IEC 62541-6

Edition 1.0 2011-10

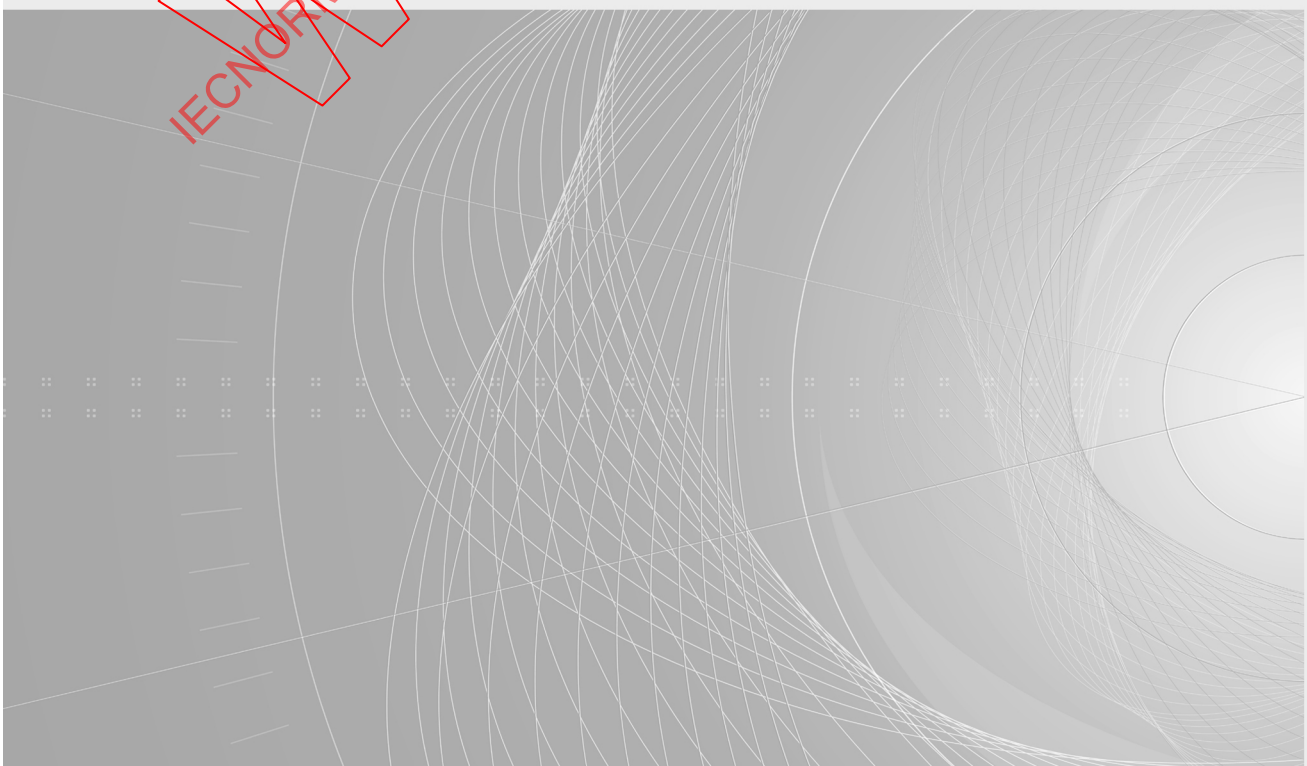
INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 6: Mappings

Architecture unifiée OPC – Partie 6: Correspondances



IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub
The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.
- IEC Just Published: www.iec.ch/online_news/justpub
Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.
- Electropedia: www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.
- Customer Service Centre: www.iec.ch/webstore/custserv
If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:
Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) est la première organisation mondiale qui élabore et publie des normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications CEI

Le contenu technique des publications de la CEI est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

- Catalogue des publications de la CEI: www.iec.ch/searchpub/cur_fut-f.htm
Le Catalogue en-ligne de la CEI vous permet d'effectuer des recherches en utilisant différents critères (numéro de référence, texte, comité d'études,…). Il donne aussi des informations sur les projets et les publications retirées ou remplacées.
- Just Published CEI: www.iec.ch/online_news/justpub
Restez informé sur les nouvelles publications de la CEI. Just Published détaille deux fois par mois les nouvelles publications parues. Disponible en-ligne et aussi par email.
- Electropedia: www.electropedia.org
Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 20 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International en ligne.
- Service Clients: www.iec.ch/webstore/custserv/custserv_entry-f.htm
Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions, visitez le FAQ du Service clients ou contactez-nous:
Email: csc@iec.ch
Tél.: +41 22 919 02 11
Fax: +41 22 919 03 00

IEC 62541-6

Edition 1.0 2011-10

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 6: Mappings

Architecture unifiée OPC – Partie 6: Correspondances

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

PRICE CODE
CODE PRIX XB

ICS 25.040.40; 25.100.01

ISBN 978-2-88912-728-3

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.

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

FDIS	Report on voting
65E/193/FDIS	65E/215/RVD

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.

INTRODUCTION

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

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 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 referenced documents are indispensable for the application of this document. 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 Unfied architecture: Part 1 – Overview and Concepts*

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

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

IEC 62541-4[1]:----, *OPC Unified architecture: Part 4 – Services*

IEC 62541-5[2], *OPC Unified architecture: Part 5 – Information Model*

IEC 62541-7[3], *OPC Unified architecture: Part 7 – Profiles*

ITU-T X.690: *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

available at <http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>

ITU-T X.200: *Information technology – Open Systems Interconnection – Basic Reference Model*

available at <http://www.itu.int/rec/T-REC-X.200-199407-I/en>

ITU-T X.509: *Information technology – Open Systems Interconnection – The directory: Public Key and Attribute Certificate Frameworks*

available at <http://www.itu.int/rec/T-REC-X.509/en>

XML Schema Part 1: *XML Schema Part 1: Structures (Second Edition)*

available at <http://www.w3.org/TR/xmlschema-1/>

XML Schema Part 2: *XML Schema Part 2: Datatypes (Second Edition)*

available at <http://www.w3.org/TR/xmlschema-2/>

---

[1] To be published.

[2] To be published.

[3] To be published.

SOAP Part 1: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*
available at <http://www.w3.org/TR/soap12-part1/>

SOAP Part 2: *SOAP Version 1.2 Part 2: Adjuncts (Second Edition)*
available at <http://www.w3.org/TR/soap12-part2/>

XML Encryption: *XML Encryption Syntax and Processing*
available at <http://www.w3.org/TR/xmlenc-core/>

XML Signature: *XML-Signature Syntax and Processing (Second Edition)*
available at <http://www.w3.org/TR/xmldsig-core/>

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

WS Addressing: *Web Services Addressing (WS-Addressing)*
available at <http://www.w3.org/Submission/ws-addressing/>

WS Trust: *WS Trust 1.3*
available at <http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html>

WS Secure Conversation: *WS Secure Conversation 1.3*
available at <http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html>

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

SSL/TLS: RFC 2246 - *The TLS Protocol Version 1.0*
available at <http://www.ietf.org/rfc/rfc2246.txt>

WS-I *Basic Profile Version 1.1*
available at <http://www.ws-i.org/Profiles/BasicProfile-1.1.html>

WS-I *Basic Security Profile Version 1.1*
available at <http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html>

HTTP: RFC 2616 - *Hypertext Transfer Protocol - HTTP/1.1*
available at <http://www.ietf.org/rfc/rfc2616.txt>

HTTPS: RFC 2818 - *HTTP Over TLS*
available at <http://www.ietf.org/rfc/rfc2818.txt>

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

IEEE-754: *Standard for Binary Floating-Point Arithmetic*
available at <http://grouper.ieee.org/groups/754/>

HMAC: RFC 2104 - *HMAC - Keyed-Hashing for Message Authentication*
available at <http://www.ietf.org/rfc/rfc2104.txt>

PKCS #1 : RFC 2437 - *PKCS #1 - RSA Cryptography Specifications Version 2.0*
available at <http://www.ietf.org/rfc/rfc2437.txt>

PKCS #12 : *PKCS 12 v1.0: Personal Information Exchange Syntax*
available at <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf>

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

FIPS 197: *Advanced Encyption Standard (AES)*
available at <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

UTF8: RFC 3629 - *UTF-8, a transformation format of ISO 10646*
available at <http://tools.ietf.org/html/rfc3629>

RFC 3280: *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*
available at <http://www.ietf.org/rfc/rfc3280.txt>

RFC 4514: *LDAP: String Representation of Distinguished Names*
available at <http://www.ietf.org/rfc/rfc4514.txt>

3 Terms, definitions and abbreviations

3.1 Terms and definitions

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

3.1.1
Data Encoding
Data Encoding is 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 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
collection of software libraries that implement one or more *Stack Profiles*; *Stacks* have an API which hides the implementation details from the application developer

3.1.5
Stack Profile
combination of *DataEncodings, SecurityProtocol* and *TransportProtocol Mappings*

NOTE OPC UA applications implement one or more *StackProfiles* and can only communicate with OPC UA applications that support a *StackProfile* that they support.

3.1.6
Transport Protocol
represents a way to exchange serialized OPC UA messages between OPC UA applications

3.2 Abbreviations

API	Application Programming Interface
ASN.1	Abstract Syntax Notation #1 (used in ITU-T X.690)
BP	WS-I Basic Profile Version
BSP	WS-I Basic Security Profile
CSV	Comma Separated Value (File Format)
HTTP	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	UA Secure Conversation
WS-*	The 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 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 Clause 7. 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 annexes.

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 unsigned integers which means any Java API will need to map unsigned integers onto a signed integer type.

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

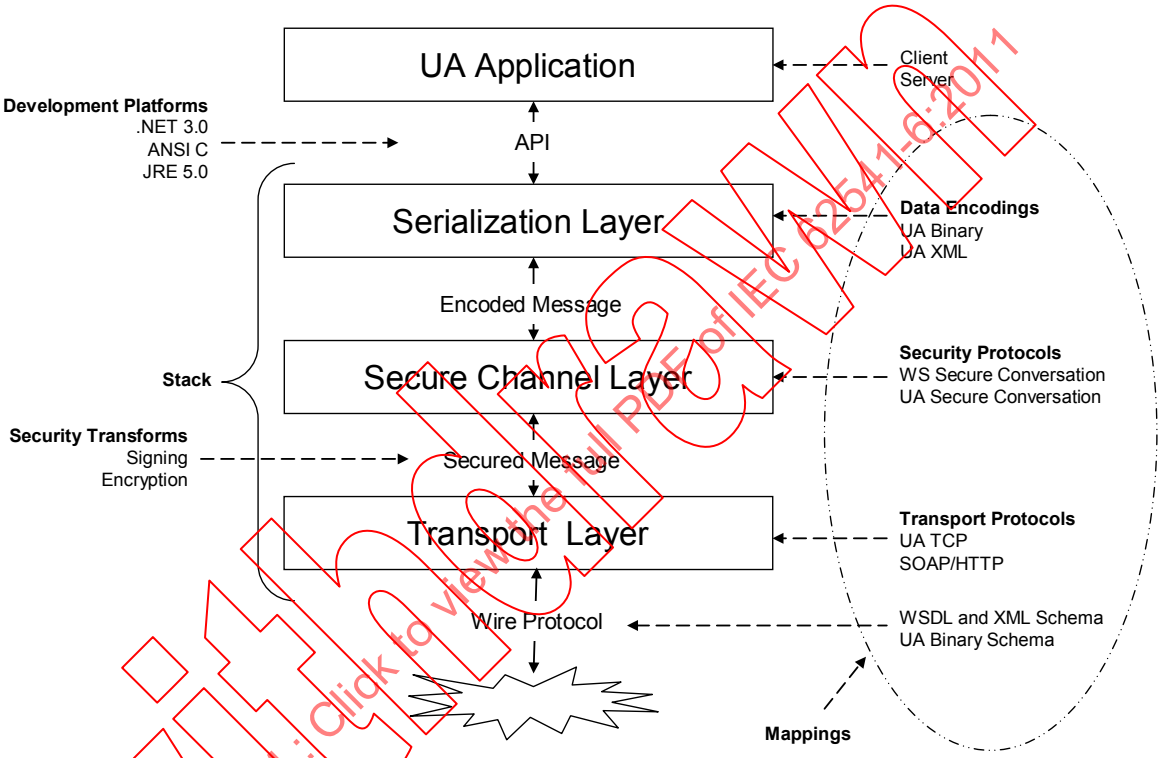


Figure 1 – The OPC UA Stack Overview

The layers described in this specification do not correspond to layers in the OSI 7 layer model [ITU-T X.200]. 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 an 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 ExtensionObject

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

Complex data types are represented in a Server address space as sub-types of the Structure data type. The encodings available for any given complex data type are represented as a DataTypeEncoding Object in the Server address space. The NodeId for the DataTypeEncoding Object is the identifier stored in the ExtensionObject. Subclause 5.8 of IEC 62541-3 describes how DataTypeEncoding nodes are related to other nodes of the address space.

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

5.1.5 Variant

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

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

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

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

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

5.2 OPC UA Binary

5.2.1 General

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

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

values. The encodings for each primitive type specify how to encode either a *Null* or a default value for the type.

