[MC-NMF]:
.NET Message Framing Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
08/10/2007	0.1	Major	Initial Availability
09/28/2007	0.2	Minor	Updated the technical content.
10/23/2007	0.3	Minor	Updated the technical content.
11/30/2007	0.3.1	Editorial	Revised and edited the technical content.
01/25/2008	0.3.2	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies the .NET Message Framing Protocol (NMF), which defines a mechanism for framing messages. While this is primarily aimed at framing SOAP messages, the protocol can be used to frame other message types as well. The protocol can run over any transport, including those that don't natively support message semantics, and provide support for sending and receiving demarcated messages.

Familiarity with SOAP and XML technologies are required for a complete understanding of this document.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Little Endian
Unicode

The following terms are specific to this document:

Endpoint: A node that sends or receives a Protocol Stream.

Initiating Stream: The Protocol Stream that flows from the Initiator.

Initiator: The node that initiates the connection over which a Protocol Stream flows.

Envelope Record: A Record that contains data, such as a SOAP message [SOAP1.1], [SOAP1.2-1].

NMF: An abbreviation for .NET Message Framing Protocol.

Property Record: A Record that contains a Protocol Stream Property.

Protocol Stream: A continuous stream of Records flowing in one direction.

Protocol Stream Property: A Protocol Stream characteristic that may be set by a Property Record and applies to subsequent Records flowing with the Protocol Stream.

Receiver: The node that is the receiver of the Protocol Stream.

Record: A sequence of octets.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as specified in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[MC-NBFS] Microsoft Corporation, ".NET Binary Format: SOAP Data Structure", July 2007.

[MC-NBFSE] Microsoft Corporation, ".NET Binary Format: SOAP Extension", July 2007.

[MS-DTYP] Microsoft Corporation, "Windows Data Types", January 2007.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[RFC2045] Freed, N., et al., "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996, http://ietf.org/rfc/rfc2045.txt

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC2234] Crocker, D. and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997, http://www.ietf.org/rfc/rfc2234.txt

[RFC2279] Yergeau, F., "UTF-8, A Transformation Format of ISO10646", RFC 2279, January 1998, http://www.ietf.org/rfc/rfc2279.txt

[RFC2396] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998, http://www.ietf.org/rfc/rfc2396.txt

[RFC2781] Hoffman, P. and Yergeau, F., "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000, http://www.ietf.org/rfc/rfc2781.txt

[RFC4178] Zhu, L., Leach, P., Jaganathan, K., and Ingersoll, W., "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism", RFC 4178, October 2005, http://www.ietf.org/rfc/rfc4178.txt

[RFC4346] Dierks, T. and Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006, http://www.ietf.org/rfc/rfc4346.txt

[SOAP1.1] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D., "Simple Object Access Protocol (SOAP) 1.1", May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[SOAP1-2.1] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. J., Nielsen, H. F., Karmarkar, A. and Lafon, Y., " SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) ", W3C Recommendation 27, April 2007, http://www.w3.org/TR/soap12-part1

[SOAP-MTOM] Gudgin, M., et al, "SOAP Message Transmission Optimization Mechanism", W3C Recommendation, 25 January 2005, http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/

1.2.2 Informative References

[MS-MQMA] Microsoft Corporation, "Message Queuing (MSMQ): Architecture Protocol Specification", August 2007.

[MSDN-NETTcp] Microsoft Corporation, "NetTcpBinding Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.nettcpbinding.aspx

[MSDN-NETNamedPipe] Microsoft Corporation, "NetNamedPipeBinding Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.netnamedpipebinding.aspx

[MSDN-NETMsmq] Microsoft Corporation, "NetMsmqBinding Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.netmsmqbinding.aspx

[MSDN-NETTcpBE] Microsoft Corporation, "TcpTransportBindingElement Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.channels.tcptransportbindingelement.aspx

[MSDN-NETNamedPipeBE] Microsoft Corporation, "NamedPipeTransportBindingElement Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.channels.namedpipetransportbindingelement.aspx

[MSDN-NETMsmqBE] Microsoft Corporation, "MsmqTransportBindingElement Class", http://msdn2.microsoft.com/en-us/library/system.servicemodel.channels.msmqtransportbindingelement.aspx

[MSDN-WCF] Microsoft Corporation, "Windows Communication Foundation", http://msdn2.microsoft.com/en-us/library/ms735119.aspx

1.3 Protocol Overview (Synopsis)

The .NET Message Framing Protocol defines a format for framing messages, including SOAP messages. Consider a scenario in which two SOAP nodes are interacting and exchanging SOAP messages. The transport used for communication may not inherently support the notion of messages. For example, if the underlying transport is TCP, it will provide a byte stream, and the **receiver** will need to have additional parsing logic to be able to extract a SOAP message from this stream.

