateTime)*.

Toutes les valeurs Date&Heure doivent être codées sous forme de Temps UTC ou avec une spécification explicite du fuseau horaire.

Correct:

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

Incorrect:

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

Il est recommandé de représenter toutes les valeurs de *xs:date&Heure* sous format UTC.

La valeur date/heure la plus récente et la valeur date/heure la plus ancienne qui sont représentées sur une *Plate-forme de Développement* ont une signification particulière et ne doivent pas être codées strictement au format XML.

La valeur date/heure la plus récente sur une *Plate-forme de Développement* doit être codée au format XML suivant: '0001-01-01T00:00:00Z'.

La valeur de date/heure la plus ancienne sur une *Plate-forme de Développement* doit être codée au format XML suivant '9999-12-31T11:59:59Z'

Si un décodeur rencontre une valeur *xs:date&Heure* qui ne peut pas être représentée sur la *Plate-forme de Développement*, il convient qu'il convertisse la valeur en date&heure la plus récente ou la plus ancienne qui peut être représentée sur la *Plate-forme de Développement*. Il convient que le décodeur XML ne génère aucune erreur s'il rencontre une valeur de date en dehors des valeurs définies.

La valeur de date/heure la plus proche sur une *Plate-forme de Développement* est équivalente à une valeur date/heure nulle.

5.3.1.7 Guid (Identificateur globalement unique)

Un *Guid* est codé au moyen de la représentation de chaîne définie au 5.1.3.

Le schéma XML applicable à un *Guid* est:

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

5.3.1.8 Chaîne d'octets (ByteString)

Une valeur *Chaîne d'Octets* est codée comme une valeur *xs:Binaire en base 64* (*xs:base64Binary)* (voir Base64).

Le schéma XML applicable à une *Chaîne d'Octets* est:

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

5.3.1.9 Élément Xml (XmlElement)

Une valeur "*Elément Xml*" est codée comme une valeur *xs:Type Complexe (xs:complexType)* avec le schéma XML suivant:

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

Les éléments Xml peuvent être utilisés uniquement dans les valeurs "*Variante*" ou "*Objet d'Extension*".

5.3.1.10 Identificateur de Nœud (NodeId)

Une valeur *Identificateur de Nœud* est codée comme une valeur xs:chaîne (xs:string) avec la syntaxe suivante:

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

Les éléments de la syntaxe sont décrits dans le Tableau 19.

Tableau 19 – Composants de l'Identificateur de Nœud

Champ	Type de données	Description
<namespaceindex>	UInt16	*Indice d'espace de nom* formaté comme nombre en base 10. Si l'indice est 0, l'article entier 'ns=0;' doit être omis.
<type>	Enum	Fanion de spécification du *Type d'identificateur*. Le fanion a les valeurs suivantes: i NUMERIC (Entier Non Signé) s STRING (Chaîne) g GUID (Identificateur Globalement Unique) b OPAQUE (Chaîne d'Octets)
<value>	*	*Identificateur* codé comme chaîne. L'*identificateur* est formaté en utilisant la correspondance de type de données XML propre au *Type d'identificateur*. Noter que l'*identificateur* peut contenir tout caractère UTF8 non nul, y compris un blanc.

Exemples *d'Identificateurs de Nœud*:

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

Le schéma XML applicable à un *Identificateur de Nœud* est:

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

5.3.1.11 Identificateur de Nœud Etendu (ExpandedNodeId)

Une valeur *Identificateur de Nœud Etendu* est codée comme une valeur xs:chaîne avec la syntaxe suivante:

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

Les champs possibles sont indiqués dans le Tableau 20.

Tableau 20 – Composants de l'Identificateur de Nœud Etendu

Champ	Type de données	Description
<serverindex>	UInt32	*Indice du serveur* formaté comme nombre en base 10. Si l'*Indice du serveur* est 0, l'article entier 'svr=0;' doit alors être omis.
<namespaceindex>	UInt16	*Indice d'Espace de nom* formaté comme nombre en base 10 . Si l'*Indice d'Espace de nom* est 0, l'article entier 'ns=0;' doit alors être omis. Il ne doit pas y avoir d'*Indice d'Espace de nom* si l'URI est présent.
<uri>	Chaîne	*URI d'Espace de nom* formaté comme une chaîne. Les caractères réservés éventuels de l'URI doivent être remplacés par un '%' suivi de sa valeur ANSI de 8 bits codée sous forme de deux chiffres hexadécimaux (insensibles à la casse). Par exemple, le caractère ';' serait remplacé par '%3B'. Les caractères réservés sont ';'et '%'. Si l'*URI d'Espace de nom* est nul ou vide, l'article 'nsu=;' doit alors être omis.
<type>	Enum	Fanion de spécification du *Type d'identificateur*. Ce champ est décrit dans le Tableau 19 .
<value>	*	*Identificateur* codé sous forme de chaîne. Ce champ est décrit dans le Tableau 19.

Le schéma XML applicable à *Identificateur de Nœud Etendu* est:

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

5.3.1.12 Code de Statut (StatusCode)

Un *Code de Statut* est codé comme *xs:Entier Non Signé* (*xs:unsignedInt)* avec le schéma XML suivant:

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

5.3.1.13 Information de diagnostic (DiagnosticInfo)

Une valeur *Information de Diagnostic* est codée comme un *xs:Type Complexe* (*xs:complexType)* avec le schéma XML suivant:

```
<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="InnerCode  de  Statut"  type="tns:Code  de  Statut"
minOccurs="0" />
  <xs:element name="InnerDiagnosticInfo" type="tns:DiagnosticInfo"
   minOccurs="0" />
 </xs:sequence>
</xs:complexType>
```

5.3.1.14 Nom Qualifié (QualifiedName)

Une valeur *Nom Qualifié* est codée comme un *xs:Type Complexe* (*xs:complexType)* avec le schéma XML suivant:

```
<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 Texte Localisé (LocalizedText)

Une valeur *Texte Localisé* est codée comme un *xs:Type Complexe* (*xs:complexType)* avec le schéma XML suivant:

```
<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 Objet d'Extension (ExtensionObject)

Une valeur *Objet d'Extension* est codée comme un *xs:Type Complexe* (*xs:complexType*) avec le schéma XML suivant:

```
<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>
```

Le corps de l'*Objet d'Extension* contient un seul élément qui est soit une *Chaîne d'Octets*, soit une *Structure* à codage XML. Un décodeur peut faire la différence entre ces deux éléments en examinant l'élément de niveau supérieur. Un élément avec le nom tns:Chaîne d'Octets (tns:ByteString) contient un corps à codage OPC UA binaire. Tout autre nom doit contenir un corps à codage OPC UA XML.

L'*identificateur de Type* est l'*identificateur de Nœud* pour l'*Objet de Codage de Type de Données*.

5.3.1.17 Variante (Variant)

Une valeur *Variante* est codée comme un *xs:Type Complexe* (*xs:complexType)* avec le schéma XML suivant:

```
<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>
```

Si la *Variante* représente une valeur scalaire, elle doit alors contenir un seul élément enfant avec le nom du type intégré. Par exemple, la valeur à virgule flottante en simple précision 3,141 5 serait codée comme suit:

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

Si la *Variante* représente une matrice unidimensionnelle, elle doit alors contenir un seul élément enfant avec le préfixe “ListeDe” ('ListOf') et le type intégré du nom. Par exemple, une *Matrice* de chaînes serait codée comme suit:

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

Si la *Variante* représente une *Matrice* multidimensionnelle, elle doit alors contenir un élément enfant avec le nom ‘*Matrice*’ comportant les deux sous-éléments présentés dans cet exemple:

```
<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>
```

Dans l’exemple, la matrice comporte les éléments suivants:

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

Les éléments d’une *Matrice* multidimensionnelle sont toujours aplatis dans une *Matrice* unidimensionnelle dans laquelle les dimensions de rang supérieur sont sérialisées en premier. Cette *Matrice* unidimensionnelle est codée comme un enfant de l’élément ‘Eléments’. L’élément ‘Dimensions’ est une *Matrice* de valeurs Int32 qui spécifient les dimensions de la matrice en commençant par la dimension de rang inférieur. Les dimensions codées permettent de reconstituer la *Matrice* multidimensionnelle.

Le Tableau 1 donne l’ensemble complet des noms de types intégrés.

5.3.1.18 Valeur de Données (DataValue)

Une valeur *Valeur de Données* est codée comme un *xs:Type Complexe* (*xs:complexType)* avec le schéma XML suivant:

```
<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 Énumérations

Les *énumérations* utilisées comme paramètres dans les *Messages* définis dans l'IEC 62541-4 sont codées comme *xs:chaîne* (*xs:string)* avec la syntaxe suivante:

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

Les éléments de la syntaxe sont décrits dans le Tableau 21.

Tableau 21 – Composants d'Énumération

Champ	Type	Description
<symbol>	Chaîne	Nom symbolique de la valeur énumérée.
<value>	UInt32	Valeur numérique associée à la valeur énumérée.

Par exemple, le schéma XML applicable à l'énumération de *Classe de Nœuds* (*NodeClass)* est:

```
<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>
```

Les *énumérations* mémorisées dans une *Variante* sont codées comme une valeur *Int32*.

Par exemple, toute *Variable* peut avoir une valeur avec un *Type de Données* de *Classe de Nœud.* Dans ce cas, la valeur numérique correspondante est placée dans la *Variante* (par exemple, *Classe de Nœud::Objet* (*NodeClass*::*Object)* serait mémorisée comme 1).

5.3.3 Matrices

Les paramètres de *Matrices* sont toujours codés en enveloppant des éléments dans un élément conteneur et en insérant ce dernier dans la structure. Il convient que le nom de l'élément conteneur soit le nom du paramètre. Le nom de l'élément dans la matrice doit être le nom de type.

Par exemple, le service *Lecture* utilise une matrice d'*Identificateurs de Valeurs de Lecture* (*ReadValueIds).* Le schéma XML serait le suivant:

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

Les attributs nillables (présents mais vides) doivent être spécifiés car les codeurs XML placent les éléments dans les matrices si ces éléments sont vides.

5.3.4 Structures

Les structures sont codées comme un *xs:Type Complexe (xs:complexType),* tous les champs apparaissant de manière séquentielle. Tous les champs sont codés comme un *xs:élément* (*xs:element*) et la valeur *xs:maxOccurs* est fixée à 1.

Par exemple, la demande du service "Lecture" a une structure d'*Identificateur de Valeur de Lecture*. Le schéma XML serait le suivant:

```
<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

Les *Messages* sont codés comme un *xs:Type Complexe (xs:complexType)*. Les paramètres de chaque *Message* sont sérialisés de la même manière que le sont les champs d'une *Structure*.

6 Protocoles de sécurité des messages

6.1 Protocole d'établissement de liaison de sécurité

Tous les *Protocoles de Sécurité* doivent mettre en œuvre les services *Ouverture de Canal Sécurisé* et *Fermeture de Canal Sécurisé* définis dans l'IEC 62541-4. Ces *Services* précisent comment établir un *Canal Sécurisé* et comment assurer la sécurité des *Messages* échangés sur ce *Canal Sécurisé.* Les *Messages* échangés et les algorithmes de sécurité qui leur sont appliqués sont présentés à la Figure 10.