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

5.2.2 Built-in Types

5.2.2.1 Boolean

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

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

5.2.2.2 Integer

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

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

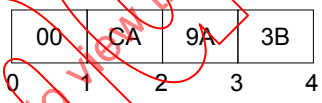


Figure 2 – Encoding Integers in a Binary Stream

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).

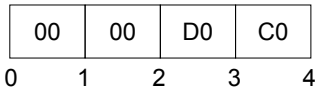


Figure 3 illustrates how the value −6,5 (Hex: C0D00000) should be encoded as a *Float*.

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 a IEEE quiet-NAN.

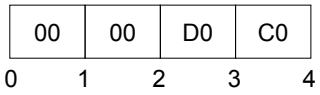


Figure 3 – Encoding Floating Points in a Binary Stream

5.2.2.4 String

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

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

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

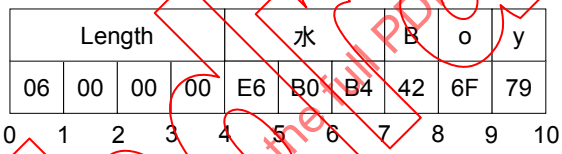


Figure 4 – Encoding Strings in a Binary Stream

5.2.2.5 DateTime

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

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

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

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

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

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.

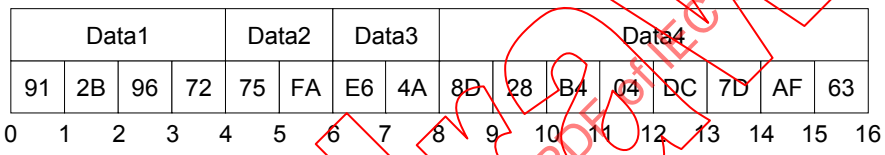


Figure 5 – Encoding Guids in a Binary Stream

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 UTF-8 string and then encoded as *ByteString*.

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

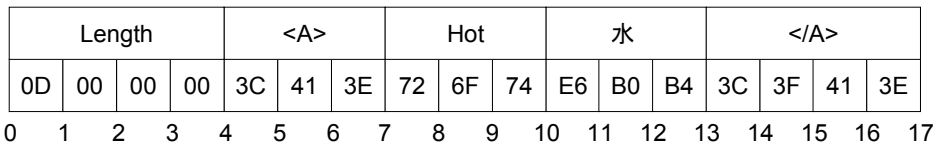


Figure 6 – Encoding XmlElements in a Binary Stream

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

Table 5 – NodeId Encoding Values

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

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

Table 6 – Standard NodeId Binary Encoding

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

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

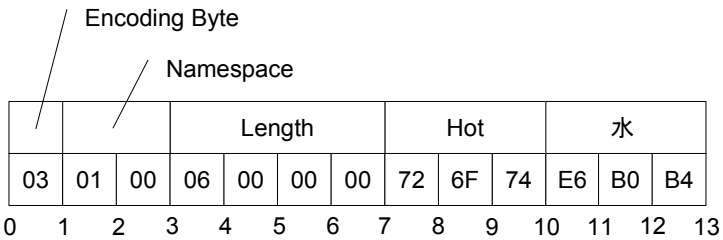


Figure 7 – A String NodeId

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

Table 7 – Two Byte NodeId Binary Encoding

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

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

Figure 8 – A Two Byte NodeId

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

Table 8 – Four Byte NodeId Binary Encoding

Name	Data Type	Description
Namespace	Byte	The Namespace shall be in the range 0 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.

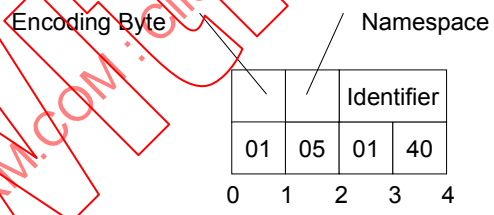


Figure 9 – A Four Byte NodeId

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 Encoding

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

5.2.2.11 StatusCode

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

5.2.2.12 DiagnosticInfo

A *DiagnosticInfo* structure is described in IEC 62541-4, 7.8. It specifies a number of fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

As described in 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 Encoding

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

5.2.2.13 QualifiedName

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

The abstract *QualifiedName* structure is defined in IEC 62541-3, 8.3.

Table 11 – QualifiedName Binary Encoding

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

5.2.2.14 LocalizedText

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

The abstract *LocalizedText* structure is defined in IEC 62541-3, 8.5.

Table 12 – LocalizedText Binary Encoding

Name	Data Type	Description
EncodingMask	Byte	A bit mask that indicates which fields are present in the streams. 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.

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

Table 13 – Extension Object Binary Encoding

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

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

5.2.2.16 Variant

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

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

Table 14 – Variant Binary Encoding

Name	Data Type	Description
EncodingMask	Byte	The type of data encoded in the stream. The mask has the following bits assigned: 0:5 Built-in Type Id (see Table 1). 6 True if the Array Dimensions field is encoded. 7 True if an array of values is encoded.
ArrayLength	Int32	The number of elements in the array. This field is only present if the array bit is set in the encoding mask. Multi-dimensional arrays are encoded as a one dimensional array and this field specifies the total number of elements. The original array can be reconstructed from the dimensions that are encoded after the value field. Higher rank dimensions are serialized first. 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 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 possible 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 Encoding

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

All fields specified in the complex type shall be encoded.

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

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

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

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

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

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

Table 16 – Sample OPC UA Binary Encoded Structure

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

5.2.6 Messages

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

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

5.3 XML

5.3.1 Built-in Types

5.3.1.1 General

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

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

5.3.1.2 Boolean

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

5.3.1.3 Integer

Integer values are encoded using one of the 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

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

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

5.3.1.6 DateTime

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

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

Correct:

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

Incorrect:

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

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

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

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

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

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

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

5.3.1.7 Guid

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

The XML schema for a *Guid* is:

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

5.3.1.8 ByteString

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

The XML schema for a *ByteString* is:

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

5.3.1.9 XmlElement

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

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

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

5.3.1.10 NodeId

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

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

The elements of the syntax are described in Table 19.

Table 19 – Components of NodeId

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

Examples of *NodeIds:*

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

The XML schema for a *NodeId* is:

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

5.3.1.11 ExpandedNodeId

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

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

Table 20 – Components of ExpandedNodeId

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

The XML schema for a *ExpandedNodeId* is:

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

5.3.1.12 StatusCode

A *StatusCode* is 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 a *xs:complexType* with the following XML schema:

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

5.3.1.14 QualifiedName

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

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

5.3.1.15 LocalizedText

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

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

5.3.1.16 ExtensionObject

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

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

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

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

5.3.1.17 Variant

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

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

If the *Variant* represents a scalar value then it shall contain a single child element with the name of the built-in type. For example, the single precision floating point value 3,141 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

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

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

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

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

5.3.4 Structures

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

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

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

5.3.5 Messages

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

6 Security Protocols

6.1 Security Handshake

All *SecurityProtocols* shall implement the *OpenSecureChannel* and *CloseSecureChannel* services defined in 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*.

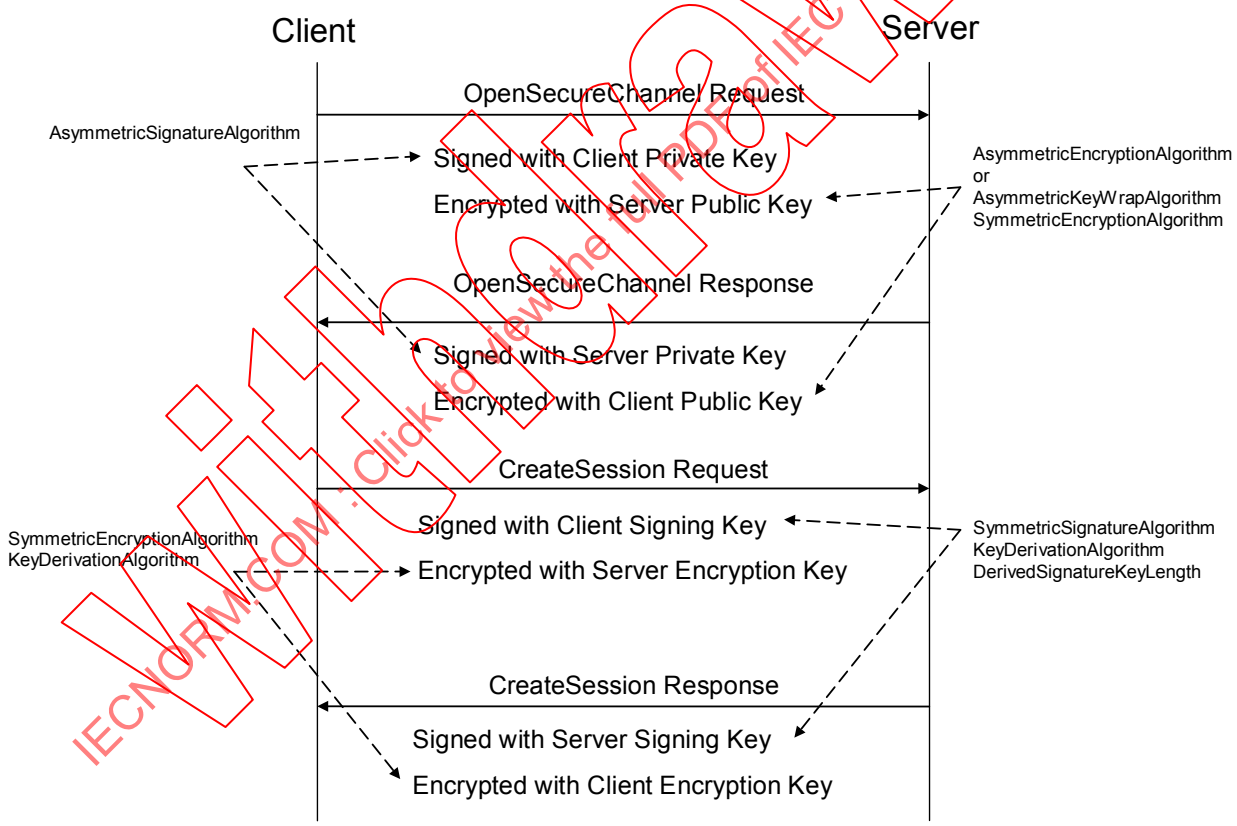


Figure 10 – Security Handshake

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.
KeyDerivationAlgorithm	The key derivation algorithm to use.
DerivedSignatureKeyLength	The length in bits of the derived key used for message authentication.

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

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

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

6.2 Certificates

6.2.1 General

OPC UA *Applications* use *Certificates* to store the public keys needed for asymmetric cryptography operations. All *SecurityProtocols* use X509 Version 3 *Certificates* (see ITU-T X.509) encoded using the DER format (see ITU-T X.690). *Certificates* used by OPC UA *Applications* shall also conform to RFC 3280 which defines a profile for X509 *Certificates* when they are used as part of an Internet based application.

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

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

6.2.2 Application Instance Certificate

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

Table 23 – ApplicationInstanceCertificate

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

6.2.3 Signed Software Certificate

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

Table 24 – SignedSoftwareCertificate

Name		Description
SignedSoftwareCertificate		An X509v3 Certificate.
version	version	Shall be “V3”
serialNumber	serialNumber	The serial number assigned by the issuer.
signatureAlgorithm	signatureAlgorithm	The algorithm used to sign the Certificate.
signature	signature	The signature created by the Issuer.
issuer	issuer	The distinguished name of the Certificate used to create the signature. The *issuer* field is completely described in RFC 3280.
validity	validTo, validFrom	When the *Certificate* becomes valid and when it expires.
subject	subject	The distinguished name of the product. The Common Name attribute shall be the same as the *productName* in the *SoftwareCertificate* and the Organization Name attribute shall 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. 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 WS Secure Conversation

6.3.1 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 Version 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.