This protocol intends to meet the following requirements:

- Supports extensibility for different message encoding formats.
- Provides delimiters for a message.
- Capability to skip past a message that is not well formed. If the message frames are well formed but the embedded content is malformed, the protocol provides a means of skipping over all such message frames.
- Support extensible upgrades of the underlying transport stream.

The basic idea is to first notify the recipient of the message properties (metadata)—what version of the framing protocol is being used, who the message is meant for, what encoding algorithm is used to encode the message content—and then send a number of message frames conforming to those properties. The recipient, based on the message properties, would be able to extract the messages from the transport stream and deliver them to the appropriate **endpoint**.

The message properties are typically controlled by the protocol configuration object (PCO).

The PCO determines the following aspects of a given instance of the protocol.

- Transport to be used.
- Version of the .NET Message Framing Protocol being used.
- Mode of communication, explained in sections 1.3.2 and 2.2.3.2.
- Via, a URI identifying the endpoint the messages are intended for.
- Encoding format being used for the messages. The different encoding schemes are covered in section 2.2.3.4.
- Chunk size. If the mode supports chunking, this determines the maximum size of a chunk.

- Implementation-defined maximum supported sizes for messages and **record** types.

How the Property Configuration Object is configured and managed is not addressed in this document.<1>

1.3.1 Scenarios

In this section, some scenarios are described that capture the various message exchange patterns between SOAP nodes. This will help one to define the communication modes (covered in the next section) that the protocol needs to support.

The case of a sales organization is used in which there are a number of salespersons: some in the head office, some offsite. They are interacting with the customers and preparing purchase orders that need to be sent to a central server as SOAP messages. The purchase orders can also be retrieved from the server, again as SOAP messages. The Asynchronous Message Relay is a mechanism that will be used to queue up messages when the salesperson is offline, and then relay the messages once connectivity is established. Description of such a mechanism is not addressed in this document. One such mechanism is Microsoft Message Queuing, as specified in [MS-MQMA].

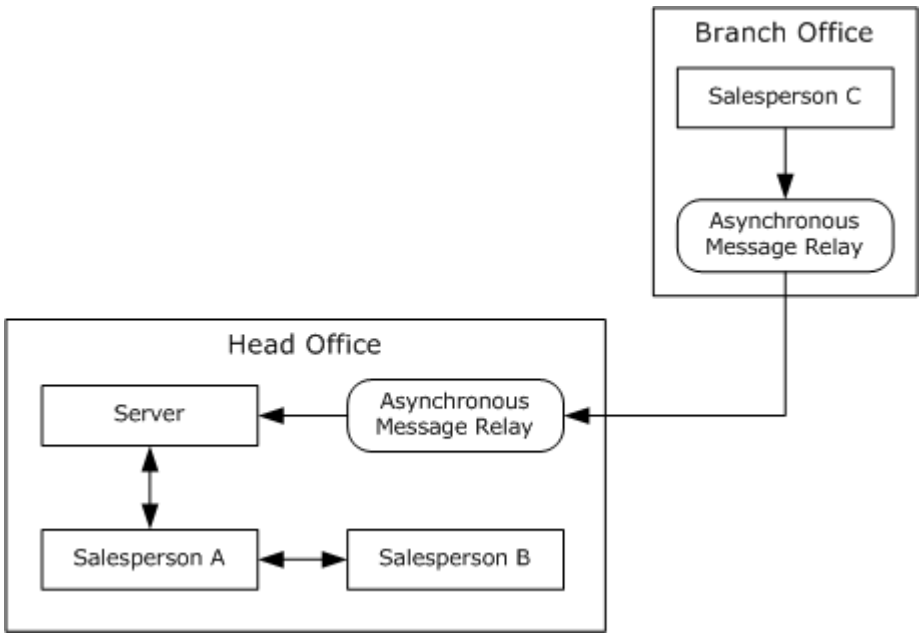


Figure 1: An Asynchronous Message Relay

1.3.1.1 Multiple Bidirectional Message Exchange Scenario

In this scenario, there are two salespersons working at the head office with a number of customers. Salesperson A is responsible for collecting the customer profiles, and salesperson B is responsible for collecting the customer requirements. The two pieces of information will need to be combined to create a purchase order. Also, the head office has a high volume of customers, and so there will be frequent message exchanges between the two salespersons.