Les *Protocoles de Sécurité* doivent prendre en charge trois *Modes de Sécurité*: *Aucun (None*), *Signature (Sign*) et *Signature et Chiffrement (SignAndEncrypt*). Si le *Mode de Sécurité* est *Aucun*, aucune sécurité n'est alors appliquée, et le protocole d'établissement de liaison de sécurité présenté à la Figure 10 n'est pas exigé. Cependant, la mise en œuvre d'un *Protocole de Sécurité* doit toujours maintenir un canal logique et fournir un identificateur unique pour le *Canal Sécurisé.*

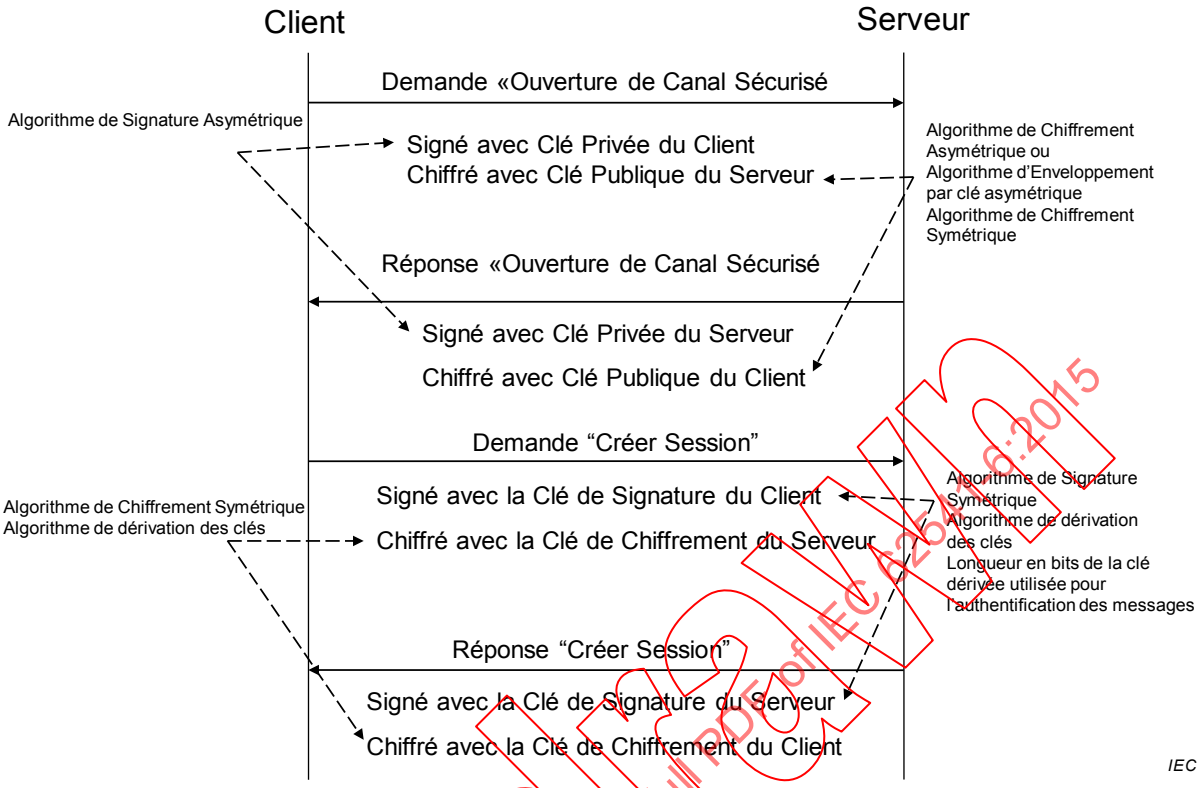

Figure 10 – Protocole d'établissement de liaison de sécurité

Chaque correspondance du *Protocole de sécurité* spécifie exactement comment appliquer les algorithmes de sécurité au *Message*. Un ensemble d'algorithmes de sécurité qui doivent être utilisés lors de l'établissement d'une liaison de sécurité est appelé *Politique de Sécurité*. L'IEC 62541-7 définit les *Politiques de Sécurité* normalisées comme partie intégrante des *Profils* normalisés que les applications OPC UA sont supposées prendre en charge. L'IEC 62541-7 définit également un URI pour chaque *Politique de Sécurité* normalisée.

Une *Pile* est supposée bien connaître les *Politiques de Sécurité* qu'elle prend en charge. Les *Applications* spécifient la *Politique de Sécurité* qu'elles souhaitent utiliser en transférant l'URI à la *Pile*.

Le Tableau 22 définit le contenu d'une *Politique de Sécurité*. Chaque *Correspondance de Protocole de Sécurité* spécifie comment utiliser chacun des paramètres de la *Politique de Sécurité*. Une correspondance *de Protocole de Sécurité* peut ne pas utiliser tous les paramètres.

Tableau 22 – Politique de Sécurité

Dénomination	Description
PolicyUri	URI attribué à la *Politique de Sécurité*.
SymmetricSignatureAlgorithm	URI de l'algorithme de signature symétrique à utiliser.
SymmetricEncryptionAlgorithm	URI de l'algorithme de chiffrement à clé symétrique à utiliser.
AsymmetricSignatureAlgorithm	URI de l'algorithme de signature asymétrique à utiliser.
AsymmetricKeyWrapAlgorithm	URI de l'algorithme d'enveloppement à clé asymétrique à utiliser.
AsymmetricEncryptionAlgorithm	URI d'algorithme de chiffrement à clé asymétrique à utiliser.
MinAsymmetricKeyLength	Longueur minimale pour une clé asymétrique
MaxAsymmetricKeyLength	Longueur maximale pour une clé asymétrique
KeyDerivationAlgorithm	Algorithme de dérivation par clé à utiliser.
DerivedSignatureKeyLength	Longueur en bits de la clé dérivée utilisée pour l'authentification des *messages*.

L'*Algorithme de Chiffrement Asymétrique (AsymmetricEncryptionAlgorithm)* est utilisé pour le chiffrement du *Message* complet à l'aide d'une clé asymétrique. Certains *Protocoles de Sécurité* ne chiffrent pas le *Message* complet à l'aide d'une clé asymétrique. Ils utilisent en revanche l'A*lgorithme d'Enveloppement à Clé Asymétrique* pour chiffrer une clé symétrique, puis utilisent l'*Algorithme de Chiffrement Symétrique* pour chiffrer le *Message*.

L'*Algorithme de Signature Asymétrique* (*AsymmetricSignatureAlgorithm)* permet de signer un *Message* à l'aide d'une clé asymétrique.

L'*Algorithme de Dérivation par Clé* (*KeyDerivationAlgorithm)* permet de créer les clés utilisées pour sécuriser les *Messages* transmis sur le *Canal Sécurisé*. La longueur des clés de chiffrement est déterminée par l'*Algorithme de Chiffrement Symétrique (SymmetricEncryptionAlgorithm)*. La longueur des clés permettant de créer des *Signatures Symétriques* dépend de l'A*lgorithme de Signature Symétrique* (*SymmetricSignatureAlgorithm)* et peut être différente de la longueur des clés de chiffrement.

6.2 Certificats

6.2.1 Généralités

Les *Applications* OPC UA utilisent des *Certificats* pour l'archivage des *Clés Publiques* nécessaires aux opérations de *Cryptographie Asymétrique*. Tous les *Protocoles de Sécurité* utilisent des *Certificats* X509 Version 3 (voir X509) codés au format DER (voir X690). Les *Certificats* utilisés par les *Applications* OPC UA doivent également être conformes au RFC 3280 qui définit un profil pour les *Certificats* X509 lorsqu'ils sont utilisés comme partie intégrante d'une *Application* Web.

Les paramètres *Certificat Serveur* et *Certificat Client* utilisés dans le service abstrait *Ouverture de Canal Sécurisé* sont des instances du *Type de Données Certificat d'Instance d'Application.* Le paragraphe 6.2.2 décrit comment créer un *Certificat* X509 qui peut être utilisé comme *Certificat d'Instance d'Application.*

Les paramètres *Certificats de Logiciels Serveur* et *Certificats de Logiciels Client* des *Services* abstraits *Créer Session* et *Activer Session* sont des instances du *Type de Données Certificat de Logiciel Signé*. Le paragraphe 6.2.3 décrit comment créer un *Certificat* X509 qui peut être utilisé comme *Certificat de Logiciel Signé.*

6.2.2 Certificat d'instance d'application

Un *Certificat d'Instance d'Application* est une *Chaîne d'Octets* contenant la forme de codage DER (voir X690) d'un *Certificat* X509v3. Ce *Certificat,* délivré par l'autorité de certification, identifie une instance d'une *Application* fonctionnant sur un hôte unique. Les champs X509v3 contenus dans un *Certificat d'Instance d'Application* sont décrits dans le Tableau 23. Les champs sont définis entièrement dans le RFC 3280.

Le Tableau 23 fournit également une correspondance entre les termes du RFC 3280 et les termes utilisés dans la définition abstraite d'un *Certificat d'Instance d'Application* défini dans l'IEC 62541-4.

Tableau 23 – Certificat d'Instance d'Application

Dénomination	Nom de paramètre donné dans la partie 4	Description
ApplicationInstanceCertificate		Certificat X509v3
version	version	Doit être «V3»
serialNumber	Numéro de série	Numéro de série attribué par l'émetteur.
signatureAlgorithm	Algorithme de signature	Algorithme utilisé pour la signature du certificat.
signature	signature	Signature créée par l'émetteur.
issuer	émetteur	Nom distinctif du *Certificat* utilisé pour créer la signature. Le champ *émetteur* est décrit intégralement dans le RFC 3280.
validity	valide Jusqu'à, valide Depuis	Date de début et de fin de validité du *Certificat*.
subject	sujet	Nom distinctif de l'I*nstance d'Application*. L'attribut Nom Commun doit être spécifié. Il convient par ailleurs qu'il soit le *Nom de produit* ou un équivalent approprié. L'attribut Nom d'Organisme doit être le nom de l'Organisme qui exécute l'I*nstance d'Application*. Cet organisme n'est habituellement pas le fournisseur de l'*Application*. D'autres attributs peuvent être spécifiés. Le champ *sujet* est décrit intégralement dans le RFC 3280.
subjectAltName	Uri d'application, noms d'hôtes	Pseudonymes de *l'Instance d'Application*. Doit inclure un Identificateur de Ressource uniforme équivalent à l'*Uri d'application*. Les *serveurs* doivent spécifier un nom de domaine (dNSName) ou une adresse IP qui identifie la machine sur laquelle fonctionne *l'Instance d'Application*. Des noms de domaine complémentaires peuvent être spécifiés si la machine a plusieurs noms. Il convient de ne pas préciser d'adresse IP si le *Serveur* a un nom de domaine. Le champ *pseudonyme de Sujet* est décrit intégralement dans le RFC 3280.
publicKey	Clé publique	Clé publique associée au *Certificat*.
keyUsage	Utilisation de la clé	Spécifie comment la clé de *Certificat* peut être utilisée. Doit inclure la Signature numérique, la non-Répudiation, le chiffrement par clé et le chiffrement de données. Il est admis d'utiliser d'autres clés.
extendedKeyUsage	Utilisation de la clé	Spécifie les utilisations de clé supplémentaires pour le *Certificat*. Doit préciser «Autorisation Serveur» et/ou Autorisation Client. Il est admis d'utiliser d'autres clés.
authorityKeyIdentifier		Fournit davantage d'informations sur la clé utilisée pour la signature du *Certificat*. Doit être spécifié pour les *Certificats* signés par une autorité de certification. Il convient de le spécifier pour les *Certificats* autosignés.

6.2.3 Certificat de logiciel signé

Un *Certificat de Logiciel Signé* est une *Chaîne d'Octets* contenant la forme de codage DER d'un certificat X509v3. Ce *Certificat*, délivré par une autorité de certification, contient une extension X509v3 avec le *Certificat de Logiciel* qui spécifie les revendications vérifiées par l'autorité de certification. Les champs X509v3 contenus dans un *Certificat de Logiciel Signé* sont décrits dans le Tableau 24. Les champs sont définis entièrement dans le RFC 3280.