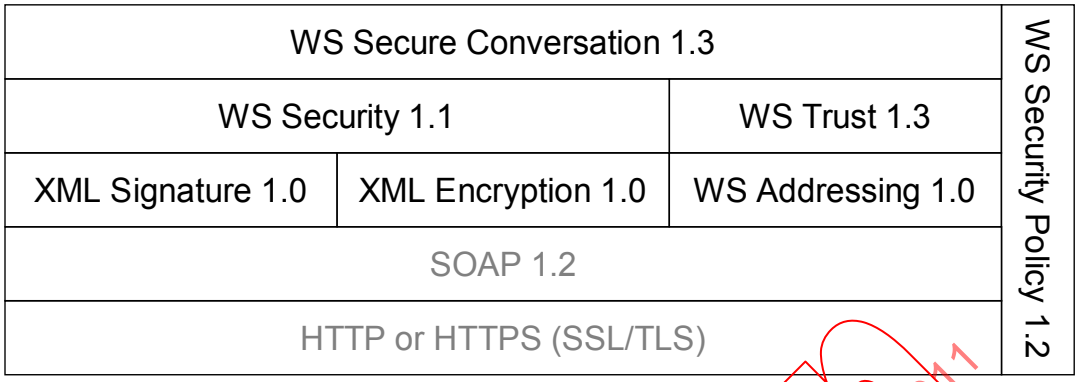


Figure 11 – Relevant XML Web Services Specifications

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

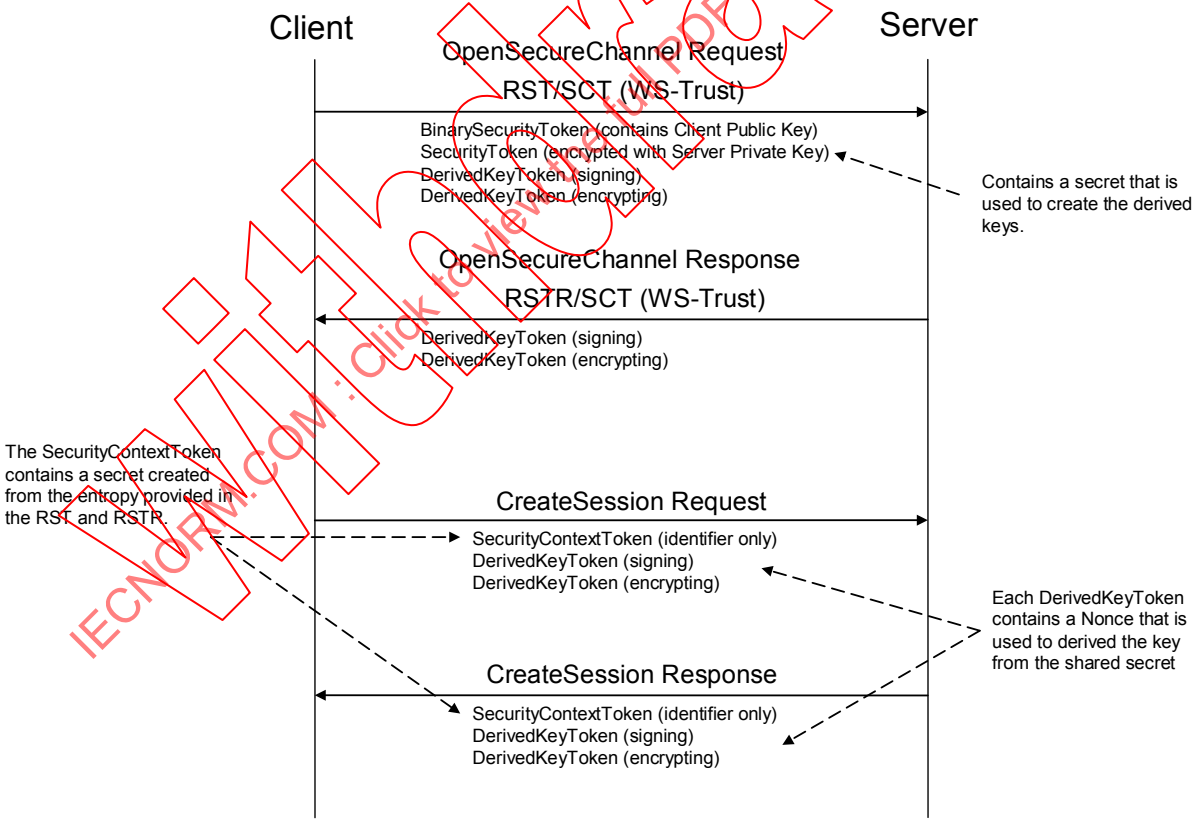


Figure 12 – The WS Secure Conversation Handshake

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.

6.3.2 Notation

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

Table 25 – WS-* Namespace Prefixes

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

6.3.3 Request Security Token (RST/SCT)

The Request Security Token message implements the abstract OpenSecureChannel request message defined in 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 token 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 26.

Table 26 – RST/SCT Mapping to an OpenSecureChannel Request

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

6.3.4 Request Security Token Response (RSTR/SCT)

The Request Security Token Response message implements the abstract OpenSecureChannel response message defined in 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 27.

Table 27 – RSTR/SCT Mapping to an OpenSecureChannel Response

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

The lifetime specifies the UTC expiration time for the security context token. The client shall renew the SCT before that time by sending the RST/SCT message again. The exact behaviour is described in IEC 62541-4, 5.5.

6.3.5 Using the SCT

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

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

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

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

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.

6.3.6 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.

6.4 OPC UA Secure Conversation

6.4.1 Overview

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

UASC is designed to operate with different *TransportProtocols* that may have limited buffer sizes. For this reason, OPC UA Secure Conversation will break OPC UA messages into several pieces (called '*MessageChunks*') that are smaller than the buffer size allowed by the *TransportProtocol*. UASC requires a *TransportProtocol* buffer size that is at least 8 196 bytes.

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

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

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

6.4.2 MessageChunk Structure

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

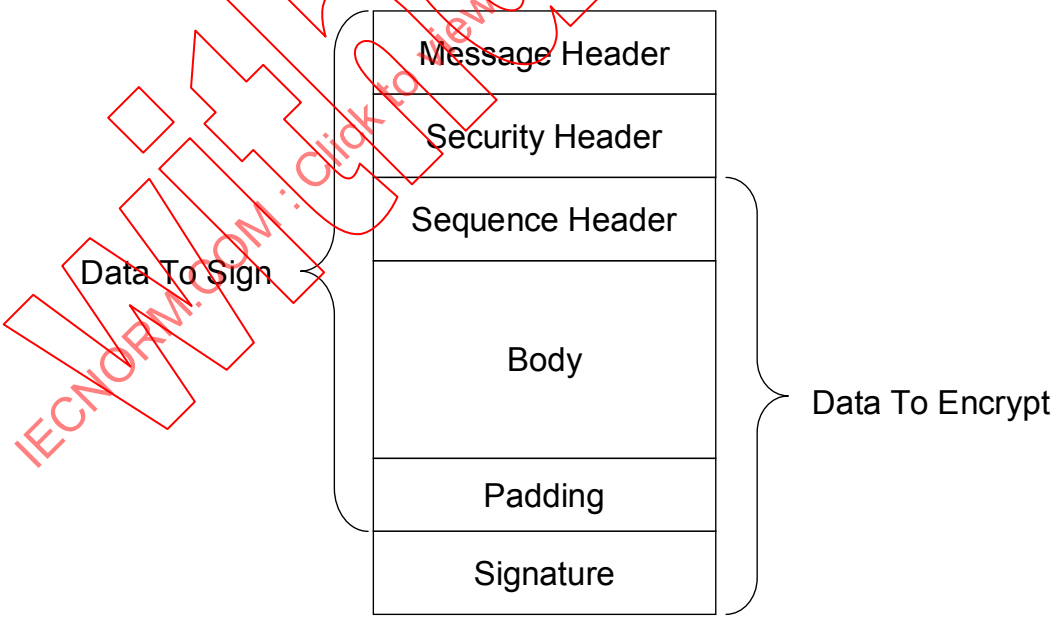


Figure 13 – OPC UA Secure Conversation MessageChunk

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

Table 28 – OPC UA Secure Conversation Message Header

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

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

Table 29 – Asymmetric Algorithm Security Header

Name	Data Type	Description
SecurityPolicyUriLength	Int32	The length of the SecurityPolicyUri in bytes.This value shall not exceed 255 bytes.
SecurityPolicyUri	Byte[*]	The URI of the security policy used to secure the message. This field is encoded as a UTF8 string without a null terminator.
SenderCertificateLength	Int32	The length of the SenderCertificate in bytes. This value shall not exceed MaxCertificateSize bytes.
SenderCertificate	Byte[*]	The X509v3 certificate assigned to the sending application instance. This is a DER encoded blob. The structure of an X509 certificate is defined in ITU-T X.509. The DER format for a certificate is defined in ITU-T X.690. This indicates what private key was used to sign the MessageChunk. The Stack shall close the channel and report an error to the application if the *SenderCertificate* is too large for the buffer size supported by the transport layer. This field shall be null if the message is not signed.
ReceiverCertificateThumbprintLength	Int32	The length of the ReceiverCertificateThumbprint in bytes. The length of this field is always 20 bytes.
ReceiverCertificateThumbprint	Byte[*]	The thumbprint of the X509v3 certificate assigned to the receiving application instance. The thumbprint is the SHA1 digest of the DER encoded form of the certificate. This indicates what public key was used to encrypt the MessageChunk. This field shall be null if the message is not encrypted.

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

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

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

The *MessageChunkSize* depends on the transport protocol but shall be at least 8 196 bytes. The *AsymmetricSignatureSize* depends on the number of bits in the public key for the *SenderCertificate*. The *Int32FieldLength* is the length of an encoded Int32 value and it is always 4 bytes.

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

Table 30 – Symmetric Algorithm Security Header

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

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

Table 31 – Sequence Header

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

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

The *SequenceNumber* shall also monotonically increase for all messages and shall not wrap around until it is greater than 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 re-established. Note that the first *MessageChunk* after a network interruption is always an *OpenSecureChannel* request or response.

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

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

Table 32 – OPC UA Secure Conversation Message Footer

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

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

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

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

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

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

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

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

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

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

6.4.3 MessageChunks and Error Handling

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

Table 33 – OPC UA Secure Conversation Message Abort Body

Name	Data Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 40.
Reason	String	A more verbose description of the error. This string shall not be more than 4 096 characters. A client shall ignore strings that are longer than this.

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

6.4.4 Establishing a SecureChannel

Most messages require a *SecureChannel* to be established. A client does this by sending an *OpenSecureChannel* request to the server. The server shall validate the message and the *ClientCertificate* and return an *OpenSecureChannel* response. Some of the parameters defined for the *OpenSecureChannel* service are specified in the security header (see 6.4.2) instead of the body of the message. For this reason, the *OpenSecureChannel* service is not the same as the one specified in IEC 62541-4. Table 34 lists the parameters that appear in the body of the message.

Table 34 – OPC UA Secure Conversation OpenSecureChannel Service

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

The *ClientProtocolVersion* and *ServerProtocolVersion* parameters are not defined in 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 specifed in the OPC UA-TCP protocol *Hello/Acknowledge* messages. The receiver shall close the channel and report a *Bad_ProtocolVersionUnsupported* error if there is a mismatch.

The server shall return an error response as described in 5.3 of 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 token by sending another *OpenSecureChannel* request. The client shall continue to accept the old token until it receives

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

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

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

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

If an error occurs after the Server has verified message security it shall return a *ServiceFault* instead of a *OpenSecureChannel* response. The *ServiceFault* message is described in IEC 62541-4, 7.28.

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

6.4.5 Deriving Keys

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

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

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

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

a) SigningKeyLength (from the DerivedSignatureKeyLength);

b) EncryptingKeyLength (implied by the SymmetricEncryptionAlgorithm);

c) EncryptingBlockSize (implied by the SymmetricEncryptionAlgorithm).

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

Table 35 – Cryptography Key Generation Parameters

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

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

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

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

6.4.6 Verifying Message Security

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

If 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 token by sending an *OpenSecureChannel* request. The mechanism for sending transport errors to the Client depends on the communication channel.

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

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

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

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

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

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

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

7 Transport Protocols

7.1 OPC UA TCP

7.1.1 Overview

OPC UA TCP is a simple TCP based protocol that establishes a full duplex channel between a client and server. This protocol has two key features that differentiate it from HTTP. First, this protocol allows responses to be returned in any order. Second, this protocol allows responses to be returned on a different TCP transport end-point if communication failures causes 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 36.

Table 36 – OPC UA TCP Message Header

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

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

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

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

Table 37 – OPC UA TCP Hello Message

Name	Data Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the *Client*. The *Server* may reject the *Client* by returning *Bad_ProtocolVersionUnsupported*. If the *Server* accepts the connection, it (the server) is responsible for ensuring that it returns messages that conform to this version of the protocol. The *Server* shall always accept versions greater than what it supports.
ReceiveBufferSize	UInt32	The largest message that the sender can receive. This value shall be greater than 8 192 bytes.
SendBufferSize	UInt32	The largest message 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 6.4.3. The message size is calculated using the unencrypted message body. A value of zero indicates that the Client has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any response message. The server shall abort the message with a *Bad_ResponseTooLarge StatusCode* if a response message exceeds this value. The mechanism for aborting messages is described fully in 6.4.3. A value of zero indicates that the Client has no limit.
EndpointUrl	String	The URL of the Endpoint which the Client wished to connect to. The encoded value shall be less than 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 38.

Table 38 – OPC UA TCP Acknowledge Message

Name	Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the *Server*. If the *Client* accepts the connection, he is responsible for ensuring that he sends messages that conform to this version of the protocol. The *Client* shall always accept versions greater than what he supports.
ReceiveBufferSize	UInt32	The largest message that the sender can receive. This value shall not be larger than what the Client requested in the Hello message. This value shall be greater than 8 192 bytes.
SendBufferSize	UInt32	The largest message that the sender will send. This value shall not be larger than what the Client requested in the Hello message. This value shall be greater than 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 6.4.3. The message size is calculated using the unencrypted message body. A value of zero indicates that the *Server* has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any request message. The client shall abort the message with a *Bad_RequestTooLarge StatusCode* if a request message exceeds this value. The mechanism for aborting messages is described fully in 6.4.3. A value of zero indicates that the *Server* has no limit.