For this scenario, it makes sense to first establish a session, initiated by say A, where the message properties would be sent out. Subsequently, the messages frames can be sent from A or B and the other end would be able to extract the message using the message properties for that session. At the end of the conversation, either of A or B can terminate the session.

1.3.1.2 Large Message Exchange Scenario

In this scenario, there is a salesperson who retrieves the entire customer inventory (in the form of a message) from the server at the start of the day.

Because this is not a very frequent operation, a session is not wanted, as was the case in the previous scenario. Instead, the protocol should send the message properties followed by the message frames, and the receiving end would apply the properties to extract the message.

In addition, because the inventory is quite big in size, the message content may need to be broken down into multiple chunks so the receiving end can stream it one chunk at a time and does not have to process the entire message at once.

1.3.1.3 Offline Message Exchange Scenario

In this scenario, there is a salesperson C who is visiting various customers and creating their purchase orders. However, the salesperson does not have access to the server, and only after he returns to his branch office he will be able to upload these to the server. The order application would use some mechanism (for example, Microsoft Message Queuing) to store these messages locally, and that mechanism will take care of relaying the message to the server when the salesperson is online.

This differs from the scenario in section 1.3.1.2 in the sense that the receiving end of protocol (that is, the relay) cannot actively participate in the protocol. This is a "store and forward" scenario in which the sending end of the protocol stores the message frame in some intermediate store, and, later, the message frame is forwarded to/retrieved by the receiving end, which then extracts the message from the message frame.

Depending on the scenario characteristics, it may be preferable to send the message properties on a per-message basis, or once in advance for a number of messages. The latter case uses the same session semantic as before except that session establishment only involves participation from one end.

1.3.2 Communication Modes

Based on the above scenarios, the messages exchange between nodes can be classified along the following four criterions.

1.3.2.1 Message Property Scope

Message properties can be sent on a per-message basis or once for a session, which spans multiple messages. If there are many messages with identical properties being sent, it is optimal to use the per-session scope.

1.3.2.2 Protocol Receiver Mode

The receiving end could be actively participating in the protocol, or it could be a passive relay entity. If the receiving end is active, it can negotiate some capabilities such as protocol upgrade.

1.3.2.3 Message Traffic Flow

The logical flow of messages could be unidirectional where only one end sends messages or it could be bidirectional, where both ends send messages. Note that for unidirectional messages, the receiver may need to acknowledge message receipt but the logical message flow is still in one direction.

1.3.2.4 Message Chunking

The entire message could be sent in one message frame or it could be split across multiple chunks. The chunking would be extremely useful when dealing with large messages.

Using these criteria, four communication modes will be specified for the protocol to operate in. These modes determine the pattern of messages exchanged between the nodes and determine when the message properties are exchanged and how the message frames are created.

Mode name	Message property scope	Protocol receiver mode	Traffic flow	Message chunking
Singleton Unsized	Single	Active	Unidirectional	Yes
Duplex	Multiple	Active	Bidirectional	No
Simplex	Multiple	Passive	Unidirectional	No
Singleton Sized	Single	Passive	Unidirectional	No

1.3.3 Protocol Upgrades

The .NET Message Framing Protocol provides the capability to upgrade the underlying **protocol stream** to a complementary protocol (say SSL/TLS). If the other end supports this protocol and goes through with the upgrade, the subsequent byte stream (messages included) will use this upgraded protocol.

The upgrade request is sent as part of the message properties. How the upgrade actually happens is specific to the other protocol and is not addressed in this document. Multiple upgrade negotiations can be performed. In addition, because this is a negotiation, it requires participation from both ends, and, hence, is available only when the communication mode is Singleton, Unsized, or Duplex.

1.4 Relationship to Other Protocols

This protocol is available for use over any network transport that needs to provide message send and receive semantics. Transports that fall in this category include TCP and Named Pipes.

1.5 Prerequisites/Preconditions

The protocol assumes that a transport session has been established. The management of the transport session (that is, how and when it is established, management of idle sessions, and closure of the transport session) is not a responsibility of the protocol. The protocol only uses the transport session to send and receive octets.

In the case of Singleton Sized mode, described in section 1.3.2, the size of the message is not contained as part of the message frame. The protocol assumes that the underlying transport has a means of computing the size and relaying it to the protocol.