Tableau 24 – Certificat de Logiciel Signé

Dénomination		Description
SignedSoftwareCertificate		Certificat X509v3
version	version	Doit être «V3»
SerialNumber	Numéro de série	Numéro de série attribué par l'émetteur.
SignatureAlgorithm	Algorithme de signature	Algorithme utilisé pour la signature du *Certificat*.
signature	signature	Signature créée par l'émetteur.
issuer	émetteur	Nom distinctif du *Certificat* utilisé pour créer la signature. Le champ *émetteur* est décrit intégralement dans le RFC 3280.
validity	valide Jusqu'à, valide Depuis	Date de début et de fin de validité du *Certificat*.
subject	sujet	Nom distinctif du produit. L'attribut Nom Commun doit être le même que le *Nom de Produit* dans le *Certificat de Logiciel* et l'attribut Nom de l'Organisme doit être le *Nom du Fournisseur* dans ledit certificat. D'autres attributs peuvent être spécifiés. Le champ *sujet* est décrit intégralement dans le RFC 3280.
subjectAltName	Uri du produit	Pseudonymes du produit. Doit inclure un identificateur de ressources uniforme équivalent à *l'Uri du produit* spécifié dans le *Certificat de Logiciel*. Le champ *pseudonyme de Sujet* est décrit intégralement dans le RFC 3280.
publicKey	Clé publique	Clé publique associée au *Certificat*.
keyUsage	Utilisation de la clé	Spécifie comment la clé de *Certificat* peut être utilisée. Doit être la Signature numérique et la non-répudiation Il n'est pas admis d'utiliser d'autres clés.
extendedKeyUsage	Utilisation de la clé	Spécifie les utilisations de clé supplémentaires pour le *Certificat*. Peut spécifier la Signature de code Il n'est pas admis d'utiliser d'autres clés.
softwareCertificate	Certificat de logiciel	Forme codée XML du *Certificat de Logiciel* archivé comme texte UTF8. Le paragraphe 5.3.4 décrit la méthode de codage d'un *Certificat de Logiciel* au format XML. L'Identificateur d'Objet (OID) ASN.1 pour cette extension est: 1.2.840.113556.1.8000.2264.1.6.1

6.3 Synchronisation horaire

Tous les *Protocoles de Sécurité* exigent une synchronisation raisonnable des horloges système des machines de communication afin de vérifier les délais d'expiration des *Certificats* ou des *Messages*. L'écart horaire qui peut être accepté dépend des exigences de sécurité des systèmes et les *Applications* doivent permettre aux administrateurs de configurer l'écart horaire acceptable lors de la vérification des horaires. Un écart de 5 minutes représente une valeur par défaut appropriée.

Le Protocole de Synchronisation Réseau (NTP) prévoit une méthode de synchronisation normalisée d'une horloge machine avec un serveur horaire sur le réseau. Les systèmes qui fonctionnent sur une machine avec un système d'exploitation complet tel que Windows ou Linux prennent déjà en charge le protocole NTP ou un protocole équivalent. Il convient que les appareils qui utilisent des systèmes d'exploitation intégrés prennent en charge le protocole NTP.

Lorsqu'un système fonctionnant avec un appareil ne peut pas en réalité prendre en charge un protocole NTP, une *Application* OPC UA peut alors utiliser les *Horodatages* de l'*En-tête de Réponse* (*ResponseHeader)* (voir l'IEC 62541-4) pour synchroniser son horloge. Dans ce scénario, il faut que l'*Application* OPC UA connaisse l'URL applicable à un *Serveur de Découverte* (*Discovery Server)* sur une machine dont on sait qu'elle est correctement réglée (bon horodatage). L'*Application* OPC UA ou un service de référence distinct appelle le *Service Trouver des Serveurs* (*FindServers)* et règle son horloge sur l'heure spécifiée dans l'*En-tête de Réponse*. Il est nécessaire de répéter ce processus de façon périodique en raison du décalage des horloges avec le temps.

6.4 Temps universel coordonné (UTC) et Temps atomique international (TAI)

Tous les horaires de l'OPC UA sont en temps UTC; toutefois, le temps UTC peut comporter des interruptions dues aux secondes intercalaires ou à la répétition des secondes, auxquelles s'ajoute la gestion des variations tant au niveau de l'orbite que de la rotation terrestres. Les *Serveurs* qui ont accès à la source pour le Temps atomique international (TAI) peuvent choisir d'utiliser ce système au lieu du temps UTC. Cela dit, il faut que les *Clients* soient toujours prêts à gérer les interruptions dues au temps UTC ou tout simplement du fait que l'horloge système est réglée sur la machine du *Serveur*.

6.5 Jetons d'identité d'utilisateur émis – Jetons Kerberos

Les *Jetons d'Identité d'Utilisateur* (*UserIdentityTokens)* Kerberos peuvent être transmis au *Serveur* à l'aide du J*eton d'identité Emis* (*IssuedIdentityToken)*. Le corps du jeton est un élément XML qui contient le jeton de sécurité WS tel que défini dans la spécification de Profil de Jeton Kerberos (Kerberos).

Les *Serveurs* qui prennent en charge l'authentification Kerberos doivent prévoir une *Politique pour le Jeton Utilisateur* (*UserTokenPolicy)* qui spécifie quelle version du Profil de Jeton Kerberos est effectivement utilisée, le domaine Kerberos et la dénomination principale Kerberos applicables au *Serveur*. Le domaine et la dénomination principale sont combinés par une syntaxe simple et placés dans l'*Uri du Point d'extrémité de l'émetteur* (*issuerEndpointUri)* comme indiqué dans le Tableau 25.

Tableau 25 – Politique pour le Jeton Utilisateur (UserTokenPolicy) Kerberos

Dénomination	Description
tokenType	ISSUEDTOKEN_3
issuedTypeType	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1
issuerEndpointUri	Chaîne sous la forme \\<realm>\<server principal name> où <realm> est le nom de domaine Kerberos (par exemple, Domaine Windows); <server principal name> est la dénomination principale Kerberos pour le Serveur OPC UA.

L'interface entre les applications *Client* et *Serveur* et le Service d'authentification Kerberos est spécifique à l'application. Le domaine correspond au Nom de domaine (DomainName) lorsque l'on utilise un contrôleur de Domaine Windows comme fournisseur Kerberos.

6.6 Conversation sécurisée WS

6.6.1 Vue d'ensemble

Tout *Message* transmis par protocole SOAP peut être sécurisé avec la Conversation sécurisée WS (WS Secure Conversation). Ce protocole spécifie une méthode de négociation des secrets partagés via la Confiance WS (WS Trust), puis utilise ces secrets pour sécuriser les *Messages* échangés à l'aide des mécanismes définis dans la Sécurité WS (WS Security).

La spécification de signature XML (XML Signature) décrit les mécanismes de signature effective des éléments XML. La spécification de chiffrement XML (XML Encryption) décrit les mécanismes de chiffrement des éléments XML.

La politique de sécurité WS (WS Security Policy) définit les suites algorithmiques normalisées qui peuvent être utilisées pour sécuriser les *Messages* SOAP. Ces suites algorithmiques mettent directement en correspondance les *Politiques de Sécurité* définies dans l'IEC 62541-7. Le Profil de sécurité de base WS-I 1.1 (WS-I Basic Security Profile 1.1) définit les meilleures pratiques applicables à la sécurité WS qui permettent d'assurer l'interopérabilité. Toutes les mises en œuvre d'OPC UA doivent être conformes à cette spécification.

L'en-tête *Horodatage* défini par Sécurité WS (WS Security) permet d'éviter la réitération des attaques et doit être présent et signé dans tous les *Messages* échangés.

La Figure 11 illustre la relation entre les différentes spécifications WS-* utilisées par cette correspondance. Les versions des spécifications WS-* présentées dans le diagramme constituaient les versions les plus courantes au moment de la publication. L'IEC 62541-7 peut définir des *Profils* qui nécessitent une prise en charge pour les versions futures de ces spécifications.

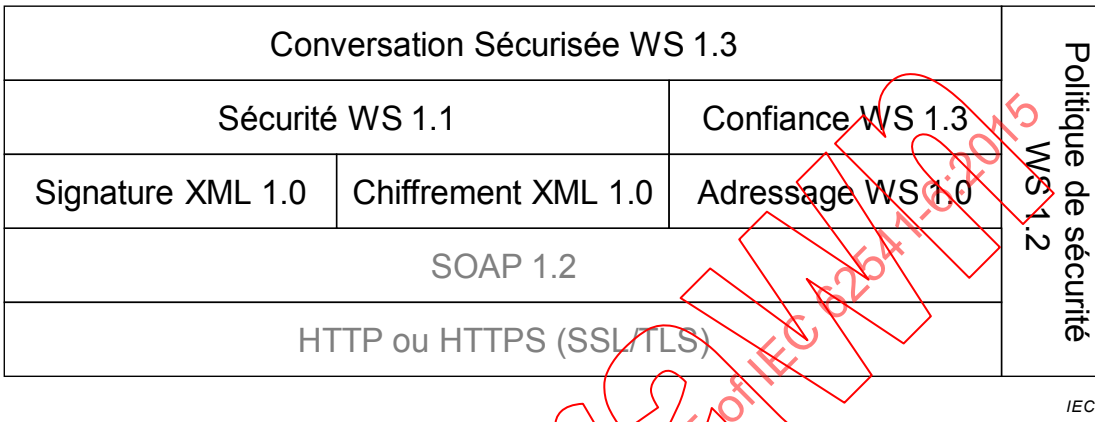

Figure 11 – Spécifications appropriées des services Web XML

La Figure 12 illustre la méthode d'utilisation de ces spécifications WS-* dans le protocole d'établissement de liaison de sécurité.