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

Table 39 – OPC UA TCP Error Message

Name	Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 40.
Reason	String	A more verbose description of the error. This string shall not be more than 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 illustration of how the message elements defined by the OPC UA Binary Encoding mapping (see 5.2) and the OPC UA Secure Conversation mapping (see 6.4) relate to the OPC UA TCP messages.

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

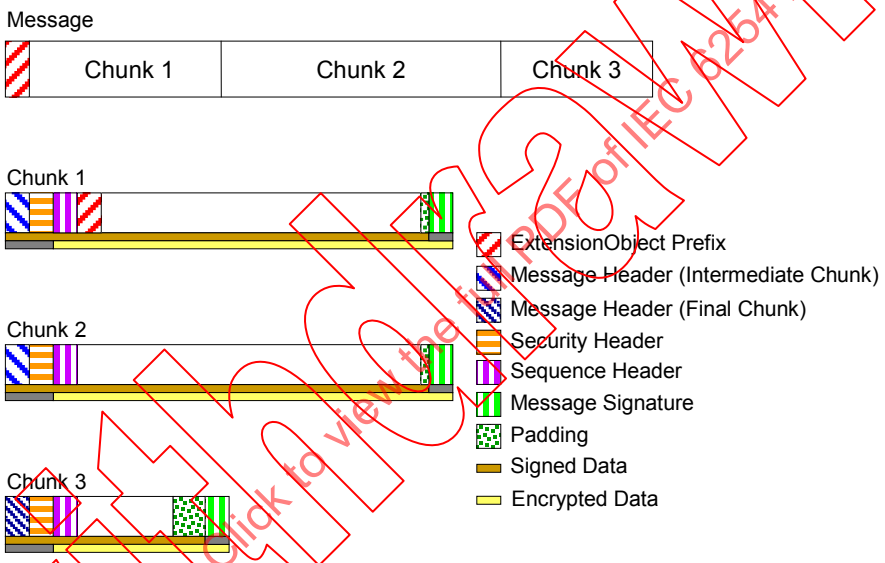


Figure 14 – OPC UA TCP Message Structure

7.1.3 Establishing a Connection

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

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

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

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

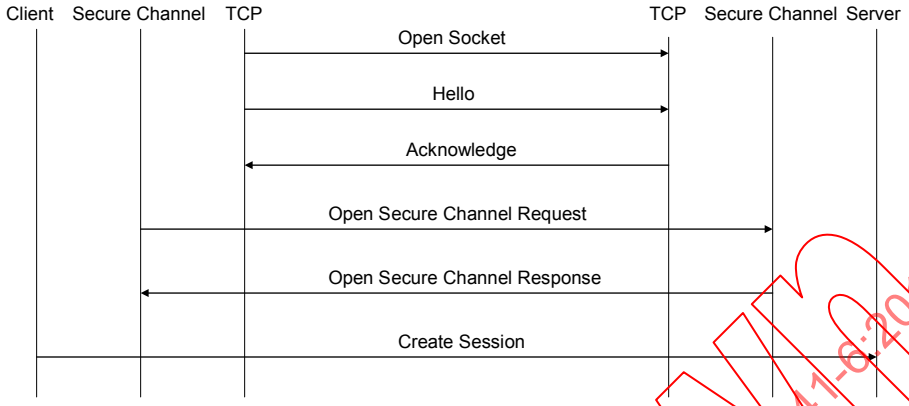


Figure 15 – Establishing a OPC UA TCP Connection

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

7.1.4 Closing a Connection

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

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

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

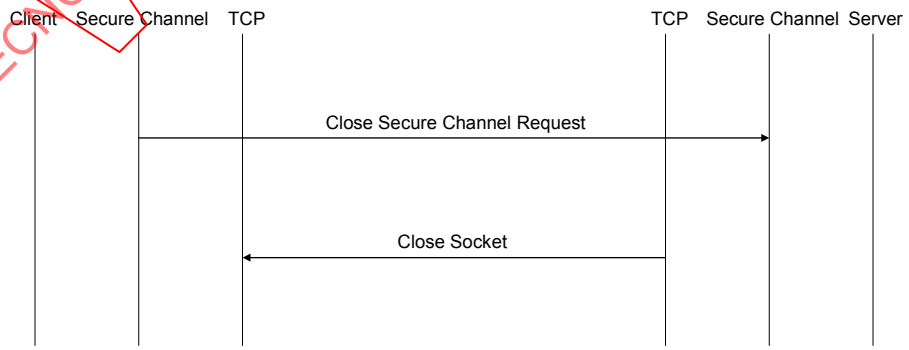


Figure 16 – Closing a OPC UA TCP Connection

The *Server* application does not do any processing when the *SecureChannel* is closed, however, the *Stack* shall provide notifications to the *Server* application whenever a

CloseSecureChannel request is received or when the *Stack* cleans up an abandoned *SecureChannel*.

7.1.5 Error Handling

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

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

Table 40 – OPC UA TCP Error Codes

Name	Description
TcpServerTooBusy	The server cannot process the request because it is too busy. It is up to the server to determine when it needs to return this message. A server can control 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 how frequently a client reconnects by waiting to return this error.
TcpInternalError	An internal error occurred. This should only be returned if an unexpected configuration or programming error occurs.
TcpUrlRejected	The Server does not recognize the EndpointUrl specified.
SecurityChecksFailed	The message was rejected because it could not be verified.
RequestInterrupted	The request could not be sent because of a network interruption.
RequestTimeout	Timeout occurred while processing the request.
SecureChannelClosed	The secure channel has been closed.
SecureChannelTokenUnknown	The token has expired or is not recognized.
CertificateUntrusted	The sender certificate is not trusted by the receiver.
CertificateTimeInvalid	The sender certificate has expired or is not yet valid.
CertificateIssuerTimeInvalid	The issuer for the sender certificate has expired or is not yet valid.
CertificateUseNotAllowed	The sender's certificate may not be used for establishing a secure channel.
CertificateIssuerUseNotAllowed	The issuer certificate may not be used as a Certificate Authority.
CertificateRevocationUnknown	Could not verify the revocation status of the sender's certificate.
CertificateIssuerRevocationUnknown	Could not verify the revocation status of the issuer certificate.
CertificateRevoked	The sender certificate has been revoked by the issuer.
IssuerCertificateRevoked	The issuer certificate has been revoked by its issuer.
CertificateUnknown	The receiver certificate thumbprint is not recognized by the receiver.

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

7.1.6 Error Recovery

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

The client shall wait between reconnect attempts. The first reconnect shall happen immediately. After that, the wait period should start as 1 second and increase gradually to a maximum of 2 minutes. One sequence would double the period each attempt until reaching the maximum. In other words, the client would use the following wait periods: { 0, 1, 2, 4, 8, 16, 32, 64, 120, 120, ...}. The client shall keep attempting to reconnect until the *SecureChannel* is closed or after the period equal to the RevisedLifetime of the last SecurityToken elapses.

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

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

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

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

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

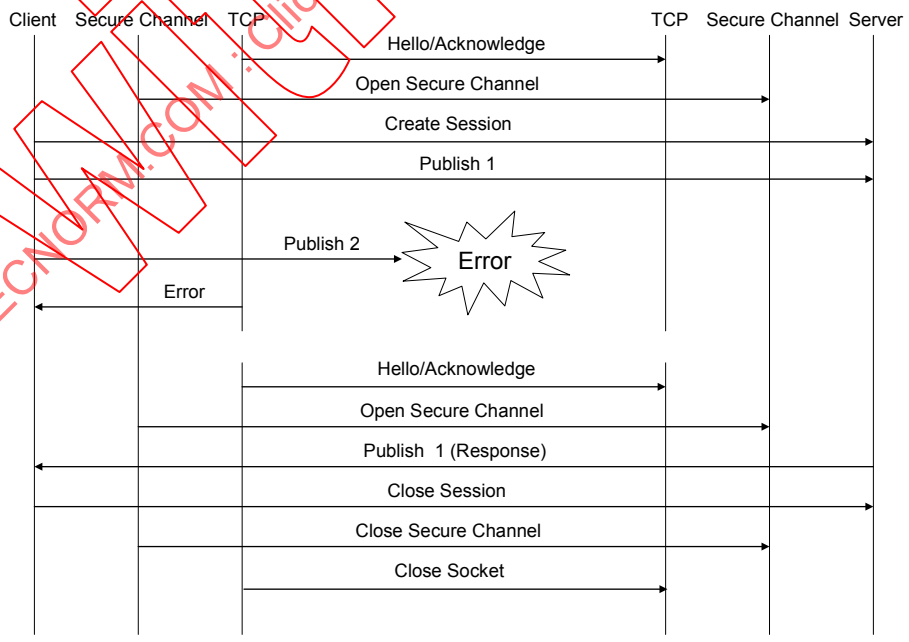


Figure 17 – Recovering an OPC UA TCP Connection

7.2 SOAP/HTTP

7.2.1 Overview

SOAP describes an XML based syntax for exchanging messages between applications. OPC UA messages are exchanged using SOAP by serializing the OPC UA messages using one of the supported encodings described in 5.3 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 Version 1.1 defines best practices when using SOAP messages which will help ensure interoperability. All OPC UA implementations shall conform to this specification.

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

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

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

SOAP faults are returned only for errors that occurred with in the SOAP stack. Errors that occur within the application are returned as OPC UA error response messages as described in 5.3 of IEC 62541-4.

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

Table 41 – WS-Addressing Headers

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

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

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

7.2.2 XML Encoding

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

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

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

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

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

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

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

7.2.3 OPC UA Binary Encoding

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

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

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

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

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

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

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

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

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

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

7.3 Well Known Addresses

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

Table 42 – Well Known Addresses for Local Discovery Servers

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

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

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

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

8 Normative Contracts

8.1 OPC Binary Schema

The normative contract for the OPC UA Binary encoded messages is an OPC Binary Schema. This file defines the structure of all types and messages. The syntax for an OPC Binary Type Schema is described in Annex B of IEC 62541-3. This schema captures normative names for types and their fields as well as 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 as the order the fields appear when encoded. The data type of each field is also captured.

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

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

Annex A
(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 this series of OPC UA standards. The identifiers are specified in a CSV file with the following syntax:

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

Where the *SymbolName* is the literal name for the error code that appears in the specification and the *SubCode* is numeric value for the *SubCode* field within a *StatusCode* (see 7.33 in IEC 62541-4). The severity associated with particular code is specified by the prefix (*Good*, *Uncertain* or *Bad*).

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

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

The most recent set of StatusCodes can be found here:

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

A.3 Numeric Node Ids

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

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

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

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

Annex B
(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 Annex C of IEC 62541-3.

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

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

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

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

Annex C
(normative)

WSDL for the XML Mapping

C.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 associated with this version of the standards can be found here:

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

The most recent XML Schema can be found here:

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

C.2 WDSL Port Types

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

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

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

The most recent WSDL can be found here:

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

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

C.3 WSDL Bindings

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

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

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

The most recent WSDL can be found here:

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

This WSDL imports the WSDL defined in C.2.

Annex D
(normative)

Security Settings Management

D.1 Overview

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

D.2 SecuredApplication

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

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

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

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

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

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

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

Table D.1 – SecuredApplication

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

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

D.3 CertificateIdentfier

The CertificateIdentifier element describes a X509 Certificate. The Certificate can be provided explicitly within the element or the element can specify the location of the CertificateStore that contains the Certificate. The elements contained in a CertificateIdentifier are described in Table D.2.

Table D.2 – CertificateIdentfier

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

A "Windows" StoreType specifies a Windows certificate store.

The syntax of the StorePath has the form:

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

where

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

StoreLocation - one of LocalMachine, CurrentUser, User or Service

ServiceName - the name of a Windows Service.

UserSid - the SID for a Windows user account.

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

Examples of Windows StorePaths are:

\\MYPC\LocalMachine\My

\CurrentUser\Trust

\\MYPC\Service\My UA Server\UA Applications
\User\S-1-5-25\Root

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

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

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

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

It is recommended that the SubjectName always be specified.

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

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

D.4 CertificateStoreIdentifier

The CertificateStoreIdentifier element describes a physical store containing X509 Certificates. The elements contained in a CertificateIdentifier are described in Table D.3.

Table D.3 – CertificateStoreIdentfier

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the Certificate. Predefined values are "Windows", "Directory" and "OpenSSL".
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType. See D.3 for a description of the syntax for different StoreTypes.
ValidationOptions	Int32	The options to use when validating the Certificates contained in the store. The possible options are described in D.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.

D.5 CertificateTrust List

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

Table D.4 – CertificateTrustList

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

The StoreType and StorePath elements are ignored for CertificateTrustLists.

D.6 CertificateValidationOptions

The CertificateValidationOptions control the process used to validate a Certificate. Any Certificate can have validation options associated with it if it is placed in TrustedPeerCertificates or TrustedIssuerCertificates lists. The possible values are described in Table D.5.

Table D.5 – CertificateValidationOptions

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

D.7 ApplicationAccessRule

ApplicationAccessRule specifies the access rights to an Application granted to a user or group. The mechanisms used to enforce access rules depend on the operating system. The elements in an ApplicationAccessRule are described in Table D.6.