1.6 Applicability Statement

This protocol is applicable for implementation by a transport module that wants to provide message demarcation to higher-layer applications. Higher-level applications can use this module to send and receive messages.

Applicable scenarios include when the communicating nodes are connected (for example, employees in the head office) or when they are disconnected (for example, an employee working remotely).

Applicable scenarios include when the communicating nodes are exchanging large messages and message level streaming is required to optimize the use of resources (memory, processing, and so on).

Applicable scenarios include when the communicating nodes want to upgrade the underlying transport to a complementary protocol and exchange messages using the complementary protocol.

Applicable scenarios include when a receiving node wants to skip past embedded messages that are not well formed and process subsequent messages that are well-formed.

The protocol is not applicable for scenarios in which applications don't need message level access, or the underlying transport's native message format is sufficient.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- **Protocol Versions:** This document describes version 1.0 of the .NET Message Framing Protocol. The version information is part of the protocol exchange as covered in section 2.2.3.1.
- **Capability Negotiation:** The .NET Message Framing Protocol does not support negotiation of the version, mode, upgrades, and message encoding. Instead, an implementation must be configured with these, as described in section 3.1.3.

1.8 Vendor-Extensible Fields

This protocol allows extensibility for the following fields:

Extensible Encoding. An implementation can opt for an extensible encoding. Vendors need to specify the encoding as specified in [RFC2045] and covered in detail in section 2.2.3.4.2

Upgrades. Vendors can define new protocol upgrades in addition to the ones specified in section 2.2.3.5

Faults. An implementation can define new faults in addition to the ones specified in section 2.2.5. The fault is a URI as defined in [RFC2396] encoding using UTF-8 encoding as specified in [RFC2279]. Vendors define a URI namespace for their faults that is different from the http://schemas.microsoft.com/ws/2006/05/framing/faults/ namespace used by the faults in this protocol.

1.9 Standards Assignments

There are no standards assignments for this protocol.

2 Messages

2.1 Transport

This protocol is available for use over any network transport that needs to provide message send and receive semantics. Transports that fall in this category include TCP and Named Pipes.

2.2 Message Syntax

2.2.1 Record Types

This protocol involves the exchange of a number of records. Records can be categorized as either **Property Records** or **Envelope Records** based on their contents. The Property Records contain message properties. The Envelope Records contain the message payload. These records and their structure are covered in detail in subsequent sections. Each of these records is prefixed with a Record Type, which is an octet, and MUST be set to one of the values specified below. Values of 0x0D – 0xFF for this octet are reserved for future use.

Value	Record type
0x00	Version Record
0x01	Mode Record
0x02	Via Record
0x03	Known Encoding Record
0x04	Extensible Encoding Record
0x05	Unsized Envelope Record
0x06	Sized Envelope Record
0x07	End Record
0x08	Fault Record
0x09	Upgrade Request Record
0x0A	Upgrade Response Record
0x0B	Preamble Ack Record
0x0C	Preamble End Record

2.2.2 Record Size Encoding

For variable-sized records used by this protocol, the record needs to contain the size of the content (in octets). An implementation SHOULD support record sizes as large as 0xffffffff octets (encoded size requires five octets).

The encoding algorithm takes the Size of the record payload as input in **little endian** format and generates a stream of octets. The octets MUST be sent in the order in which they are generated.

```
While (Size is not zero)
{
    Take the last 7 bits from Size
    If (Size, after extraction of 7 bits, is not zero)
    {
        Next octet in encoded size = 1 followed by the extracted 7 bits
    }
    Else
    {
        Next octet in encoded size = 0 followed by the extracted 7 bits
    }
}
```

Figure 2: The Encoding Algorithm

The table below lists the encoded sizes for the range of values of Size, computed as explained above. The network ordering of octets is top-down. For example, if the size is in the range 0x80 – 0x3FFF, the network ordering of encoded size octets would be (Size & 0x7F) | 0x80 followed by Size >> 0x07.

Integer value (size)	Encoding
0x00 – 0x7F	Size
0x80 – 0x3FFF	(Size & 0x7F)| 0x80 Size >> 0x07
0x4000 – 0x1FFFFF	(Size & 0x7F)| 0x80 ((Size >> 0x07) & 0x7F)| 0x80 Size >> 0x0E
0x200000 – 0x0FFFFFFF	(Size & 0x7F)| 0x80 ((Size >> 0x07) & 0x7F)| 0x80 ((Size >> 0x0E) & 0x7F)| 0x80 Size >> 0x15
0x10000000 – 0x0FFFFFFFF	(Size & 0x7F)| 0x80 ((Size >> 0x07) & 0x7F)| 0x80 ((Size >> 0x0E) & 0x7F)| 0x80 ((Size >> 0x15) & 0x7F)| 0x80 Size >> 0x1C

In the table above, "&" refers to a bit-wise "and" operation, "|" refers to a bit-wise "or" operation, and ">>" refers to a right-shift operation.