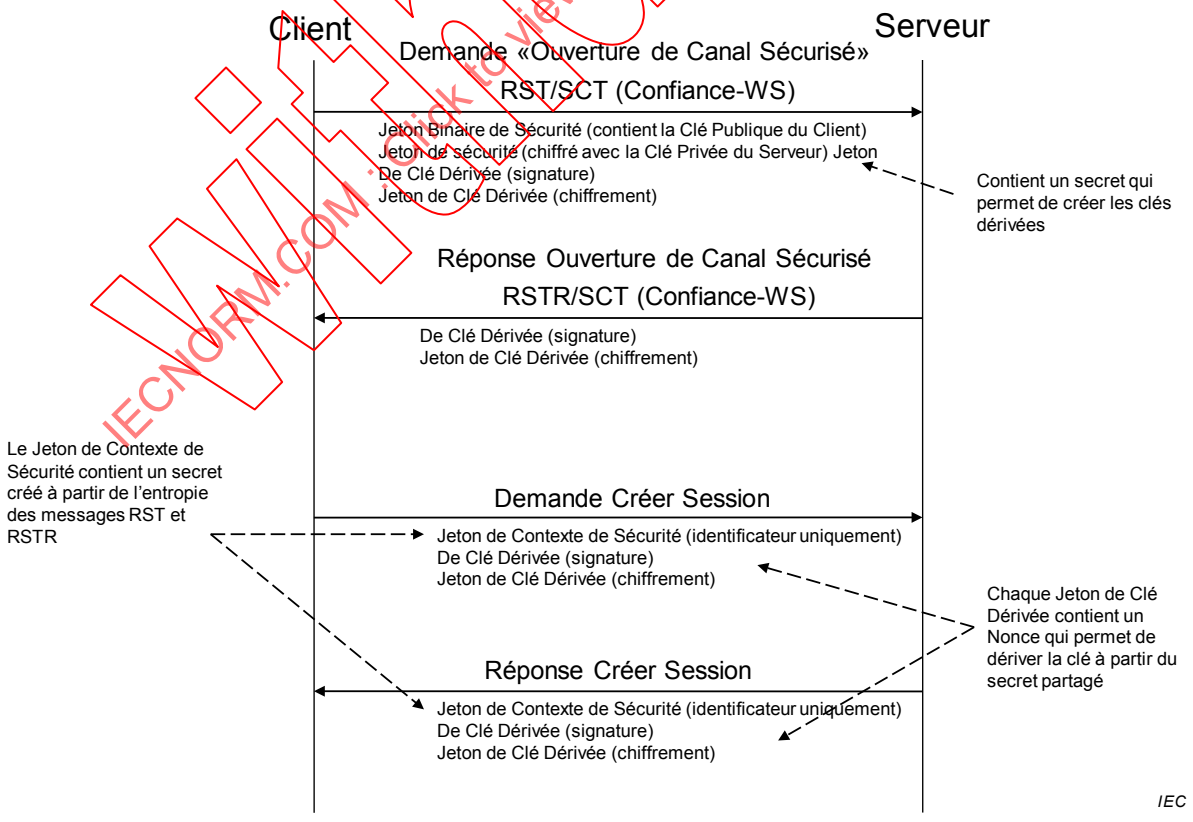

Figure 12 – Protocole d'établissement de liaison de Conversation sécurisée WS

Les *Messages* RST (Demande de jeton de sécurité) et RSTR (Réponse à une demande de jeton de sécurité) sont définis par Confiance WS (WS Trust). La Conversation sécurisée WS (WS Secure Conversation) définit de nouvelles actions pour ces *Messages* qui indiquent au *Serveur* que le client souhaite créer un SCT (Jeton de contexte de sécurité). Le SCT comporte les clés partagées que les *Applications* utilisent pour sécuriser les *Messages* transmis sur le *Canal Sécurisé*.

Les *Messages* individuels sont sécurisés à l'aide de clés issues du SCT utilisant le mécanisme défini dans Conversation sécurisée WS (WS Secure Conversation). Les paragraphes ci-dessous spécifient la structure des *Messages* individuels et définissent les fonctionnalités issues des spécifications WS* nécessaires à la mise en œuvre du protocole d'établissement de liaison de sécurité OPC. UA

6.6.2 Notation

Les *Messages* SOAP utilisent les éléments XML définis dans un certain nombre de spécifications différentes. Ce document utilise les préfixes mentionnés dans le Tableau 26 pour identifier la spécification qui définit un élément XML.

Tableau 26 – Préfixes d'Espace de nom WS-*

Préfixe	Spécification
wsu	Services de sécurité WS
wsse	Extensions de sécurité WS
wst	Confiance WS
wsc	Conversation sécurisée WS
wsa	Adressage WS
xenc	Chiffrement XML

6.6.3 Demande de jeton de sécurité (RST/SCT)

Le *Message* de demande de jeton de sécurité met en œuvre le *message* de demande abstrait d'*Ouverture de Canal Sécurisé* défini dans l'IEC 62541-4. La syntaxe de ce *Message* est définie par une Confiance WS (WS Trust). La structure du *Message* est décrite en détail dans Conversation Sécurisée WS (WS Secure Conversation.

Ce *Message* doit avoir les jetons suivants:

a) Un wsse:Jeton de sécurité binaire (wsse:BinarySecurityToken) contenant la *Clé Publique du Client*. La *Clé Publique* est transmise dans un *Certificat* X509v3 à codage DER.

b) Un wsse:Jeton de sécurité (wsse:SecurityToken) chiffré contenant le *Nonce* (Nombre à usage unique) du *Client* utilisé pour dériver les clés. Ce *Jeton de sécurité* doit être chiffré avec l'*Algorithme d'Enveloppement à Clé Asymétrique* et la *Clé Publique* associée au *Certificat d'instance d'application du serveur*.

c) Un wsc:Jeton de Clé Dérivée (wsc:DerivedKeyToken) utilisé pour la signature du corps, les en-têtes WS Addressing et l'en-tête wsu:Horodatage (wsu:Timestamp) qui utilisent l'*Algorithme de Signature Symétrique*. L'élément de signature doit alors être signé avec l'*Algorithme de Signature Asymétrique* avec la *Clé privée du client*. Le jeton wsc:Jeton de Clé Dérivée doit également spécifier un Nonce.

d) Un wsc:Jeton de Clé Dérivée ((wsc:DerivedKeyToken)) utilisé pour chiffrer le corps du *Message* avec l'*Algorithme de Chiffrement Symétrique*.

Ce *Message* doit comporter les en-têtes wsa:Action, wsa:Identificateur de message (wsa:MessageId), wsa:Répondre A (wsa:ReplyTo) et wsa:A destination de (wsa:To) définis par WS Addressing. Le *Message* doit également comporter l'en-tête wsu:Horodatage défini par WS Security. Ces en-têtes doivent également être signés avec la clé dérivée utilisée pour signer le corps du *Message*.

La signature doit être calculée avant tout chiffrement, puis doit être chiffrée.

Les correspondances entre les paramètres de demande «*Ouverture de Canal Sécurisé*» et les éléments du *Message* RST/SCT sont présentées dans le Tableau 27

Tableau 27 – Correspondance RST/SCT avec une demande Ouverture de Canal Sécurisé

Paramètre d'Ouverture de Canal Sécurisé	Élément ST/SCT	Description
Certificat client	wsse:BinarySecurityToken	Transmis dans l'en-tête SOAP
Type de demande	wst:RequestType	Doit se présenter sous la forme "http://schemas.xmlsoap.org/ws/2005/02/trust/Issue" lors de la création d'un nouvel élément SCT. Doit se présenter sous la forme "http://schemas.xmlsoap.org/ws/2005/02/trust/Renew" lors du renouvellement d'un élément SCT.
Identificateur de Canal sécurisé	wsse:SecurityTokenReference	Transmis dans l'en-tête SOAP lors du renouvellement d'un élément SCT.
Mode de sécurité Uri de Politique de Sécurité	wst:SignatureAlgorithm wst:EncryptionAlgorithm wst:KeySize	Ces éléments décrivent la *Politique de Sécurité* demandée par le *Client*. Ces éléments doivent correspondre à la *Politique de Sécurité* appliquée par le *Point d'extrémité* auquel le *Client* souhaite se connecter. Ces éléments sont facultatifs.
Nonce du Client	wst:Entropy	Contient le *Nonce* spécifié par le *Client*. Le *Nonce* est spécifié avec l'élément wst:Secret Binaire.
durée de Vie utile requise	wst:Lifetime	Durée de vie utile requise pour l'élément SCT. Cet élément est facultatif.

6.6.4 Réponse à la Demande de jeton de sécurité (RSTR/SCT)

Le *Message* de réponse à la demande de jeton de sécurité met en œuvre le *Message* abstrait de réponse *d'Ouverture de Canal Sécurisé* défini dans l'IEC 62541-4. La syntaxe de ce *Message* est définie par Confiance WS (WS Trust). L'utilisation du *Message* est décrite en détail dans Conversation Sécurisée WS (WS Secure Conversation. Ce *Message* n'est pas signé ou chiffré avec les algorithmes asymétriques décrits dans l'IEC 62541-4. Les algorithmes symétriques et une clé fournie dans le *Message* de demande sont utilisés en lieu et place.

Ce *Message* doit avoir les jetons suivants:

a) Un wsc:Jeton de Clé Dérivée wsc:DerivedKeyToken) utilisé pour la signature du corps, les en-têtes d'WS Addressing et l'en-tête wsu:Horodatage qui utilisent l'*Algorithme de Signature Symétrique*. Cette clé est dérivée du *Jeton de Sécurité* chiffré spécifié dans le *Message* RST/SCT. Le jeton wsc:Jeton de Clé Dérivée doit également spécifier un *Nonce*.

b) Un wsc:Jeton de Clé Dérivée utilisé pour chiffrer le corps du *Message* avec l'Algorithme de Chiffrement Symétrique. Cette clé est dérivée du *Jeton de Sécurité* chiffré spécifié dans le *Message* RST/SCT. Le jeton wsc:Jeton de Clé Dérivée doit également spécifier un *Nonce*.

Ce *Message* doit comporter les en-têtes wsa:Action et wsa:Relatif à (wsa:RelatesTo) définis par WS Addressing). Le *Message* doit également comporter l'en-tête wsu:Horodatage défini par WS Security. Ces en-têtes doivent également être signés avec la clé dérivée utilisée pour signer le corps du *Message*.

La signature doit être calculée avant tout chiffrement, puis doit être chiffrée.

La correspondance entre les paramètres de réponse d'*Ouverture de Canal Sécurisé* et les éléments du *Message* RST/SCT sont présentés dans le Tableau 28.

Tableau 28 – Correspondance RSTR/SCT avec une Réponse d'Ouverture de Canal Sécurisé

Paramètre d'Ouverture de Canal Sécurisé	Elément RSTR/SCT	Description
---	wst:RequestedProofToken	Contient un élément wst:Clé Calculée (wst:ComputedKey) qui spécifie l'algorithme servant au calcul de la clé à secret partagée à partir des *Nonces* fournis par le *Client* et le *Serveur*.
---	wst:TokenType	Spécifie le type de J*eton de sécurité* émis.
Jeton de sécurité	wst:RequestedSecurityToken	Spécifie le nouveau SCT (Jeton de contexte de sécurité) ou le SCT renouvelé.
Identificateur de canal	wsc:Identifier	URI absolu d'identification du SCT.
Identificateur de jeton	wsc:Instance	Identificateur d'un ensemble de clés émis pour un contexte donné. Il doit être unique dans le contexte.
créé A	wsu:Created	Élément facultatif dans le jeton wsc:Jeton de contexte de sécurité renvoyé dans l'en-tête.
Durée de vie utile révisée	wst:Lifetime	Durée de vie utile révisée du SCT.
Nonce du serveur	wst:Entropy	Contient le *Nonce* spécifié par le *Serveur*. Le *Nonce* est spécifié avec l'élément wst:Secret Binaire. L'élément xenc:Données Chiffrées n'est pas utilisé dans OPC UA car le corps du *Message* doit être chiffré

La durée de vie spécifie le temps d'expiration UTC pour le jeton de contexte de sécurité. Le *Client* doit renouveler le SCT avant ce temps d'expiration par un nouvel envoi du *Message* RST/SCT. Le comportement exact est décrit dans l'IEC 62541-4.

6.6.5 Utilisation du SCT

Une fois que le C*lient* reçoit le *Message* RSTR/SCT, il peut utiliser le SCT pour sécuriser tous les autres *Messages*.

Un identificateur pour le SCT utilisé doit être transmis comme un wsc:Jeton de Contexte de Sécurité (wsc:SecurityContextToken) dans chaque *Message* de demande. Le *Message* de réponse doit référencer le *Jeton de Contexte de Sécurité* utilisé dans la demande.

Si on utilise un chiffrement, il doit être appliqué avant de calculer la signature.

Tout *Message* sécurisé avec le *Jeton de contexte de sécurité* doit comporter les jetons supplémentaires suivants:

a) Un wsc:Jeton de Clé Dérivée (wsc:DerivedKeyToken) utilisé pour la signature du corps, les en-têtes d'WS Addressing) et l'en-tête wsu:Horodatage qui utilisent l'*Algorithme de Signature Symétrique*. Cette clé est dérivée du *Jeton de Contexte de Sécurité*. Le jeton wsc:Jeton de Clé Dérivée doit également spécifier un *Nonce*.

b) Un wsc:Jeton de Clé Dérivée utilisé pour chiffrer le corps du *Message* avec l'*Algorithme de Chiffrement Symétrique*. Cette clé est dérivée du *Jeton de Contexte de Sécurité*. Le jeton wsc:Jeton de Clé Dérivée doit également spécifier un *Nonce*.