Table D.6 – ApplicationAccessRule

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

D.8 ApplicationSecurityPolicy

ApplicationSecurityPolicy specifies the security policies permitted by an Application. The elements in an ApplicationSecurityPolicy are described in Table D.7.

Table D.7 – ApplicationSecurityPolicy

Field	Type	Description
UrlScheme	String	The URL scheme that the policy applies to. If blank, it applies to all types of endpoints.
Address	String	The address that the policy applies to. If this is a complete URL, then the policy only applies to the specified URL. Non-URLs may be used to construct URLs from the BaseAddresses. An URL that does not start with one of the BaseAddresses is ignored.
SecurityPolicyUri	String	The URI of the SecurityPolicy to use. These URIs are defined in IEC 62541-7.
SecurityModes	Int32	A mask that specifies the security modes that are permitted. The following value are permitted: 0x1 - Sign 0x2 - SignAndEncrypt A value of 0 indicates that no security is used.
SecurityLevel	Byte	A relative measure of the security provided by the policy. A higher number means better security. A value of 0 indicates the policy is not recommended.

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

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

SOMMAIRE

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIEE OPC –

Partie 6: Correspondances

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (CEI) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de la CEI). La CEI a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, la CEI – 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 la CEI"). 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 la CEI, participent également aux travaux. La CEI 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 la CEI concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de la CEI intéressés sont représentés dans chaque comité d'études.
3) Les Publications de la CEI se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de la CEI. Tous les efforts raisonnables sont entrepris afin que la CEI s'assure de l'exactitude du contenu technique de ses publications; la CEI 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 la CEI s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de la CEI dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de la CEI et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) La CEI 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 la CEI. La CEI 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 la CEI, 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 la CEI, 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 la CEI ou de toute autre Publication de la CEI, 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 la CEI peuvent faire l'objet de droits de brevet. La CEI 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 CEI 62541-6 a été élaborée par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de la CEI: Mesure, commande et automation dans les processus industriels.

Le texte de cette norme est issu des documents suivants:

FDIS	Rapport de vote
65E/193/FDIS	65E/215/RVD

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/CEI, Partie 2.

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

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 la CEI sous "http://webstore.iec.ch" dans les données relatives à la publication recherchée. A 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.

INTRODUCTION

La présente Norme internationale est une spécification destinée aux développeurs d'applications OPC UA. La spécification est le résultat d'un processus d'analyse et de conception visant à développer une interface normalisée qui facilite l'interopérabilité d'applications issues de divers fournisseurs.

ARCHITECTURE UNIFIEE OPC –

Partie 6: Correspondances

1 Domaine d'application

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

2 Références normatives

Les documents de référence suivants sont indispensables pour l'application du présent document. 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* (disponbile uniquement en anglais)

IEC 62541-2, *OPC Unified architecture: Part 2 – Security Model* (disponible uniquement en anglais)

IEC 62541-3, *OPC Unified architecture: Part 3 – Address Space Model* (disponible uniquement en anglais)

IEC 62541-4[1] :----, *OPC Unified architecture: Part 4 – Services* (disponbile uniquement en anglais)

CEI 62541-5[2], *Architecture unifiée OPC: Partie 5: Modèle d'Information*

CEI 62541-7[3], *Architecture unifiée OPC: Partie 7: Profils*

ITU-T X.690: *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

available at <http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>

ITU-T X.200: *Information technology – Open Systems Interconnection – Basic Reference Model*

available at <http://www.itu.int/rec/T-REC-X.200-199407-I/en>

ITU-T X.509: *Information technology – Open Systems Interconnection – The directory: Public Key and Attribute Certificate Frameworks*

available at <http://www.itu.int/rec/T-REC-X.509/en>

XML Schema Part 1: *XML Schema Part 1: Structures (Second Edition)*

---

1 A publier.

2 A publier.

3 A publier.

available at <http://www.w3.org/TR/xmlschema-1/>

XML Schema Part 2: *XML Schema Part 2: Datatypes (Second Edition)*

available at <http://www.w3.org/TR/xmlschema-2/>

SOAP Part 1: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*

available at <http://www.w3.org/TR/soap12-part1/>

SOAP Part 2: *SOAP Version 1.2 Part 2: Adjuncts (Second Edition)*

available at <http://www.w3.org/TR/soap12-part2/>

XML Encryption: *XML Encryption Syntax and Processing*

available at <http://www.w3.org/TR/xmlenc-core/>

XML Signature: *XML-Signature Syntax and Processing (Second Edition)*

available at <http://www.w3.org/TR/xmldsig-core/>

WS Security: *SOAP Message Security 1.1*

available at <http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>

WS Addressing: *Web Services Addressing (WS-Addressing)*

available at <http://www.w3.org/Submission/ws-addressing/>

WS Trust: *WS Trust 1.3*

available at <http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html>

WS Secure Conversation: *WS Secure Conversation 1.3*

available at <http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html>

WS Security Policy: *WS Security Policy 1.2*

available at <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>

SSL/TLS: RFC 2246 - *The TLS Protocol Version 1.0*

available at <http://www.ietf.org/rfc/rfc2246.txt>

WS-I *Basic Profile Version 1.1*

available at <http://www.ws-i.org/Profiles/BasicProfile-1.1.html>

WS-I *Basic Security Profile Version 1.1*

available at <http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html>

HTTP: RFC 2616 - *Hypertext Transfer Protocol - HTTP/1.1*

available at <http://www.ietf.org/rfc/rfc2616.txt>

HTTPS: RFC 2818 - *HTTP Over TLS*

available at <http://www.ietf.org/rfc/rfc2818.txt>

Base64: RFC 3548 - *The Base16, Base32, and Base64 Data Encodings*

availble at <http://www.ietf.org/rfc/rfc3548.txt>

IEEE-754: *Standard for Binary Floating-Point Arithmetic*

available at <http://grouper.ieee.org/groups/754/>

HMAC: RFC 2104 - *HMAC - Keyed-Hashing for Message Authentication*

available at <http://www.ietf.org/rfc/rfc2104.txt>

PKCS #1 : RFC 2437 - *PKCS #1 - RSA Cryptography Specifications Version 2.0*
available at <http://www.ietf.org/rfc/rfc2437.txt>

PKCS #12 : *PKCS 12 v1.0: Personal Information Exchange Syntax*
available at <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf>

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

FIPS 197: *Advanced Encyption Standard (AES)*
available at <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

UTF8: RFC 3629 - *UTF-8, a transformation format of ISO 10646*
available at <http://tools.ietf.org/html/rfc3629>

RFC 3280: *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*
available at <http://www.ietf.org/rfc/rfc3280.txt>

RFC 4514: *LDAP: String Representation of Distinguished Names*
available at <http://www.ietf.org/rfc/rfc4514.txt>

3 Termes, définitions et abréviations

3.1 Termes et définitions

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

3.1.1
codage de données (Data Encoding)
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 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
pile (Stack)
regroupement de bibliothèques logicielles qui mettent en œuvre un ou plusieurs Profils de Piles (Stack Profiles); les Piles comportent une interface API qui masque les détails de mise en œuvre du développeur d'applications

3.1.5
profil de pile (Stack Profile)
combinaison des correspondances Codages de Données, Protocole de Sécurité et Protocole de Transport

NOTE 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.

3.1.6
protocole de transport (TransportProtocol)
protocole qui représente une méthode d'échange des messages OPC UA sérialisés entre des applications OPC UA

3.2 Abréviations

API	Interface de programme d'application (Application Programming Interface)
ASN.1	Notation syntaxique abstraite #1 (Abstract Syntax Notation) (utilisée dans la recommandation ITU-T X.690)
BP	Version de profil de base WS-I (Basic Profile)
BSP	Profil de sécurité de base WS-I (Basic Security Profile)
CSV	Valeur séparée par des virgules (Comma Separated Value) (format de fichier)
HTTP	Protocole de Transport hypertexte (Hypertext Transfer Protocol)
IPSec	Sécurité de protocole Internet (Internet Protocol Security)
RST	Requête de jeton de sécurité (Request Security Token)
OID	Identifiant d'objet (Object Identifier) (utilisé avec ASN.1)
RSTR	Réponse à une requête de jeton de sécurité (Request Security Token Response)
SCT	Jeton de contexte de sécurité (Security Context Token)
SHA1	Algorithme de hachage sécurisé (Secure Hash Algorithm)
SOAP	Protocole SOAP (protocole d'accès d'objet simple) (Single Object Access Protocol)
SSL	Protocole SSL (Protocole de sécurisation) (Secure Sockets Layer) (défini en SSL/TLS)
TCP	Protocole TCP (protocole de contrôle de transmission) (Transmission Control Protocol)
TLS	Protocole TLS (Transport Layer Security) (défini en SSL/TLS)
UTF8	Format de transformation Unicode (Unicode Transformation Format) (8 bits) (Défini en UTF8)
UA	Architecture unifiée (Unified Architecture)
UASC	Conversation sécurisée UA (UA Secure Conversation)
WS-*	Spécifications des services Web XML
WSS	Sécurité WS (WS Security)
WS-SC	Conversation sécurisée WS (WS Secure Conversation)
XML	Langage de balisage extensible (eXtensible Markup Language)

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 CEI 62541-1 à 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 pouvant 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*.

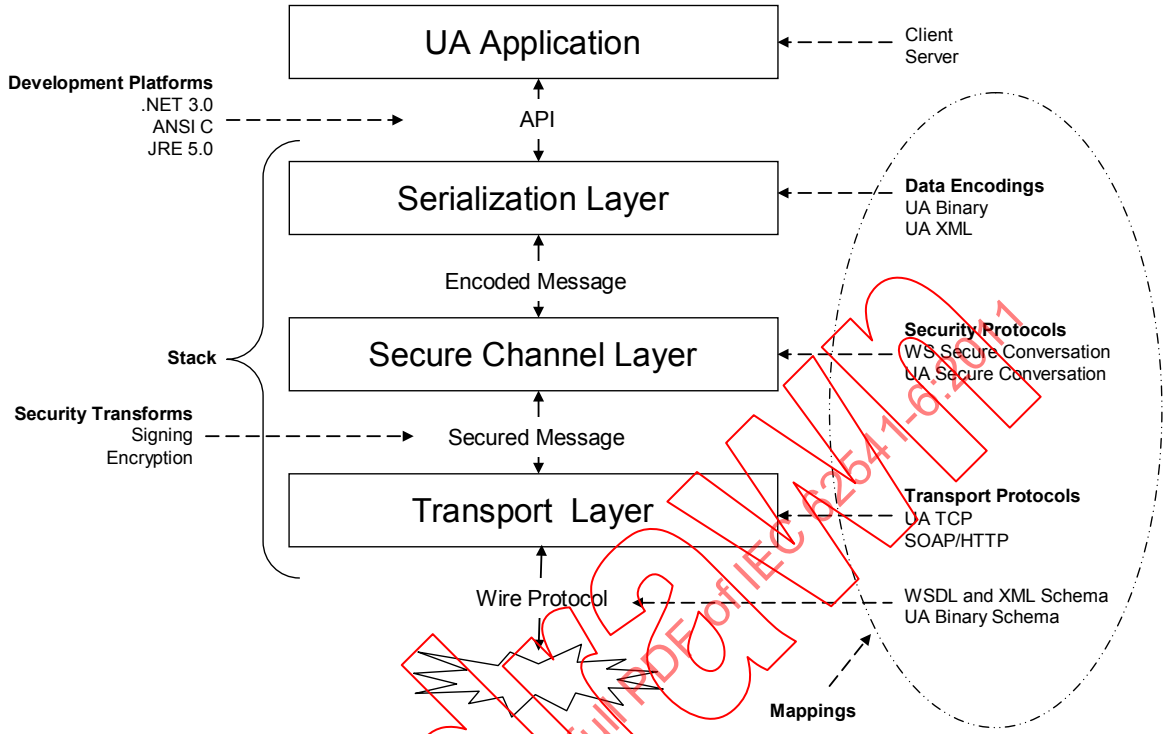
Toutes les applications OPC UA doivent mettre en œuvre au moins un *Profil de Pile* et peuvent communiquer uniquement avec d'autres applications OPC UA qui mettent en œuvre le même *Profil de Pile*.

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* dans l'Article 7. Les *Profils de Piles* sont définis dans la CEI 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 la IEC 62541-4. Toutefois, leur format est spécifié par le *Codage de Données* et le *Protocole de Transport*. Ainsi, chaque *Message* défini dans la 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. A 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 les entiers non signés, ce qui signifie que toute interface API Java doit mettre 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.