2.2.3 Property Records

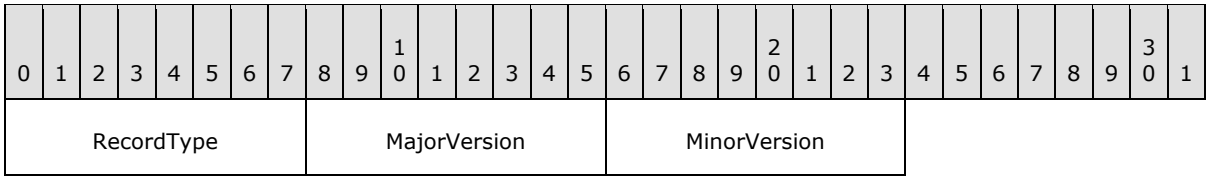
The Property Records contain metadata about the Protocol Stream. When Property Records are received, they set a property of the Protocol Stream and affect the interpretation of the subsequent records within the Protocol Stream.

2.2.3.1 Version Record

The Version Record is a Property Record used to indicate the version of the .NET Message Framing Protocol being used. The Version Record enables later versions of this specification to define additional Record types and associated semantics.

The data portion of a Version Record is a pair of octets indicating the major and minor version numbers. New sets of values for existing record types (for example, additional values of the Known Encoding Type Record) MUST be indicated with a different Minor Version value. All other types of changes MUST be indicated with a different Major Version value.

The major and minor values of the Version Record denote the version of the framing format, not that of the payload Envelope.



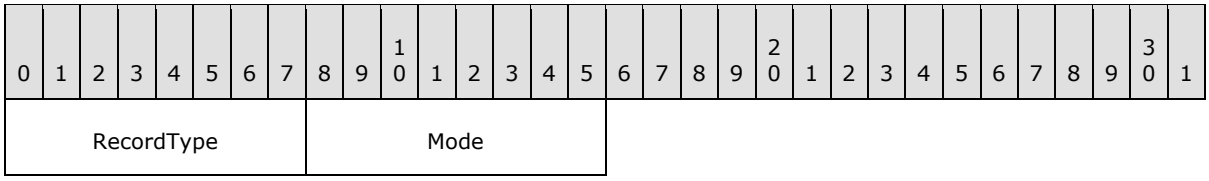
RecordType (1 byte): This octet MUST be set to 0x00 to indicate this is a Version Record.

MajorVersion (1 byte): Specifies the major version of the .NET Message Framing Protocol. An implementation confirming to this specification MUST set this to 0x01. A value of 0x00 is invalid for this octet, and values of 0x02 – 0xff are reserved for future use.

MinorVersion (1 byte): Specifies the minor version of the .NET Message Framing Protocol. An implementation conforming to this specification MUST set this to 0x00. The values 0x01 – 0xff for this octet are reserved for future use.

2.2.3.2 Mode Record

The Mode Record is a Property Record that defines the communication mode for the session. The data portion of a Mode Record is a single octet.



RecordType (1 byte): This octet MUST be set to 0x01 to indicate this is a Mode record.

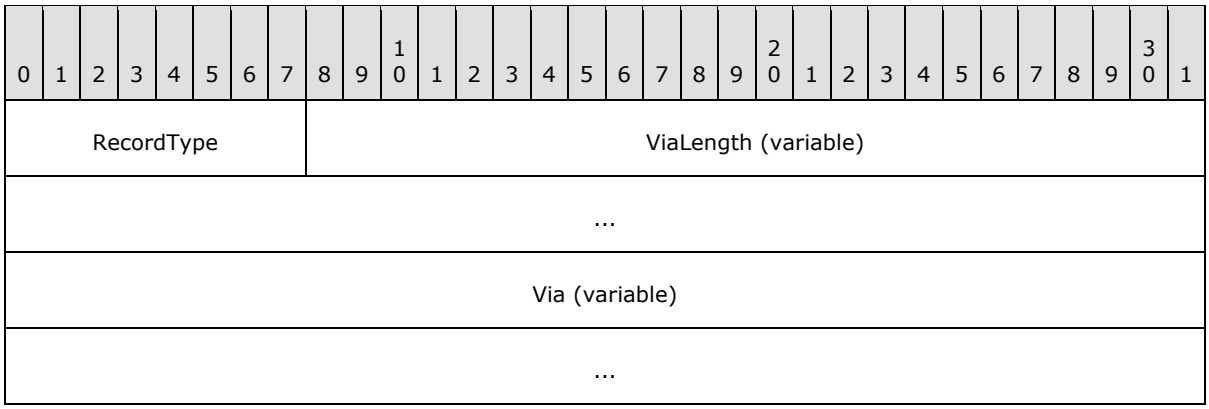
Mode (1 byte): The mode value MUST be set to one of the below values. A value of 0x00 is invalid for this octet, and values of 0x05 – 0xff are reserved for future use.