Ce *Message* doit comporter les en-têtes wsa:Action et wsa:Relatif à définis par WS Addressing). Le *Message* doit également comporter l'en-tête wsu:Horodatage défini par WS Security.

6.6.6 Annulation des contextes de sécurité

Le *Message d'Annulation* défini par WS Trust met en œuvre le *Message* de demande abstrait de *Fermeture de Canal Sécurisé* défini dans l'IEC 62541-4.

Ce *Message* doit être sécurisé avec l'élément SCT.

6.7 Conversation OPC UA sécurisée

6.7.1 Vue d'ensemble

Une conversation OPC UA sécurisée (UASC) est une version binaire de Conversation Sécurisée WS. Elle permet une communication sécurisée pour les transports qui n'utilisent ni le protocole SOAP, ni le langage XML.

La conversation UASC est conçue pour fonctionner avec différents *Protocoles de Transport* dont les capacités de mémoire tampon peuvent être limitées. C'est la raison pour laquelle la Conversation OPC UA sécurisée répartit les *Messages* OPC UA en plusieurs éléments (appelés «*Blocs de Messages*» ou '*MessageChunks*') de taille inférieure à la capacité de mémoire tampon admise par le *Protocole de Transport.* La Conversation UASC requiert une capacité de mémoire tampon du *Protocole de Transport* d'au moins 8 196 octets.

Une sécurité totale est appliquée aux *Blocs de Messages* individuels et non au *Message* OPC UA complet. Une *Pile* qui met en œuvre une conversation UASC est chargée de vérifier la sécurité de chaque *Bloc de Messages* reçu et de reconstituer le *Message* OPC UA d'origine.

Une séquence de 4 octets est attribuée à tous les *Blocs de Messages*. Ces numéros de séquence permettent de détecter et d'éviter la réitération des attaques.

Une conversation UASC nécessite un *Protocole de Transport* qui maintient l'ordre des *Blocs de Messages*. Cependant, une mise en œuvre UASC ne traite pas nécessairement les *Messages* dans leur ordre de réception d'origine.

6.7.2 Structure des Blocs de Messages

La Figure 13 présente la structure d'un *Bloc de Messages* et la méthode d'application de la sécurité au *Message*.

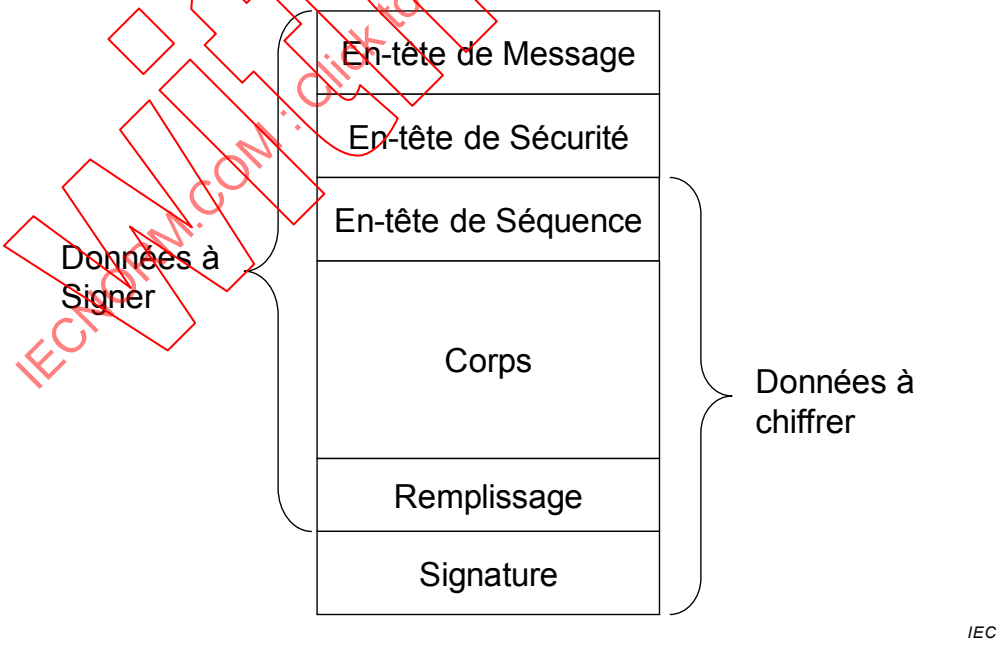

Figure 13 – Bloc de Messages de Conversation sécurisée OPC UA

Chaque *Bloc de Messages* comporte un en-tête comprenant les champs définis dans le Tableau 29.

Tableau 29 – En-tête de message de Conversation OPC UA Sécurisée

Dénomination	Type de données	Description
MessageType	Octet[3]	Code ASCII à trois octets qui identifie le type de *Message*. Les valeurs suivantes sont définies à ce moment: MSG *Message* sécurisé à l'aide des clés associées à un canal. OPN *Message* Ouverture de Canal Sécurisé. CLO *Message* Fermeture de Canal Sécurisé.
IsFinal	Octet	Code ASCII à un octet qui indique si le *Bloc de Messages* est le bloc final dans un *Message*. Les valeurs suivantes sont définies à ce moment: C Bloc intermédiaire. F Bloc final. A Bloc final (utilisé en cas d'erreur et lorsque le *Message* est abandonné).
MessageSize	UInt32	Longueur du *Bloc de Messages*, en octets. Cette valeur inclut la taille de l'en-tête de *Message*.
SecureChannelId	UInt32	Identificateur unique du *Canal Sécurisé* attribué par le *Serveur*. Si un *Serveur* reçoit un Identificateur de Canal Sécurisé qu'il ne reconnaît pas, il doit renvoyer une erreur de couche de transport appropriée. Au démarrage du *Serveur,* il convient que le premier *Identificateur de canal sécurisé* soit une valeur susceptible d'être unique après chaque redémarrage. Ceci garantit que le redémarrage du *Serveur* ne conduit pas les *Clients* connectés précédemment à" réutiliser" de manière fortuite des *Canaux sécurisés* qui ne leur appartenaient pas.

L'en-tête de *Message* est suivi d'un en-tête de sécurité qui spécifie les opérations de cryptographie effectivement appliquées au *Message*. Il existe deux versions de l'en-tête de sécurité qui dépendent du type de sécurité appliqué au *Message*. L'en-tête de sécurité utilisé pour les algorithmes asymétriques est défini dans le Tableau 30. Les algorithmes asymétriques permettent de sécuriser les *Messages Ouverture de Canal Sécurisé*. PKCS #1 définit un ensemble d'algorithmes asymétriques qui peuvent être utilisés par les mises en œuvre UASC. Les mises en œuvre UASC n'utilisent pas l'élément *Algorithme d'Enveloppement par Clé Asymétrique* de la structure de *Politique de Sécurité* définie dans le Tableau 22.

Tableau 30 – En-tête de sécurité d'algorithme asymétrique

Dénomination	Type de données	Description
SecurityPolicyUriLength	Int32	Longueur de *l'Uri de Politique de Sécurité* en octets. Cette valeur ne doit pas dépasser 255 octets.
SecurityPolicyUri	Octet[*]	URI de la *Politique de sécurité* utilisé pour sécuriser le *Message*. Ce champ est codé sous forme de chaîne UTF8 sans terminateur nul.
SenderCertificateLength	Int32	Longueur du *Certificat de l'Émetteur* (*SenderCertificate*) en octets. Cette valeur ne doit pas dépasser les octets de la *Taille de Certificat Maximale*
SenderCertificate	Octet[*]	*Certificat* X509v3 attribué à l'*Instance d'application* d'émission. Il s'agit d'un objet binaire de grande taille à codage DER. La structure d'un *Certificat* X509 est définie dans la norme X509. Le format DER pour un *Certificat* est défini dans la norme X690 Ceci indique la *Clé Privée* effectivement utilisée pour signer le *Bloc de Messages*. La *Pile* doit fermer le canal et signaler une erreur à l'*Application* si la taille du *Certificat de l'Émetteur* est trop grande pour la capacité de mémoire tampon prise en charge par la couche de transport. Ce champ doit être nul si le *Message* n'est pas signé. Si le *Certificat* est signé par une autorité de certification, le *Certificat* CA à codage DER peut être annexé après le Certificat dans la matrice d'octets. Si le *Certificat CA* est également signé par une autre autorité de certification, ce processus est répété jusqu'à ce que la chaîne de Certificat complète soit dans la mémoire tampon ou si la limite de *Taille Maximale du Certificat de l'Émetteur* est atteinte (le processus s'arrête après le dernier *Certificat* complet qui peut être ajouté sans dépasser la limite de T*aille Maximale du Certificat de l'Émetteur*). Les récepteurs peuvent extraire les *Certificats* de la matrice d'octets au moyen de la taille de *Certificat* contenue dans l'en-tête DER (voir X509). Les récepteurs qui ne gèrent pas les chaînes de *Certificats* doivent ignorer les octets supplémentaires.
ReceiverCertificateThumbprintLength	Int32	Longueur de l'*Empreinte de signature du Certificat du Récepteur* (*ReceiverCertificateThumbprint*) en octets. La longueur de ce champ est toujours de 20 octets.
ReceiverCertificateThumbprint	Octet[*]	Empreinte de signature du *Certificat* X509v3 attribué à l'*Instance d'application* de réception. L'empreinte de signature est le condensé numérique SHA1 de la forme à codage DER du *Certificat*. Ceci indique la clé publique effectivement utilisée pour chiffrer le *Bloc de Messages*. Ce champ doit être nul si le *Message* n'est pas chiffré.

Le récepteur doit fermer le canal de communication si la longueur de l'un des champs de l'en-tête de sécurité est invalide.

Le *Certificat de l'Émetteur*, y compris les chaînes éventuelles, doit être suffisamment réduit pour s'ajuster dans un seul *Bloc de Messages* et laisser de l'espace à au moins un octet d'information textuelle. La taille maximale du *Certificat de l'Émetteur* peut être calculée à l'aide de la formule suivante:

```
MaxSenderCertificateSize =
   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
   ExtraPadding -          // ExtraPadding if present
   AsymmetricSignatureSize // If present
```

La *Taille de Bloc de Messages* dépend du protocole de transport, mais doit être au moins de 8 196 octets. La *Taille de Signature Asymétrique* dépend du nombre de bits de la clé publique

pour le *Certificat de l'Émetteur*. La *Longueur de Champ Int32* est la longueur d'une valeur Int32 codée comprenant toujours 4 octets.

L'en-tête de sécurité utilisé pour les algorithmes symétriques est défini dans le Tableau 31. Les algorithmes symétriques permettent de sécuriser tous les *Messages* autres que les *Messages Ouverture de Canal Sécurisé*. La norme FIPS 197 définit les algorithmes de chiffrement symétriques que les mises en œuvre UASC peuvent utiliser. La norme FIPS 180-2 et le code HMAC définissent quelques algorithmes de signature symétriques.

Tableau 31 – En-tête de sécurité d'algorithme symétrique

Dénomination	Type de données	Description
TokenId	UInt32	Identificateur unique du *Jeton de Canal Sécurisé/Jeton de sécurité* utilisé pour sécuriser le *Message*. Cet identificateur est renvoyé par le *Serveur* dans un *Message* de réponse *d'Ouverture de Canal Sécurisé*. Si un *Serveur* reçoit un Identificateur de jeton qu'il ne reconnaît pas, il doit renvoyer une erreur de couche de transport appropriée.

L'en-tête de sécurité est toujours suivi de l'en-tête de séquence défini dans le Tableau 32. L'en-tête de séquence garantit que le premier bloc chiffré de chaque *Message* transmis sur un canal commence avec des données différentes.