Légende

Anglais	Français
Development platforms	Plates-formes de développement
UA Application	Application UA
Client Server	Serveur client
API	API
Serialization layer	Couche de sérialisation
Data Encodings/ UA Binary / UA XML	Codages de données / UA Binaire / UA XML
Encoded Message	Message codé
Stack	Pile
Secure Channel Layer	Couche de canal sécurisé
Security Protocols	Protocoles de sécurité
WS / UA Secure Conversation	Conversation Sécurisée WS / UA
Security Transforms / Signing / Encryption	Transformées de sécurité / Signature / Cryptage
Secured Message	Message sécurisé
Transport Layer	Couche de transport
Transport Protocols	Protocoles de transport
UA TCP	TCP UA
SOAP/HTTP	SOAP/HTTP
Wire Protocol	Protocole par fil
WSDL et XML Schema	WDSL et Schéma XML
UA Binary Schema	Schéma UA Binaire
Mappings	Correspondances

Figure 1 – Aperçu général des piles OPC UA

Les couches décrites dans la présente spécification ne correspondent pas aux couches du modèle OSI 7 [ITU-T X.200]. Il convient de traiter chaque *Profil de Pile* OPC UA comme un

protocole avec une Couche Application unique bâtie sur un protocole existant (Couches 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 identifiant unique. Les Utilisateurs et les Administrateurs doivent 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 (Identifiant)	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 pouvant être utilisée comme identifiant globalement unique.
15	ByteString	Séquence d'octets.
16	XmlElement	Un élément XML.
17	NodeIdentifier	Identifiant d'un nœud dans l'espace d'adresse d'un serveur OPC UA.
18	ExtendedNodeId	Identifiant de Nœud permettant de spécifier l'espace de nom URI 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 identifiant de localisation géographique 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 d'état 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 d'Etat

La plupart de ces types de données sont identiques aux types abstraits définis dans les IEC 62541-3 et 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 (Identifiant Globalement Unique)* défini dans la IEC 62541-3.

5.1.3 Guid (Identifiant globalement Unique)

Identifiant 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 Objet d'Extension

Un *Objet d'Extension* contient tous types de données complexes qui ne peuvent pas être codés par un type 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 identifiant 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'adresse du Serveur sous forme de sous-types du type de données de Structure. Les codages disponibles pour tout type de données complexe sont représentés comme un Objet de Codage de Type de Données dans l'espace d'adresse du Serveur. L'*Identifiant de Nœud* de l'Objet de Codage de Type de Données est l'identifiant archivé dans l'Objet d'Extension. Le paragraphe 5.8 de la IEC 62541-3 décrit comment les nœuds Codage de Type de Données sont liés aux autres nœuds de l'espace d'adresse.

Il faut que les ingénieurs qui implémentent des *Serveurs* utilisent des *Identifiants 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.*

5.1.5 Variante (Variant)

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 archiver 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 qui est traitée 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*. C'est la raison pour laquelle tous les *Codages de Données* doivent maintenir cette distinction lors du codage des *Variantes*.

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

Le type *Information de Diagnostic* a une signification que lorsqu'il est renvoyé dans un message de réponse avec un *Code d'Etat* associé. De ce fait, les *Variantes* ne peuvent pas contenir des instances 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 Binaire 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 rapide. Toutefois, il a également été tenu compte de la taille des données codées mises sur le réseau.

Le Codage OPC UA Binaire 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 Binaire 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 identifiant et une taille pour la structure complexe 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 » 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.

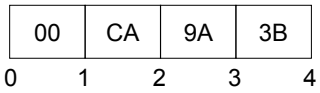


Figure 2 – Codage des entiers dans une séquence binaire

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 endians).

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 IEEEE 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.

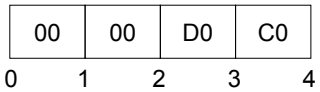


Figure 3 – Codage des virgules flottantes dans une séquence binaire

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.

Length				水			B	o	y
06	00	00	00	E6	B0	B4	42	6F	79
0	1	2	3	4	5	6	7	8	9	10

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 ne sont pas toutes capables de représenter la plage complète de dates et d'heures pouvant être représentée par ce codage. Par exemple, la structure time_t sous UNIX a uniquement une résolution de 1 s 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. 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 pouvant être représentée avec le codage de la plate-forme.
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 pouvant être représentée avec le codage de la plate-forme.
c) Une valeur date/heure est décodée comme l'heure la plus proche pouvant 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 pouvant être représentée avec le codage de la plate-forme.
d) Une valeur date/heure est décodée comme l'heure la plus éloignée pouvant 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 pouvant être représentée avec le codage de la plate-forme.

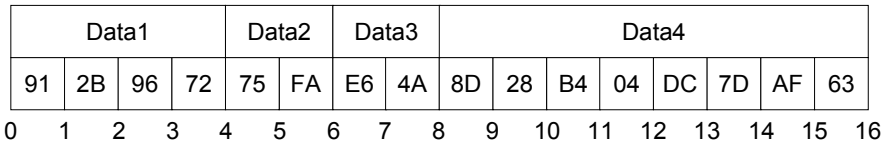
Ces règles impliquent que les heures les plus proches et les plus lointaines qui peuvent être représentées sur une plate-forme donnée sont des valeurs de date/heure non valides, qu'il convient de traiter comme tel par les applications.

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.

5.2.2.6 Guid (Identifiant 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.



Légende

Anglais	Français
Data	Données

Figure 5 – Codage des Guid dans une séquence binaire

5.2.2.7 Chaîne d'Octets (ByteString)

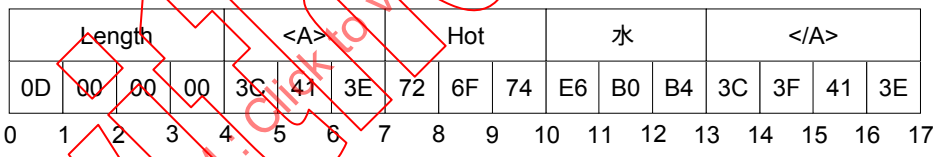
Une *Chaîne d'Octets* est codée comme une séquence d'octets précédée de sa longueur en octets. La longueur est codée comme un entier signé de 32 bits tel que décrit ci-dessus.

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

5.2.2.8 Elément Xml (XmlElement)

Un *Elément Xml* est un fragment XML sérialisé sous forme d'une chaîne UTF-8, 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.



Légende

Anglais	Français
Length	Longueur
Hot	Chaud

Figure 6 – Codage des Eléments Xml dans une séquence binaire

5.2.2.9 Identifiant de Nœud (NodeId)

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

Tableau 4 – Composants d'un Identifiant 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 *Identifiants de Nœud* définis OPC UA.
IdentifierType	Enum	Le format et le type de données de l'identifiant. 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	*	Identifiant d'un nœud dans l'espace d'adresse d'un serveur OPC UA.

Le codage d'un *Identifiant de Nœud* varie selon le contenu de l'instance. Ainsi, le premier octet de la forme codée indique le format du reste de l'*Identifiant de Nœud* codé. Les formats de codage 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 l'Identifiant de Nœud

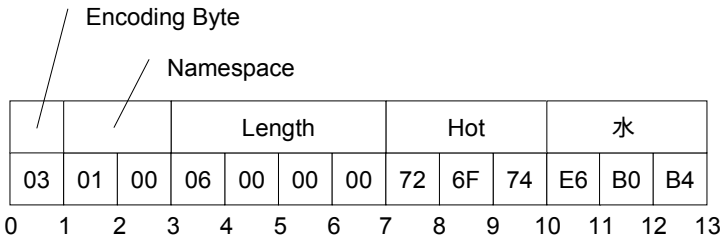
Dénomination	Valeur	Description
Two Byte	0x00	Valeur numérique ajustée à la représentation à deux octets.
Four Byte	0x01	Valeur numérique ajustée à la représentation à quatre octets.
Numeric	0x02	Valeur numérique non ajustée à la représentation à deux ou à quatre octets.
String	0x03	Valeur Chaîne.
Guid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

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

Tableau 6 – Codage binaire normalisé d'*Identifiant de Nœud*

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

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



Légende

Anglais	Français
Encoding Byte	Octet de Codage
Namespace	Espace de nom
Length	Longueur
Hot	Chaud

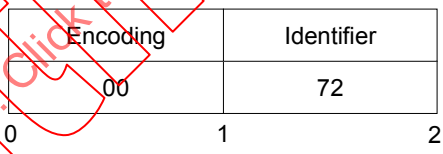
Figure 7 – Identifiant de Nœud de chaîne

La structure du codage de l'*Identifiant de Nœud* à deux octets est décrite dans le Tableau 7.

Tableau 7 – Codage binaire de l'Identifiant 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'identifiant est « Numérique ». L'identifiant doit se situer dans la plage comprise entre 0 et 255.

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



Légende

Anglais	Français
Encoding	Codage
Identifier	Identifiant

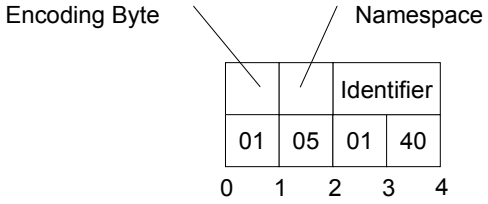
Figure 8 – Identifiant de Nœud à deux octets

La structure du codage de l'*Identifiant de Nœud* à quatre octets est décrite dans le Tableau 8.

Tableau 8 – Codage binaire de l'Identifiant 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'identifiant est « Numérique ». L'identifiant doit être un entier situé dans la plage comprise entre 0 et 65 535.

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



Légende

Anglais	Français
Anglais	Français
Encoding Byte	Octet de Codage
Namespace	Espace de nom
Identifier	Identifiant

Figure 9 – Identifiant de Nœud à quatre octets

5.2.2.10 Identifiant de Nœud Etendu (ExpandedNodeId)

Un *Identifiant de Nœud Etendu* étend la structure de l'*Identifiant 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'*Identifiant de Nœud* doit être ignoré.

Le codage de l'*Identifiant de Nœud Etendu* s'effectue tout d'abord par le codage d'un *Identifiant 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 *Identifiant 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 de l'indicateur *Uri d'Espace de Nom* dans l'octet du format de codage propre à l'*Identifiant 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 *Identifiant de Nœud* est codée. Pour être cohérent l'*Indice d'Espace de Nom* non utilisé est inclus dans la séquence des bits.

Un *Identifiant 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'*Identifiant 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'*Identifiant de Nœud Etendu* est décrite dans le Tableau 9.

Tableau 9 – Codage binaire de l'Identifiant de Nœud Etendu

Dénomination	Type de données	Description
NodeId	Identifiant de Nœud	Les indicateurs Uri d'Espace de Nom et Indice de Serveur dans le codage de l'Identifiant 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 d'Etat (StatusCode)

Un *Code d'Etat* est codé comme un *UInt32*.

5.2.2.12 Information de diagnostic (DiagnosticInfo)

Une structure d'*Information de Diagnostic* est décrite au 7.8 de la 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 la IEC 62541-4, les champs Identifiant Symbolique, Uri d'Espace de Nom, Texte Localisé et Paramètres de Localisation Géographique (Locale) constituent des indices dans une table de chaînes renvoyée dans l'en-tête de réponse. Seul l'indice de la chaîne correspondante dans la table de chaînes est codé. Un indice de -1 indique qu'il n'existe aucune valeur pour la chaîne.

Tableau 10 – Codage binaire 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 Identifiant Symbolique 0x02 Espace de Nom 0x04 Texte Localisé 0x08 Paramètres de localisation géographique 0x10 Information complémentaire 0x20 Code d'Etat Interne 0x40 Information de Diagnostic Interne
SymbolicId	Int32	Nom symbolique du code d'état.
NamespaceUri	Int32	Espace de nom qui qualifie l'identifiant symbolique.
LocalizedText	Int32	Résumé lisible en clair du code d'état.
Locale	Int32	Localisation géographique utilisée pour le texte localisé.
Additional Info	Chaîne	Information de diagnostic spécifique à l'application détaillée.
InnerStatusCode	Code d'Etat	Code d'état fourni par un système sous-jacent.
InnerDiagnosticInfo	Information de Diagnostic	Information de diagnostic associée au code d'état interne.

5.2.2.13 Nom Qualifié (QualifiedName)

Une structure de *Nom Qualifié* est codée tel qu'indiqué dans le Tableau 11.

La structure abstraite de *Nom Qualifié* est définie au 8.3 de la IEC 62541-3.

Tableau 11 – Codage binaire 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 au 8.5 de la Partie 3.

Tableau 12 – Codage binaire 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 localisation géographique 0x02 Texte
Locale	Chaîne	Paramètres de localisation géographique. Omis signifie nul ou vide.
Text	Chaîne	Texte dans le paramètre de localisation géographique 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'*Identifiant 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'*Elément 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 rétro-recherche 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'identifiant 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 *Elément Xml*, puis décoder le corps de l'objet.

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

Tableau 13 – Codage binaire de l'Objet d'Extension

Dénomination	Type de Données	Description
TypeId	Identifiant de Nœud	Identifiant du nœud de codage de Type de Données dans l'espace d'adresse du serveur. Les *Objets d'Extension* définis par la spécification OPC UA ont un identifiant de nœud numérique qui leur est attribué avec un *Indice d'Espace de Nom* de 0. Les identifiants numériques sont définis en A.1.
Encoding	Octet	Enumération 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 Elément Xml.
Length	Int32	Longueur du corps de l'objet. La longueur doit toujours être spécifiée.
Body	Octet[*]	Le corps de l'objet. Ce champ contient les octets bruts des corps de Chaîne d'Octets. Pour les corps d'Elément 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 binaire 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 Identifiant 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. 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 l'indicateur des dimensions de matrice est établi dans le masque de codage. Les dimensions de rang inférieur apparaissent en premier dans la matrice.