Short Name	Meaning
Singleton-Unsized 0x01	**Initiating Stream** for a single, one-way message or for a pair of messages in a request-reply manner between two nodes.
Duplex 0x02	Initiating Stream for multiple, bi-directional messages between two nodes.

Short Name	Meaning
Simplex 0x03	Initiating Stream for multiple, one-way messages from a single source.
Singleton-Sized 0x04	Initiating Stream for a single, one-way message from a single source.

2.2.3.3 Via Record

The Via Record is a Property Record that defines the URI for which subsequent messages are bound. The data portion of a Via Record is of variable length.



RecordType (1 byte): This octet MUST be set to 0x02 to indicate this is a Via record.

ViaLength (variable): The value MUST be set to the size of the Via in octets, encoded based on the scheme defined in section 2.2.2. The length MUST NOT be set to 0.

Via (variable): A URI (as defined in [RFC2396] except that the 'escaped' construct is never used). The URI MUST be encoded using UTF-8, as specified in [RFC2279].

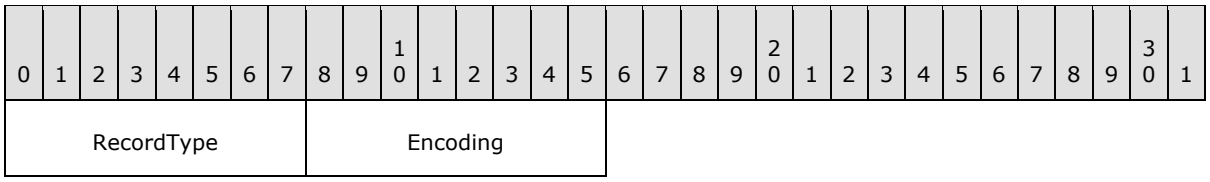
2.2.3.4 Envelope Encoding Record

Envelope Encoding Records are the Property Records that define the encoding format used to encode the message envelope within subsequent Envelope Records. Such records come in two forms: Known Encoding Records and Extensible Encoding Records.

When this record is depicted in messages, we portray it as a variable-sized record to capture the fact that it could be either of the two forms. If it is using Known Encoding, it will be fixed-size; else it will be variable-sized.

2.2.3.4.1 Known Encoding Record

The Known Encoding Record indicates a previously known encoding for the subsequent Envelope Records. The data portion of this record is a single octet.