Tableau 32 – En-tête de séquence

Dénomination	Type de données	Description
SequenceNumber	UInt32	Numéro de séquence croissant monotone attribué par l'émetteur à chaque *Bloc de messages* transmis sur le *Canal Sécurisé*.
RequestId	UInt32	Identificateur attribué par le *Client* au *Message* de demande OPC UA. Tous les *Blocs de Messages* pour la demande et la réponse associée utilisent le même identificateur.

Les *Numéros de séquence* peuvent ne pas être réutilisés pour un *Identificateur de jeton*. Il convient que la durée de vie utile d'un *Jeton de sécurité* soit suffisamment courte pour s'assurer que cela ne se produit jamais. Toutefois, si cela se produit, il convient que le récepteur traite ce phénomène comme une erreur de transport et conduise à une reconnexion.

La croissance du *Numéro de Séquence* doit également être monotone pour tous les *Messages*, et le *Numéro de Séquence* ne doit pas retrouver sa valeur initiale avant d'atteindre la valeur maximum de 4 294 966 271 (UInt32.MaxValue – 1 024). La valeur initiale doit être inférieure à 1 024. Noter que cette exigence signifie que les *Numéros de Séquence* ne se réinitialisent pas lors de l'émission d'un nouvel *Identificateur de jeton*. Le *Numéro de Séquence* doit être incrémenté de un pour chaque *Bloc de Messages* transmis, à moins que le canal de communication n'ait été interrompu et rétabli. Des espaces sont admis entre le *Numéro de Séquence* pour le dernier *Bloc de Messages* reçu avant l'interruption et le *Numéro de Séquence* pour le premier *Bloc de Messages* reçu après le rétablissement de la communication. Noter que le premier *Bloc de Messages* après une interruption de réseau est toujours une demande ou une réponse d'*Ouverture de Canal Sécurisé*.

L'en-tête de séquence est suivi du corps de *Message* qui est codé au moyen du codage binaire OPC UA décrit au 5.2.6. Le corps peut être réparti entre plusieurs *Blocs de Messages*.

Chaque *Bloc de Messages* comporte également une cartouche comprenant les champs définis dans le Tableau 33.

Tableau 33 – Cartouche de message de Conversation Sécurisée OPC UA

Dénomination	Type de données	Description
PaddingSize	Octet	Le nombre d'octets de remplissage (octet de la Taille de Remplissage non compris).
Padding	Octet[*]	Remplissage ajouté à la fin du *Message* pour s'assurer que la longueur des données à chiffrer est un entier multiple de la taille de bloc de chiffrement. La valeur de chaque octet du remplissage est égale à la Taille de Remplissage.
ExtraPaddingSize	Octet	L'octet de poids fort d'un entier à deux octets utilisé pour spécifier la taille de remplissage lorsque la longueur de la clé utilisée pour chiffrer le bloc de messages est supérieure à 2048 bits. Ce champ est omis si la longueur de clé est inférieure ou égale à 2048 bits.
Signature	Octet[*]	Signature du *Bloc de Messages*. La signature inclut tous les en-têtes, toutes les données de *Message*, la Taille de Remplissage et le Remplissage.

La formule permettant de calculer le volume de remplissage dépend du nombre de données qu'il est nécessaire de transmettre (appelées *Octets à Ecrire*) (*BytesToWrite*). L'émetteur doit calculer en premier lieu le volume maximal d'espace disponible dans le *Bloc de Messages* (appelé *Taille de Corps Maximale*, en anglais *MaxBodySize*) à l'aide de la formule suivante:

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

La *Taille de l'En-tête* incluant l'*En-tête de Message* et l'*En-tête de sécurité*. La *Taille de l'En-tête de Séquence* est toujours de 8 octets.

Le chiffrement consiste à traiter un bloc dont la taille est égale à la *Taille de Bloc de Texte Clair* pour générer un bloc dont la taille est égale à la *Taille de Bloc de Texte Chiffré*. Ces valeurs dépendent de l'algorithme de chiffrement et peuvent être identiques.

Le *Message* OPC UA peut s'ajuster dans un seul bloc si *Octets à Ecrire* est inférieur ou égal à la *Taille de Corps Maximale*. Dans ce cas, la *Taille de Remplissage* est calculée à l'aide de la formule:

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

Si les *Octets à Ecrire* sont supérieurs à la *Taille de Corps Maximale*, l'émetteur doit écrire les octets correspondants avec une Taille de Remplissage égale à 0. Les octets *Octets à Ecrire – Taille de Corps Maximale* restants doivent être transmis dans des *Blocs de Messages* ultérieurs.

Les champs *Taille de Remplissage* et *Remplissage* sont absents si le *Bloc de Messages* n'est pas chiffré.

Le champ *Signature* est absent si le *Bloc de Messages* n'est pas signé.

6.7.3 Blocs de Messages et traitement d'erreurs

Les *Blocs de Messages* sont transmis sous leur forme codée. Les *Blocs de Messages* appartenant au même *Message* doivent être transmis de manière séquentielle. Lorsqu'il se produit une erreur générant un *Bloc de Messages*, l'émetteur doit alors transmettre un *Bloc de Messages* final au récepteur qui signifie l'occurrence d'une erreur et il convient qu'il élimine les blocs antérieurs. L'émetteur indique que le *Bloc de Messages* contient une erreur en réglant le fanion «Est Final» (IsFinal) sur «A» (pour Abandon). Le Tableau 34 indique le contenu du *Bloc de Messages* d'abandon.

Tableau 34 – Corps de l'abandon de message de Conversation Sécurisée OPC UA

Dénomination	Type de données	Description
Error	UInt32	Code numérique de l'erreur. Doit être l'une des valeurs énumérées dans le Tableau 41
Reason	Chaîne	Description plus prolixe de l'erreur. Cette chaîne ne doit pas comporter plus de 4 096 caractères. Un *Client* doit ignorer les chaînes de plus grande longueur.

Le récepteur doit vérifier la sécurité du *Bloc de Messages* d'abandon avant de le traiter. Si tout est normal, le récepteur doit alors ignorer le *Message*, mais ne doit pas fermer le *Canal Sécurisé*. Le *Client* doit signaler en retour l'erreur à l'*Application* sous la forme d'un *Code de Statut* pour la demande. Si le *Client* est l'émetteur, il doit alors signaler l'erreur sans attendre la réponse du *Serveur*.

6.7.4 Établissement d'un Canal Sécurisé

La plupart des *Messages* nécessitent la mise en place d'un *Canal Sécurisé*. Un *Client* effectue cette opération en transmettant au *Serveur* une demande *Ouverture de Canal Sécurisé*. Le *Serveur* doit valider le *Message* et le *Certificat Client* et renvoyer une réponse d'*Ouverture de Canal Sécurisé.* Certains paramètres, parmi les paramètres définis pour le service d'*Ouverture de Canal Sécurisé*, sont spécifiés dans l'en-tête de sécurité (voir 6.7.2) et non dans le corps du *Message*. Ainsi, le *Service d'Ouverture de Canal Sécurisé* n'est pas identique à celui spécifié dans l'IEC 62541-4. Le Tableau 35 dresse la liste des paramètres qui apparaissent dans le corps du *Message*.

Tableau 35 – Service d'Ouverture d'un Canal Sécurisé pour une Conversation Sécurisée OPC UA

Dénomination	Type de données
Demande	
RequestHeader	En-tête de Demande
ClientProtocolVersion	UInt32
RequestType	Type de Demande de Jeton de Sécurité
SecurityMode	Mode de Sécurité de Message
ClientNonce	Chaîne d'Octets
RequestedLifetime	Int32
Réponse	
ResponseHeader	En-tête de réponse
ServerProtocolVersion	UInt32
SecurityToken	Jeton de Sécurité de Canal
Secure ChannelId	UInt32
TokenId	UInt32
CreatedAt	Date&Heure
RevisedLifetime	Int32
ServerNonce	Chaîne d'Octets

Les paramètres *Version de Protocole Client* et *Version de Protocole Serveur* ne sont pas définis dans l'IEC 62541-4 et viennent s'ajouter au *Message* pour permettre une rétrocompatibilité si la Conversation Sécurisée OPC UA nécessite de faire l'objet d'une actualisation ultérieure. Les récepteurs acceptent toujours des numéros de version ultérieurs à la dernière version qu'ils prennent en charge. Le récepteur ayant le numéro de version le plus élevé est supposé assurer une rétrocompatibilité.

Si la *Conversation Sécurisée* OPC UA est utilisée avec le protocole OPC UA-TCP (voir 7.1), alors les numéros de version spécifiés dans les *Messages* d'*Ouverture de Canal Sécurisé* doivent être les mêmes que les numéros de version spécifiés dans les *Messages* d'*Accueil/Acquittement* qui utilisent le protocole OPC UA-TCP. Le récepteur doit fermer le canal et signaler l'erreur *Echec_Version de Protocole Non prise en charge* (*Bad_ProtocolVersionUnsupported)* en cas de non-correspondance.

Le *Serveur* doit renvoyer une réponse d'erreur tel que décrit dans l'IEC 62541-4 si les paramètres spécifiés par le *Client* comportent des erreurs.

La *Durée de Vie Révisée* indique au *Client* le moment où il doit renouveler le *Jeton de sécurité* en envoyant une autre demande *Ouverture de Canal Sécurisé*. Le *Client* doit toujours accepter l'ancien *Jeton de sécurité* jusqu'à ce qu'il reçoive la réponse d'*Ouverture de Canal Sécurisé*. Le *Serveur* est tenu d'accepter les demandes sécurisées avec l'ancien *Jeton de sécurité*, jusqu'à la fin de validité de ce dernier ou jusqu'à ce qu'il reçoive un *Message* du *Client* sécurisé avec le nouveau *Jeton de sécurité*. Le *Serveur* doit rejeter les demandes de renouvellement si le *Certificat de l'Émetteur* n'est pas le même que celui utilisé pour créer le *Canal Sécurisé* ou si un problème de déchiffrement ou de vérification de signature se pose. Le *Client* doit abandonner le *Canal Sécurisé* si le *Certificat* utilisé pour signer la réponse n'est pas le même que le *Certificat* utilisé pour chiffrer la demande.

Les *Messages* d'*Ouverture de Canal Sécurisé* ne sont ni signés ni chiffrés si le *Mode de Sécurité* est réglé sur *Aucun*. Les *Nonces* sont ignorés et il convient de les régler sur nul. L'*Identificateur de Canal Sécurisé* et l'*Identificateur de Jeton* sont toujours attribués, mais les *Messages* sont échangés via le canal sans aucune sécurité. Le *Jeton de sécurité* doit toujours être renouvelé avant l'expiration de la *Durée de Vie Révisée*. Les récepteurs doivent toujours ignorer les *Identificateurs de Jeton* non valides ou expirés.

En cas d'interruption du canal de communication, le *Serveur* doit maintenir un *Canal Sécurisé* suffisamment long pour permettre au *Client* de se reconnecter. Le paramètre *Durée de Vie Révisée* indique également au *Client* le temps d'attente du *Serveur*. Le *client* doit supposer que le *Canal Sécurisé* a été fermé s'il ne parvient pas à se reconnecter dans la période spécifiée.

Le *Jeton d'Authentification* dans l'*En-tête de Demande* doit être réglé sur nul.

Si une erreur survient après que le *Serveur* a vérifié la sécurité du *Message*, un message *Erreur de Service* doit être renvoyé et non pas une réponse d'*Ouverture de Canal Sécurisé*. Le *Message* d'*Erreur de Service* est décrit dans l'IEC 62541-4.