Les types possibles et leurs identifiants pouvant 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 binaire 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 d'Etat 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 d'Etat	L'état associé à la valeur. Absent si le bit de Code d'Etat 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 Enumérations

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 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 de données 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'Identifiant 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'identifiant de type dans la mesure où il est défini de manière explicite dans le Type1.

Tableau 16 – Echantillon de structure codée binaire OPC UA

Champ	Octets	Valeur
Identifiant de Type	4	Identifiant 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 "Identifiant de Type" contient l'identifiant 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 la Partie 4 a un message de requête et de réponse. Les identifiants du Codage de Type de Données attribués à chaque service sont définis en A.1.

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 Partie 2 du schéma XML, (voir 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 Partie 1 du Schéma XML, (voir 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

5.3.1.5 Le type à virgule flottante XML prend en charge la Chaîne "Infinité Positive" (INF[4]), "Infinité négative"(-INF[5]) et "Pas un nombre"(NaN[6]).

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

---

4 INF = *positive infinity*

5 INF = *negative infinity*

6 NaN = *Not a Number*

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 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 doit être codée au format XML suivant: '0001-01-01T00:00:00Z'.

La valeur de date/heure la plus ancienne sur une plate-forme 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, il convient qu'il convertisse la valeur en date&heure la plus récente. 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 est équivalente à une valeur date/heure *Nulle*.

5.3.1.7 Guid (Identifiant 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)*.

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

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

5.3.1.9 Elément 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 Identifiant de Nœud (NodeId)

Une valeur *Identifiant 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'Identifiant 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	Indicateur de spécification du type d'identifiant. L'indicateur a les valeurs suivantes: i NUMERIC (Entier Non Signé) s STRING (Chaîne) g GUID (Identifiant Globalement Unique) b OPAQUE (Chaîne d'Octets)
<value>	*	Identifiant codé comme chaîne. L'identifiant est formaté en utilisant la correspondance de type de données XML propre au type d'identifiant. Noter que l'identifiant peut contenir tout caractère UTF8 non nul, y compris un blanc.

Exemples d'*Identifiants de Nœud:*

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

Le schéma XML applicable à un *Identifiant 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 Identifiant de Nœud Etendu (ExpandedNodeId)