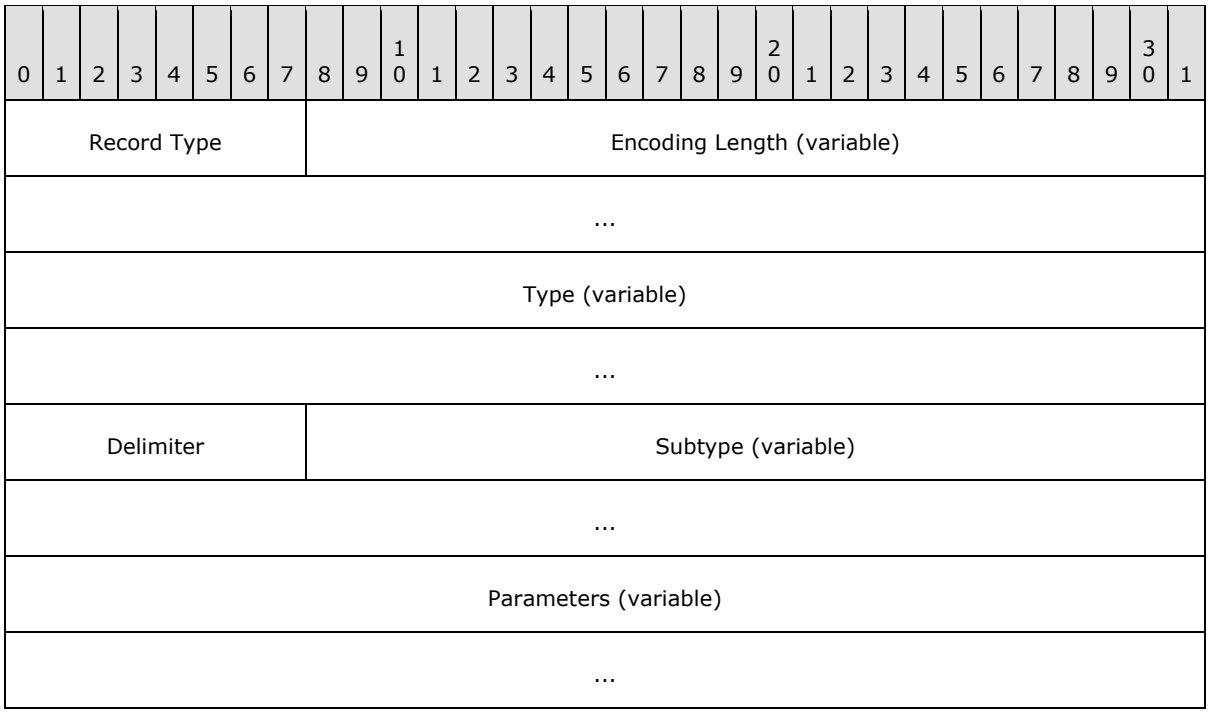
RecordType (1 byte): This octet MUST be set to 0x03 to indicate this a Known Encoding Record.

Encoding (1 byte): This octet MUST be set to one of the values below. Values of 0x09 – 0xFF are reserved for future use.<2>

SOAP version	Meaning
1.1 0x00	UTF-8, as specified in [RFC2279].
1.1 0x01	UTF-16, as specified in [RFC2781].
1.1 0x02	**Unicode** little endian.
1.2 0x03	UTF-8.
1.2 0x04	UTF-16.
1.2 0x05	Unicode little endian.
1.2 0x06	MTOM, [MTOM].
1.2 0x07	Binary, as specified in [MC-NBFS].
1.2 0x08	Binary with in-band dictionary, as specified in [MC-NBFSE].

2.2.3.4.2 Extensible Encoding Record

The Extensible Encoding Record indicates an ad hoc encoding for subsequent Envelope Records. The record data in this case is a Multipurpose Internet Mail Extensions (MIME) [RFC2045] content type encoded using UTF-8 encoding.



Record Type (1 byte): This octet MUST be set to 0x04 to indicate this is an Extensible Encoding Record.

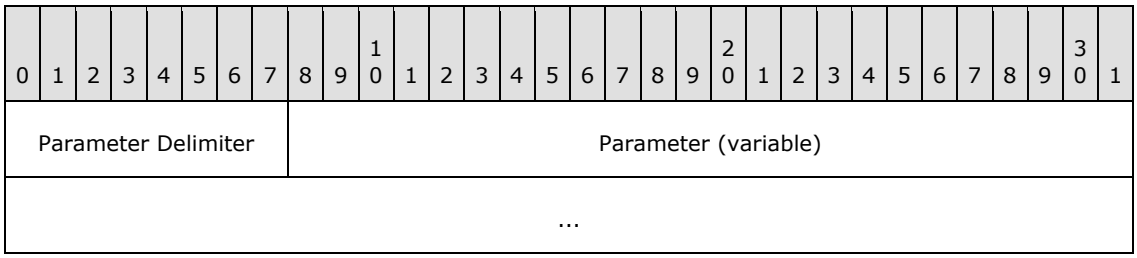
Encoding Length (variable): The value MUST be set to the size of the payload in octets, encoded based on the scheme described in section 2.2.2. The length MUST NOT be set to 0.

Type (variable): This MUST be set to a type specified in [RFC2045] section 5.1.

Delimiter (1 byte): This MUST be set to the octet 0x2F (UTF-8 encoding for "/").

Subtype (variable): This MUST be set to a subtype specified in [RFC2045] section 5.1.

Parameters (variable): There can be one or more parameters in which the parameter structure is defined as follows.