Si le *Mode de Sécurité* n'est pas réglé sur *Aucun*, le *Serveur* doit alors vérifier que l'*En-tête de Sécurité* mentionnait un *Certificat de l'Émetteur* et une *Empreinte de Signature du Certificat du Récepteur*.

6.7.5 Dérivation des clés

Une fois établi le *Canal Sécurisé,* les *Messages* sont signés et chiffrés à l'aide des clés dérivées des *Nonces* échangés lors de l'appel d'*Ouverture de Canal Sécurisé.* Ces clés sont dérivées par transmission des *Nonces* à une fonction pseudoaléatoire qui produit une séquence d'octets à partir d'un ensemble d'entrées. Une fonction pseudoaléatoire est représentée par la déclaration de fonction suivante:

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

Où *longueur* (*length*) est le nombre d'octets à renvoyer et *décalage (offset)* est un nombre d'octets depuis le début de la séquence.

Les longueurs des clés qu'il est nécessaire de générer dépendent de la *Politique de Sécurité* utilisée pour le canal. Les informations suivantes sont spécifiées par la *Politique de Sécurité:*

a) *Longueur de la Clé de Signature* (à partir de la *Longueur de Clé de Signature Dérivée*);

b) *Longueur de la Clé de Chiffrement* (induite par l'*Algorithme de Chiffrement Symétrique*);

c) *Taille du Bloc de Chiffrement* (induite par l'*Algorithme de Chiffrement Symétrique*).

Les paramètres transmis à la fonction pseudoaléatoire sont spécifiés dans le Tableau 36.

Tableau 36 – Paramètres de génération de clés de cryptographie

Légende	Secret	Valeur de départ (seed)	Longueur	Décalage
Clé de Signature du Client (ClientSigningKey)	Nonce du Serveur	Nonce du Client	Longueur de la Clé de Signature	0
Clé de Chiffrement du Client (ClientEncryptingKey)	Nonce du Serveur	Nonce du Client	Longueur de la Clé de Chiffrement	Longueur de la Clé de Signature
Vecteur d'Initialisation du Client (ClientInitializationVector)	Nonce du Serveur	Nonce du Client	Taille du Bloc de Chiffrement	Longueur de la Clé de Signature+Longueur de la Clé de Chiffrement
Clé de Signature du Serveur (ServerSigningKey)	Nonce du Client	Nonce du Serveur	Longueur de la Clé de Signature	0
Clé de Chiffrement du Serveur (ServerEncryptingKey)	Nonce du Client	Nonce du Serveur	Longueur de la Clé de Chiffrement	Longueur de la Clé de Signature
Vecteur d'Initialisation du Serveur (ServerInitializationVector)	Nonce du Client	Nonce du Serveur	Taille du Bloc de Chiffrement	Longueur de la Clé de Signature+Longueur de la Clé de Chiffrement

Les clés du *Client* permettent de sécuriser les *Messages* transmis par ce dernier. Les clés du *Serveur* permettent de sécuriser les *Messages* transmis par ce dernier.

La spécification SSL/TLS définit une fonction pseudoaléatoire appelée P_SHA1 utilisée pour certains *Profils de Sécurité*. L'algorithme P_SHA1 est défini comme suit:

```
P_SHA1(secret, seed) = HMAC_SHA1(secret, A(1) + seed) +
                       HMAC_SHA1(secret, A(2) + seed) +
                       HMAC_SHA1(secret, A(3) + seed) + …
où A(n) est défini sous la forme:
 A(0) = seed
 A(n) = HMAC_SHA1(secret, A(n-1))
+ indique la concaténation des résultats aux résultats précédents.
```

6.7.6 Vérification de la sécurité d'un message

Le contenu du *Bloc de Messages* ne doit pas être interprété tant que le *Message* n'a pas été déchiffré et que la signature et le numéro de séquence n'ont pas été vérifiés.

Le récepteur doit fermer le canal de communication si une erreur survient lors de la vérification du *message*. Si le récepteur est le *Serveur*, il doit également envoyer un *message* d'erreur de transport avant de fermer le canal. Une fois le canal fermé, le *Client* doit essayer de rouvrir le canal et demander un nouveau jeton par la transmission d'une demande *Ouverture de Canal Sécurisé*. Le mécanisme de transmission des erreurs de transport au *Client* dépend du canal de communication.

Le récepteur doit vérifier tout d'abord l'*Identificateur de Canal Sécurisé*. Cette valeur peut être 0 si le *message* est une demande *Ouverture de Canal Sécurisé*. Pour les autres *messages*, il doit signaler l'erreur *Echec_Canal Sécurisé Inconnu* si l'*Identificateur de Canal Sécurisé* n'est pas reconnu. Si le *message* est une demande *Ouverture de Canal Sécurisé* et si l'*Identificateur de Canal Sécurisé* n'est pas 0, alors le *Certificat de l'Émetteur* doit être identique au *Certificat de l'Émetteur* utilisé pour créer le canal.

Si le message est sécurisé par des algorithmes asymétriques, alors le récepteur doit vérifier qu'il prend en charge la *Politique de Sécurité* spécifiée dans la demande. Si le *Message* est la

réponse transmise au *Client*, alors la *Politique de Sécurité* doit être identique à celle spécifiée dans la demande. La *Politique de Sécurité* du *Serveur* doit être identique à celle appliquée à l'origine pour la création du *Canal Sécurisé.* Le récepteur doit vérifier que le Certificat est fiable d'abord et retourner *Echec_Certificat Non Fiable (Bad_CertificateUntrusted)* sur erreur. Le récepteur doit alors vérifier le *Certificat de l'Émetteur* en appliquant les règles définies dans l'IEC 62541-4. Il doit par ailleurs signaler l'erreur appropriée en cas d'échec de la validation du *Certificat*. Le récepteur doit également vérifier l'*Empreinte de Signature du Certificat du Récepteur* et signaler une erreur *Echec_Certificat Inconnu* (*Bad_CertificateUnknown)* s'il ne la reconnaît pas.

Si le *Message* est sécurisé par des algorithmes symétriques, alors une erreur *Echec_Jeton de Canal Sécurisé Inconnu* doit être signalée lorsque l'*Identificateur du Jeton* fait référence à un jeton dont la validité a expiré ou qui n'est pas reconnu.

En cas d'échec de la validation du déchiffrement ou de la signature, une erreur *Echec_Contrôles de Sécurité Non aboutis* est alors signalée. Si une mise en œuvre permet l'utilisation de plusieurs *Modes de Sécurité*, le récepteur doit également vérifier que le *message* a été correctement sécurisé tel que le requiert le *Mode de Sécurité* spécifié dans la demande *Ouverture de Canal Sécurisé*.

Une fois la validation de sécurité achevée, le récepteur doit vérifier l'*Identificateur de Demande* et le *Numéro de Séquence*. Une erreur *Echec_Contrôles de Sécurité Non aboutis* est signalée en cas d'échec de ces contrôles. L'*Identificateur de Demande* n'est à vérifier que par le *Client*, dans la mesure où seul ce dernier sait s'il est ou non valide.

À ce stade, le *Canal Sécurisé* sait qu'il traite un *message* authentifié qui n'a pas été falsifié, ni transmis une nouvelle fois. Cela signifie que le *Canal Sécurisé* peut renvoyer des réponses d'erreur sécurisées en cas d'autres problèmes.

Les *Piles* qui mettent en œuvre le protocole UASC doivent comporter un mécanisme de consignation d'erreurs lorsque les *messages* non valides sont rejetés. Ce mécanisme est destiné aux développeurs, intégrateurs et administrateurs de systèmes afin qu'ils déboguent les problèmes de configuration de systèmes et de réseaux et détectent les attaques sur le réseau.

7 Protocoles de Transport

7.1 Protocole OPC UA TCP

7.1.1 Vue d'ensemble

Le protocole OPC UA TCP est un protocole TCP simple qui établit un canal duplex intégral entre un *Client* et le *Serveur*. Ce protocole présente deux caractéristiques principales qui le différencient du protocole HTTP. Premièrement, ce protocole permet le renvoi des réponses dans n'importe quel ordre. Deuxièmement, il permet le renvoi de ces mêmes réponses sur un point d'extrémité de transport TCP différent si les échecs de communication provoquent une interruption provisoire de la session TCP.

Le protocole OPC UA TCP est conçu pour fonctionner avec mise en œuvre du *Canal Sécurisé* par une couche supérieure de la pile. Ainsi, le protocole OPC UA TCP définit ses interactions avec le *Canal Sécurisé*, outre le protocole par fil.

7.1.2 Structure de message

Chaque *Message* OPC UA TCP comporte un en-tête comprenant les champs définis dans le Tableau 37.

Tableau 37 – En-tête de message OPC UA TCP

Dénomination	Type	Description
MessageType	Octet[3]	Code ASCII à trois octets qui identifie le type de *Message.* Les valeurs suivantes sont définies à ce moment: HEL *Message d'accueil.* ACK *Message d'acquittement*. ERR *Message d'erreur*. La couche de *Canal Sécurisé* définit les valeurs supplémentaires que la couche OPC UA TCP doit accepter.
Reserved	Octet[1]	Ignoré. doit être réglé sur les codes ASCII pour «F» si le *Type de Message (MessageType*) est l'une des valeurs prises en charge par le protocole OPC UA TCP.
MessageSize	UInt32	Longueur du *Message,* en octets. Cette valeur inclut les 8 octets de l'en-tête de *Message.*

La présentation de l'en-tête de *Message* OPC UA TCP est volontairement identique aux premiers 8 octets de l'en-tête de *Message* de Conversation Sécurisée OPC UA définie dans le Tableau 29. Ceci permet à la couche OPC UA TCP d'extraire les *Messages* de *Canal Sécurisé* de la série de bits entrant même si elle ne comprend pas leur contenu.

La couche OPC UA TCP doit vérifier le *Type de Message* et s'assurer que la *Taille du Message* est inférieure à la *Capacité de la mémoire tampon de Réception* négociée, avant de transmettre tout *Message* à la couche de *Canal Sécurisé*.

Le *Message* d'Accueil comporte les champs supplémentaires présentés dans le Tableau 38.

Tableau 38 – Message d'Accueil OPC UA TCP

Dénomination	Type de données	Description
ProtocolVersion	UInt32	Dernière version du protocole OPC UA TCP pris en charge par le *Client*. Le *Serveur* peut refuser le *Client* en renvoyant le message *Echec_Version de Protocole Non prise en charge*. Si le *Serveur* accepte la connexion, il est chargé de s'assurer qu'il renvoie des *Messages* conformes à cette version du protocole. Le *Serveur* doit toujours accepter des versions supérieures à celles qu'il prend en charge.
ReceiveBufferSize	UInt32	*Bloc de Messages* le plus grand que l'émetteur peut recevoir. Cette valeur doit être supérieure à 8 192 octets.
SendBufferSize	UInt32	*Bloc de Messages* l le plus grand que l'émetteur transmet. Cette valeur doit être supérieure à 8 192 octets.
MaxMessageSize	UInt32	Taille maximale de tout *Message* de réponse. Le *Serveur* doit abandonner le message par un *Echec_Code de Statut trop long de Réponse* si un *Message* de réponse dépasse cette valeur. Le mécanisme d'abandon des *Messages* est décrit intégralement en 6.7.3. Le corps de *Message* non chiffré permet de calculer la taille du *Message*. Une valeur de zéro indique que le *Client* n'a pas de limite.
MaxChunkCount	UInt32	Nombre maximal de blocs dans tout *message* de réponse. Le *serveur* doit abandonner le *message* par un *Echec_Code de Statut trop long de Réponse* si un *Message* de réponse dépasse cette valeur. Le mécanisme d'abandon des *Messages* est décrit intégralement en 6.7.3. Une valeur de zéro indique que le *Client* n'a pas de limite.
EndpointUrl	Chaîne	URL du point d'extrémité auquel le *Client* souhaite se connecter. La valeur codée doit être inférieure à 4 096 octets. Les *Serveurs* doivent renvoyer un message d'erreur Echec_Url Tcp Refusée et fermer la connexion si la longueur dépasse 4 096 ou si elle ne reconnaît pas la ressource identifiée par l'URL.