Une valeur *Identifiant 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>
```

Tableau 20 – Composants de l'Identifiant 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 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 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 être omis.
<type>	Enum	Indicateur de spécification du type d'identifiant. Ce champ est décrit dans le Tableau 19 .
<value>	*	Identifiant codé sous forme de chaîne. Ce champ est décrit dans le Tableau 19.

Le schéma XML applicable à *Identifiant 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 d'Etat (StatusCode)

Un *Code d'Etat* 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   d'Etat"   type="tns:Code   d'Etat"
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'*identifiant de Type est l'identifiant 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 Enumérations

Les énumérations utilisées comme paramètres dans les Messages définis dans la 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'Enumération

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 enveloppement 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'*Identifiants 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 requête du service "Lecture" a une structure d'*Identifiant 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

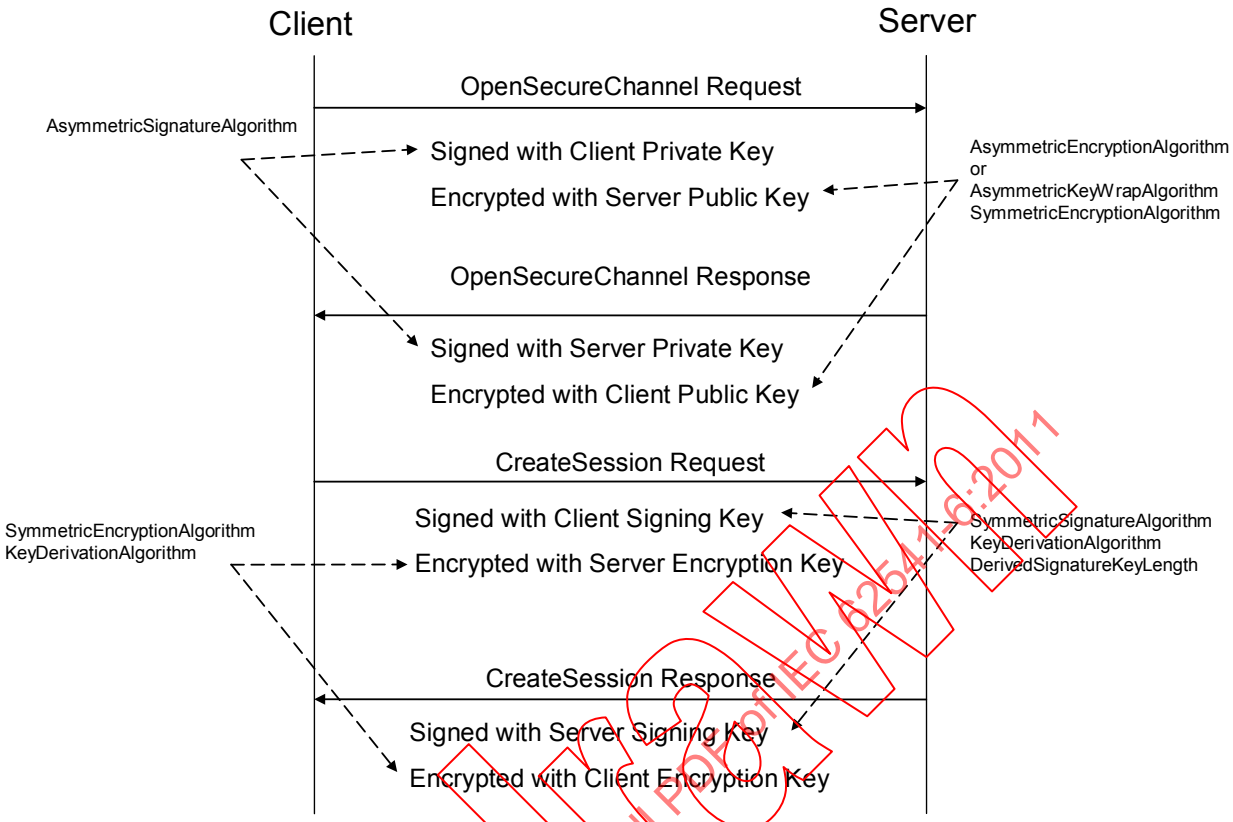
Les messages sont codés comme un *xs:Type Complexe (xs:complexType)*. Les paramètres de chaque message sont sérialisés de la même manière que le sont les champs d'une structure.

6 Protocoles de sécurité

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 la 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 Cryptage (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 identifiant unique pour le *Canal Sécurisé*.



Légende

Anglais	Français
Client	Client
Server	Serveur
AsymmetricSignatureAlgorithm	Algorithme de Signature Asymétrique
OpenSecureChannelRequest	Requête « Ouverture de Canal Sécurisé
Signed with Client Private Key	Signé avec Clé Privée du Client
Encrypted with Server Public Key	Crypté avec Clé Publique du Serveur
OpenSecureChannel Response	Réponse « Ouverture de Canal Sécurisé
AsymmetricEncryptionAlgorithm or	Algorithme de Cryptage Asymétrique ou
AsymmetricKeyWrapAlgorithm	Algorithme d'Enveloppement par clé asymétrique
Symmetric Encryption Algorithm	Algorithme de Cryptage Symétrique
Signed with Server Private Key	Signé avec Clé Privée du Serveur
Encrypted with Client Public Key	Crypté avec Clé Publique du Client
CreateSession Request	Requête "Créer Session"
SymmetricEncryptionAlgorithm	Algorithme de Cryptage Symétrique
KeyDerivationAlgorithm	Algorithme de dérivation des clés
Signed with Client Signing Key	Signé avec la Clé de Signature du Client
Encrypted with Server Encryption Key	Crypté avec la Clé de Cryptage du Serveur
CreateSession Response	Réponse "Créer Session"
Signed with Server Signing Key	Signé avec la Clé de Signature du Serveur
Encrypted with Client Encryption Key	Crypté avec la Clé de Cryptage du Client
SymmetricSignatureAlgorithm	Algorithme de Signature Symétrique
DerivedSignatureKeyLength	Longueur en bits de la clé dérivée utilisée pour l'authentification des messages

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*. La Partie 7 définit les *Politiques de Sécurité* normalisées comme partie intégrante des *Profils* normalisés que les applications OPC UA sont supposées prendre en charge. La Partie 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 cryptage 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 cryptage à clé asymétrique à utiliser.
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 Cryptage Asymétrique* est utilisé pour le cryptage du message complet à l'aide d'une clé asymétrique. Certains *Protocoles de Sécurité* ne cryptent pas le message complet à l'aide d'une clé asymétrique. Ils utilisent en revanche l'A*lgorithme d'Enveloppement à Clé Asymétrique* pour crypter une clé symétrique, puis utilisent l'*Algorithme de Cryptage Symétrique* pour crypter 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 cryptage est déterminée par l'*Algorithme de Cryptage Symétrique*. La longueur des clés permettant de créer des signatures symétriques dépend de l'A*lgorithme de Signature Symétrique* et peut être différente de la longueur des clés de cryptage.

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 asymétriques. Tous les *Protocoles de Sécurité* utilisent des *Certificats* X509 Version 3 (voir ITU-T X.509) codés au format DER (voir ITU-T X.690). Les *Certificats* utilisés par les *Applications* OPC UA doivent également être conformes au RFC 3280 qui définit un profil pour les *Certificat*s 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 pouvant 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 pouvant 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 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.

Tableau 23 – Certificat d'Instance d'Application

Dénomination	Paramètre abstrait	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'instance d'application. L'attribut Nom Courant 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'instance 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 Identifiant 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 la méthode d'utilisation possible de la clé de certificat. Doit inclure la Signature Numérique, la non-Répudiation, le cryptage par clé et le cryptage 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 .

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 Courant 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 identifiant 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 la méthode d'utilisation possible de la clé de certificat. 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'Identifiant d'Objet (OID) ASN.1 pour cette extension est: 1.2.840.113556.1.8000.2264.1.6.1

6.3 Conversation sécurisée WS

6.3.1 Vue d'ensemble

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

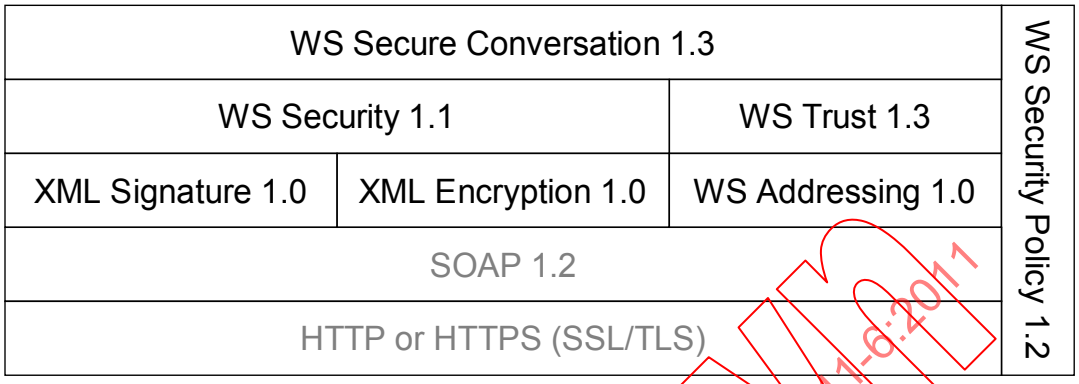
La spécification de signature XML (voir XML Signature) décrit les mécanismes de signature effective des éléments XML. La spécification de cryptage XML décrit les mécanismes de cryptage des éléments XML (voir XML Encryption).

La politique de sécurité WS (voir WS Security Policy) définit les suites algorithmiques normalisées qui permettent de sécuriser les messages SOAP. Ces suites algorithmiques mettent directement en correspondance les Politiques de Sécurité définies dans la Partie 7. Le Profil de sécurité de base WS-I 1.1 (voir WS-I Basic Security Profile Version 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. La CEI 62541-7

peut définir des Profils qui nécessitent une prise en charge pour les versions futures de ces spécifications.

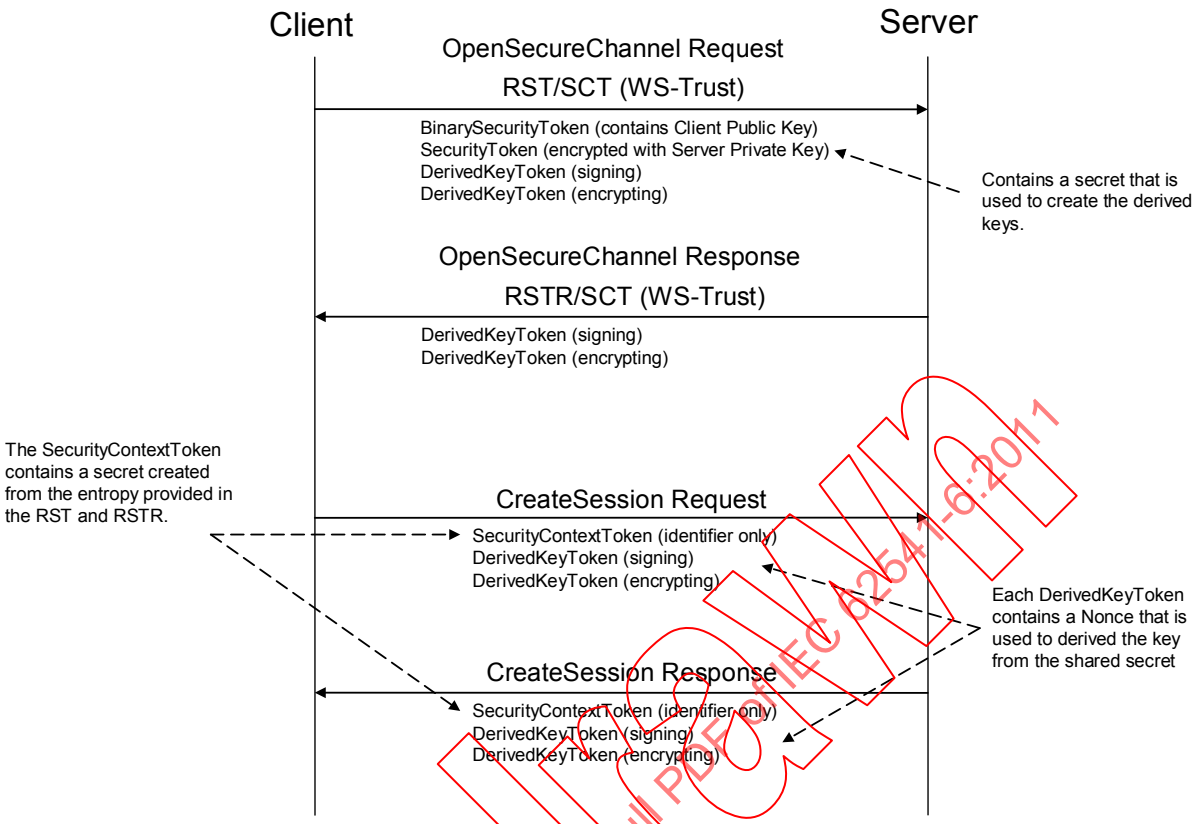


Légende

Anglais	Français
WS Secure Conversation	Conversation Sécurisée WS
WS Security	Sécurité WS
WS Trust	Confiance WS
XML Signature	Signature XML
XML Encryption	Cryptage XML
WS Addressing	Adressage WS
or	Ou
WS Security Policy	Politique de sécurité WS

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.



Légende

Anglais	Français
Client	Client
Server	Serveur
OpenSecureChannel Request	Requête «Ouverture de Canal Sécurisé»
WS Trust	Confiance WS
BinarySecurityToken (contains Client Public Key)	Jeton Binaire de Sécurité (contient la Clé Publique du Client)
SecurityToken (encrypted with Server Private Key)	Jeton de sécurité (crypté avec la Clé Privée du Serveur)
Contains a secret that is used to create the derived keys.	Contient un secret qui permet de créer les clés dérivées.
DerivedKeyToken (signing)	Jeton de Clé Dérivée (signature)
DerivedKeyToken (encrypting)	Jeton de Clé Dérivée (cryptage)
OpenSecureChannel Response	Réponse Ouverture de Canal Sécurisé
The SecurityContextToken contains a secret created from the entropy provided in the RST and RSTR.	Le Jeton de Contexte de Sécurité contient un secret créé à partir de l'entropie des messages RST et RSTR.
CreateSession Request	Requête Créer Session
CreateSession Response	Réponse Créer Session
SecurityContextToken (identifier only)	Jeton de Contexte de Sécurité (identifiant uniquement)
Each DerivedKeyToken contains a Nonce that is used to derive the key from the shared secret	Chaque Jeton de Clé Dérivée contient un Nonce qui permet de dériver la clé à partir du secret partagé

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

Les messages RST (Requête de jeton de sécurité) et RSTR (Réponse à une requête de jeton de sécurité) sont définis par Confiance WS (voir WS Trust). La Conversation sécurisée WS (voir 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 (voir 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

6.3.2 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 25 pour identifier la spécification qui définit un élément XML.

Tableau 25 – 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	Cryptage XML

6.3.3 Requête de jeton de sécurité (RST/SCT)

Le message de requête de jeton de sécurité met en œuvre le message de requête abstrait d'Ouverture de Canal Sécurisé défini dans la IEC 62541-4. La syntaxe de ce message est définie par une Confiance WS (voir WS Trust). La structure du message est décrite en détail dans Conversation Sécurisée WS (voir 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) crypté contenant le Nonce (Nombre à usage unique) du Client utilisé pour dériver les clés. Ce jeton doit être crypté 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 d'adressage WS 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 crypter le corps du message avec l'*Algorithme de Cryptage Symétrique.*

Ce message doit comporter les en-têtes wsa:Action, wsa:Identifiant de message (wsa:MessageId), wsa:Répondre A (wsa:ReplyTo) et wsa:A destination de (wsa:To) définis par Adressage WS. Le message doit également comporter l'en-tête wsu:Horodatage défini par

Sécurité WS. 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 cryptage puis doit être cryptée.

Les correspondances entre les paramètres de requête «Ouverture de Canal Sécurisé» et les éléments du message RST/SCT sont présentées dans le Tableau 26.

Tableau 26 – Correspondance RST/SCT avec une requête Ouverture de Canal Sécurisé

Paramètre d'Ouverture de Canal Sécurisé	Elément ST/SCT	Description
Certificat client	wsse:BinarySecurityToken	Transmis dans l'en-tête SOAP
Type de requête	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.
Identifiant 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 final 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.

6.3.4 Réponse à la Requête de jeton de sécurité (RSTR/SCT)

Le message de réponse à la requête de jeton de sécurité met en œuvre le message abstrait de réponse d'Ouverture de Canal Sécurisé défini dans la IEC 62541-4. La syntaxe de ce message est définie par Confiance WS (voir WS Trust). L'utilisation du message est décrite en détail dans Conversation Sécurisée WS (voir WS Secure Conversation. Ce message n'est pas signé ou crypté avec les algorithmes asymétriques décrits dans la IEC 62541-4. Les algorithmes symétriques et une clé fournie dans le message de requête 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'adressage WS et l'en-tête wsu:Horodatage qui utilisent l'*Algorithme de Signature Symétrique.* Cette clé est dérivée du *Jeton de Sécurité* crypté 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 crypter le corps du message avec l'Algorithme deCryptage Symétrique. Cette clé est dérivée du *Jeton de Sécurité* crypté 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 Adressage WS (voir WS Addressing). Le message doit également comporter l'en-tête wsu:Horodatage défini par Sécurité WS (voir 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 cryptage.

La correspondance entre les paramètres de réponse d'*Ouverture de Canal Sécurisé* et les éléments du message RST/SCT sont présentés dans le Tableau 27.

Tableau 27 – 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 jeton émis.
Jeton de sécurité	wst:RequestedSecurityToken	Spécifie le nouveau SCT (Jeton de contexte de sécurité) ou le SCT renouvelé.
Identifiant de canal	wsc:Identifier	URI absolu d'identification du SCT.
Identifiant de jeton	wsc:Instance	Identifiant d'un ensemble de clés émis pour un contexte donné. Il doit être unique dans le contexte.
créé A	wsu:Created	Elément 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 Cryptées n'est pas utilisé dans OPC UA car le corps du message doit être crypté

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 au 5.5 de la IEC 62541-4.

6.3.5 Utilisation du SCT

Une fois que le client reçoit le message RSTR/SCT, il peut utiliser le SCT pour sécuriser tous les autres messages.

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

Si on utilise un cryptage, 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'adressage WS (voir 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 crypter le corps du message avec l'*Algorithme de Cryptage 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 Adressage WS (voir WS Addressing). Le message doit également comporter l'en-tête wsu:Horodatage défini par Sécurité WS (voir WS Security) .

6.3.6 Annulation des contextes de sécurité

Le message d'Annulation défini par Confiance WS met en œuvre le message de requête abstrait de Fermeture de Canal Sécurisé défini dans la IEC 62541-4.

Ce message doit être sécurisé avec l'élément SCT.

6.4 Conversation OPC UA sécurisée

6.4.1 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.

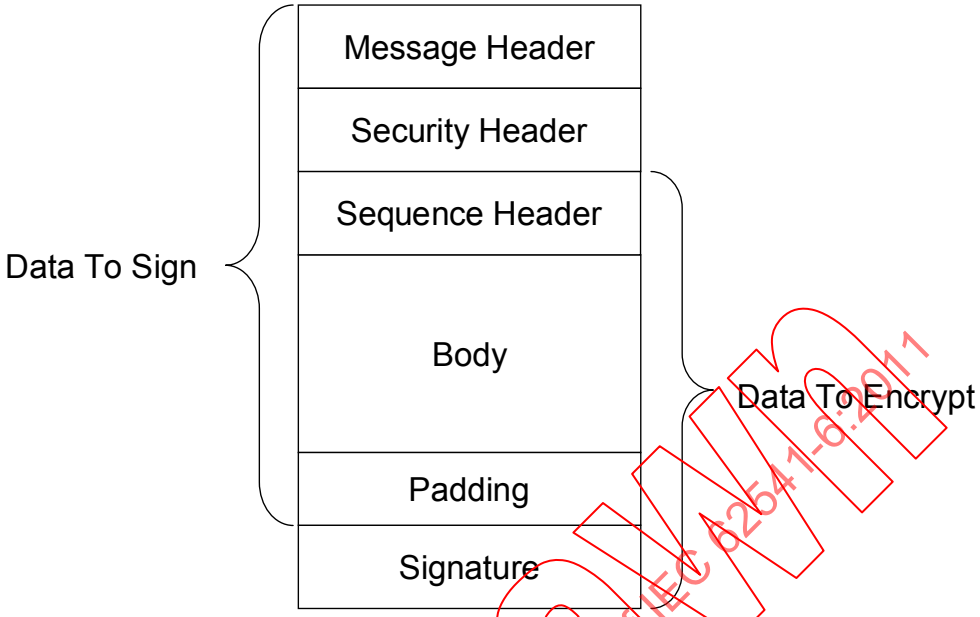
Une sécurité totale est appliquée aux *Blocs de Messages* individuels et non au message OPC UA complet. Une *Pile* qui met en œuvre une conversation UASC est chargée de vérifier la sécurité de chaque *Bloc de Messages* reçu et de reconstituer le message OPC UA d'origine.

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.

6.4.2 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.



Légende

Anglais	Français
Data To Sign	Données à Signer
Message Header	En-tête de Message
Security Header	En-tête de Sécurité
Sequence Header	En-tête de Séquence
Data To Encrypt	Données à crypter
Body	Corps
Padding	Remplissage
Signature	Signature

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 28.

Tableau 28 – 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	Identifiant unique du Canal Sécurisé attribué par le serveur. Si un Serveur reçoit un Identifiant de Canal Sécurisé qu'il ne reconnaît pas, il doit renvoyer une erreur de couche de transport appropriée.

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 29. Les algorithmes asymétriques permettent de sécuriser les messages *Ouverture de Canal Sécurisé*. PKCS #1 définit un ensemble d'algorithmes asymétriques que 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 29 – 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'Emetteur (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 ITU-T X.509. Le format DER pour un certificat est défini dans la norme .ITU-T X.690 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'Emetteur* 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.
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 crypter le Bloc de Messages. Ce champ doit être nul si le message n'est pas crypté.

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'Emetteur* 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'Emetteur* peut être calculée à l'aide de la formule suivante:

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

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'Emetteur.* 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 30. 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 cryptage 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 30 – En-tête de sécurité d'algorithme symétrique

Dénomination	Type de données	Description
TokenId	UInt32	Identifiant unique du jeton de *Canal Sécurisé* utilisé pour sécuriser le message. Cet identifiant est renvoyé par le serveur dans un message de réponse d'Ouverture de Canal Sécurisé. Si un Serveur reçoit un Identifiant 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 définie dans le Tableau 31. L'en-tête de séquence garantit que le premier bloc crypté de chaque message transmis sur un canal commence avec des données différentes.

Tableau 31 – 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	Identifiant attribué par le client au Message de requête OPC UA. Tous les *Blocs de Messages* pour la requête et la réponse associée utilisent le même identifiant.

Les *Numéros de séquence* peuvent ne pas être réutilisés pour un *Identifiant de jeton*. Il convient que la durée de vie utile d'un jeton 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 *Identifiant 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 requête 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 32.

Tableau 32 – 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 crypter est un entier multiple de la taille de bloc de cryptage. La valeur de chaque octet du remplissage est égale à la Taille de Remplissage.
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 devant être transmises (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 cryptage 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 Crypté.* Ces valeurs dépendent de l'algorithme de cryptage et peuvent être identiques.

Le message 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 Message* n'est pas crypté.

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