Le paramètre *Url de point d'extrémité* permet à plusieurs *Serveurs* de partager le même port d'une machine. Le processus d'écoute (également appelé passerelle de procuration, Proxy) du port se connecterait au *Serveur* identifié par l'*Url de point d'extrémité* et transmettrait tous les *Messages* au *Serveur* par l'intermédiaire de ce connecteur logiciel. Lorsqu'un connecteur se ferme, la passerelle par procuration (proxy) doit alors fermer l'autre connecteur.

Le *Message* d'*Acquittement* comporte les champs supplémentaires présentés dans le Tableau 39.

Tableau 39 – Message d'Acquittement de protocole OPC UA TCP

Dénomination	Type	Description
ProtocolVersion	UInt32	Dernière version du protocole OPC UA TCP pris en charge par le *Serveur*. Si le *Client* accepte la connexion, il est chargé de s'assurer qu'il transmet des *Messages* conformes à cette version du protocole. Le *Client* doit toujours accepter des versions supérieures à celles qu'il prend en charge.
ReceiveBufferSize	UInt32	*Bloc de messages* le plus grand que l'émetteur peut recevoir. Cette valeur ne doit pas être supérieure à celle demandée par le *Client* dans le *Message* d'accueil. Cette valeur doit être supérieure à 8 192 octets.
SendBufferSize	UInt32	*Bloc de messages* le plus grand que l'émetteur transmet. Cette valeur ne doit pas être supérieure à celle demandée par le *Client* dans le *Message* d'accueil. Cette valeur doit être supérieure à 8 192 octets.
MaxMessageSize	UInt32	Taille maximale de tout *Message* de demande. Le *Client* doit abandonner le *message* avec un *Echec_Code de Statut trop long de Demande* si un *message* de demande dépasse cette valeur. Le mécanisme d'abandon des *Messages* est décrit intégralement en 6.7.3. Le corps de *Message* non chiffré permet de calculer la taille du *message*. Une valeur de zéro indique que le *Serveur* n'a pas de limite.
MaxChunkCount	UInt32	Nombre maximal de blocs dans tout *Message* de demande. Le *Client* doit abandonner le *message* avec un *Echec_Code de Statut trop long de Demande* si un *Message* de demande dépasse cette valeur. Le mécanisme d'abandon des *Messages* est décrit intégralement en 6.7.3. Une valeur de zéro indique que le *Serveur* n'a pas de limite.

Le *Message* d'erreur comporte les champs supplémentaires présentés dans le Tableau 40.

Tableau 40 – Message d'erreur OPC UA TCP

Dénomination	Type	Description
Error	UInt32	Code numérique de l'erreur. Doit être l'une des valeurs énumérées dans le Tableau 41.
Reason	Chaîne	Description plus prolixe de l'erreur. Cette chaîne ne doit pas comporter plus de 4 096 caractères. Un *Client* doit ignorer les chaînes de plus grande longueur.

La Figure 14 illustre la structure d'un *Message* à la volée. Ceci inclut également les illustrations du processus d'association des éléments de *Message* définis par la correspondance de Codage OPC UA Binaire (voir 5.2) et la correspondance de Conversation Sécurisée OPC UA (voir 6.7) avec les *Messages* OPC UA TCP.

Le connecteur logiciel est toujours fermé par une commande du *Serveur* après la transmission d'un *Message d'erreur*.

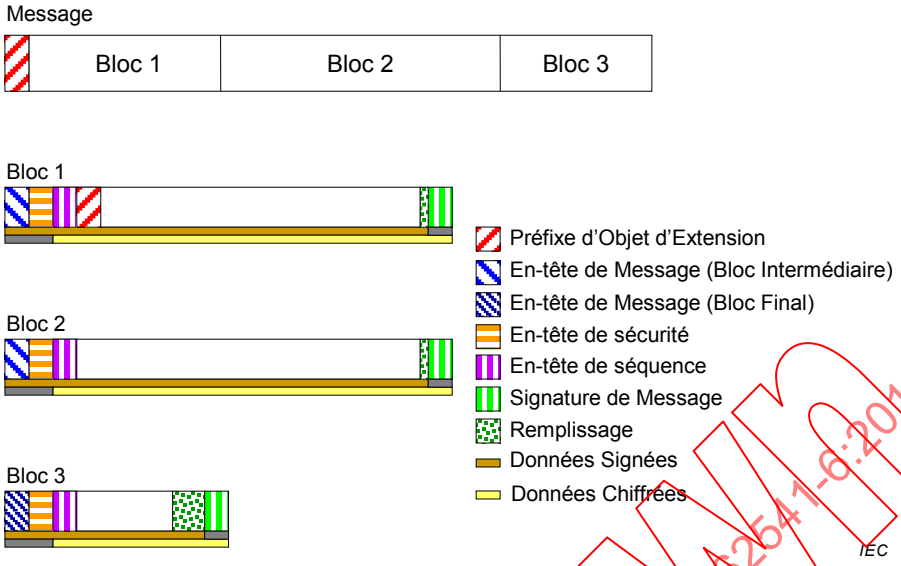

Figure 14 – Structure de message OPC UA TCP

7.1.3 Établissement d'une connexion

Les connexions sont toujours lancées par le *Client* qui crée le connecteur logiciel avant de transmettre la première demande *Ouverture de Canal Sécurisé*. Une fois le connecteur logiciel créé, le premier *Message* transmis doit être un message d'*Accueil* qui spécifie les capacités de la mémoire tampon prises en charge par le *Client*. Le *Serveur* doit répondre par un *Message d'Acquittement* qui complète la négociation sur la capacité de la mémoire tampon. La capacité de la mémoire tampon négociée doit être signalée à la couche du *Canal Sécurisé*. La *Capacité de la mémoire tampon de Transmission* négociée spécifie la taille des *Blocs de Messages* à utiliser pour les *Messages* transmis par la connexion.

Les *Messages* d'*Accueil*/*Acquittement* ne peuvent être transmis qu'une seule fois. S'ils font l'objet d'une nouvelle réception, le récepteur doit signaler une erreur et fermer le connecteur logiciel. Les *Serveurs* doivent fermer le connecteur logiciel après un temps déterminé s'ils ne reçoivent pas de *Message* d'*Accueil*. Cette durée doit être configurable et avoir une valeur par défaut qui ne dépasse pas deux minutes.

Le *Client* transmet la demande *Ouverture de Canal Sécurisé* à réception du message d'*Acquittement* en provenance du *Serveur*. Si le *Serveur* accepte le nouveau canal, il doit associer le connecteur logiciel à l'*Identificateur de Canal Sécurisé*. Le *Serveur* utilise cette association pour déterminer le connecteur logiciel à utiliser lorsqu'il est tenu de transmettre une réponse au *Client*. Le *Client* procède de même lorsqu'il reçoit la réponse *Ouverture de Canal Sécurisé*.

La séquence de *Messages* au moment de l'établissement d'une connexion OPC UA TCP est présentée à la Figure 15.

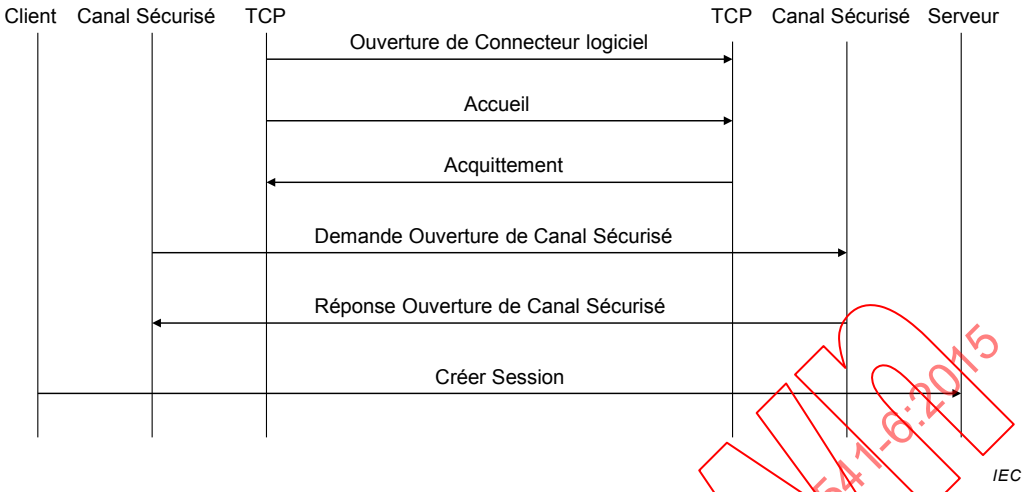

Figure 15 – Établissement d'une connexion OPC UA TCP

L'*Application Serveur* n'effectue aucune opération pendant la période de négociation du *Canal Sécurisé*. Toutefois, elle doit fournir à la *Pile* la liste des *Certificats* de confiance. La *Pile* doit notifier à l'*Application Serveur* toute réception d'une demande *Ouverture de Canal Sécurisé*. Ces notifications doivent inclure la réponse *Ouverture de Canal Sécurisé* ou *Erreur* renvoyée au client.

7.1.4 Fermeture d'une connexion

Le *Client* ferme la connexion par la transmission d'une demande *Fermeture de Canal Sécurisé* et par la fermeture du connecteur logiciel par commande. Lorsque le *Serveur* reçoit ce *Message*, il doit libérer toutes les ressources affectées au canal. Le *Serveur* ne transmet aucune réponse *Fermeture de Canal Sécurisé*.

En cas d'échec de vérification de la sécurité pour le *Message Fermeture de Canal Sécurisé*, le *Serveur* doit alors signaler l'erreur et fermer le connecteur logiciel. Le *Serveur* doit permettre au *Client* de tenter de se reconnecter.

La séquence de *Messages* au moment de la fermeture d'une connexion OPC UA TCP est présentée à la Figure 16.

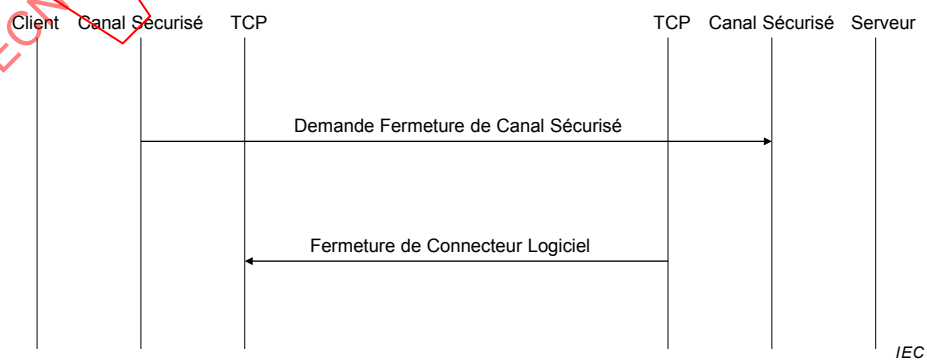

Figure 16 – Fermeture d'une connexion OPC UA TCP

L'*Application Serveur* n'effectue aucune opération lorsque le *Canal Sécurisé* est fermé. Cependant, la *Pile* doit notifier à l'*Application Serveur* toute réception d'une demande *Fermeture de Canal Sécurisé* ou tout nettoyage par la *Pile* d'un *Canal Sécurisé* abandonné.