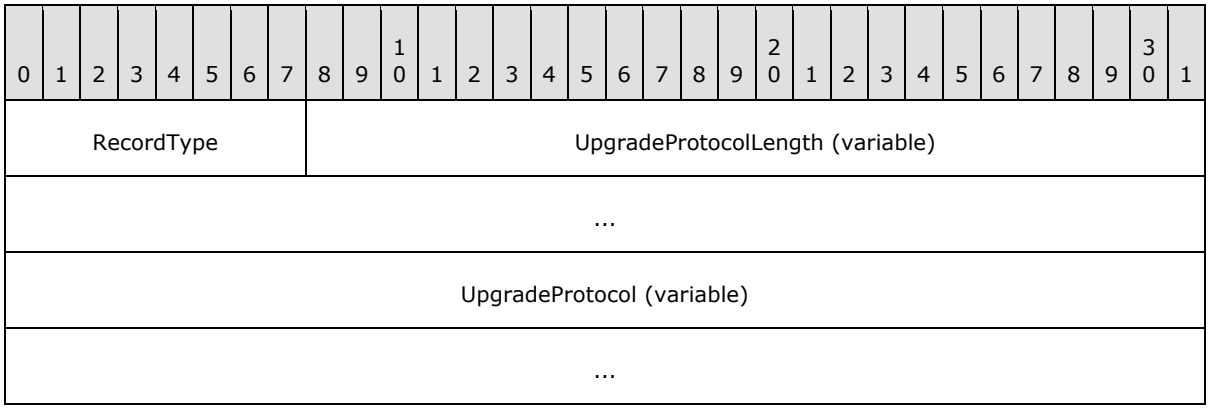


Parameter Delimiter (1 byte): This MUST be set to the octet 0x3B (UTF-8 encoding for ";").

Parameter (variable): This MUST be set to a parameter, as specified in [RFC2045] section 5.1.

2.2.3.5 Upgrade Request Record

The Upgrade Request Record is a Property Record requesting a protocol upgrade.



RecordType (1 byte): This octet MUST be set to 0x09 to indicate this is an Upgrade Request Record.

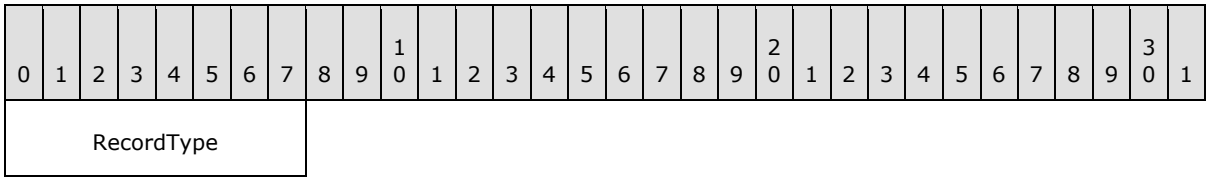
UpgradeProtocolLength (variable): The value MUST be set to the size of the Upgrade Protocol name in octets, encoded based on the scheme described in section 2.2.2. The length field MUST NOT be set to 0.

UpgradeProtocol (variable): Name of the protocol to upgrade to, encoded using UTF-8. The below table identifies some known upgrade protocol names. An implementation SHOULD implement these upgrades, and MAY define additional upgrade protocol definitions.<3>

Protocol	Meaning
SSL/TLS application/ssl-tls	As defined in [RFC4346].
Negotiate application/negotiate	As defined in [RFC4178].

2.2.3.6 Upgrade Response Record

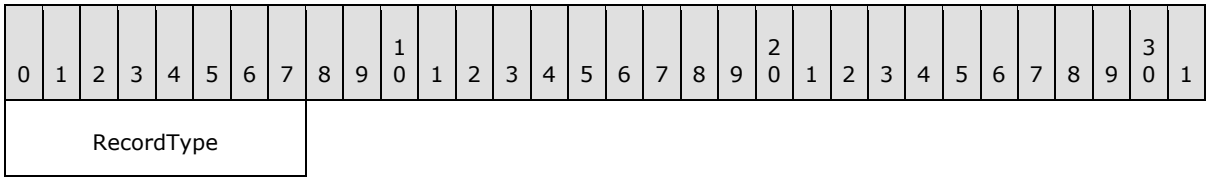
The Upgrade Response Record is a Property Record sent in response to an Upgrade Request Record indicating a willingness to upgrade the protocol stream. This record has no data.



RecordType (1 byte): This octet MUST be set to 0x0A to indicate this is an Upgrade Response Record.

2.2.3.7 Preamble End Record

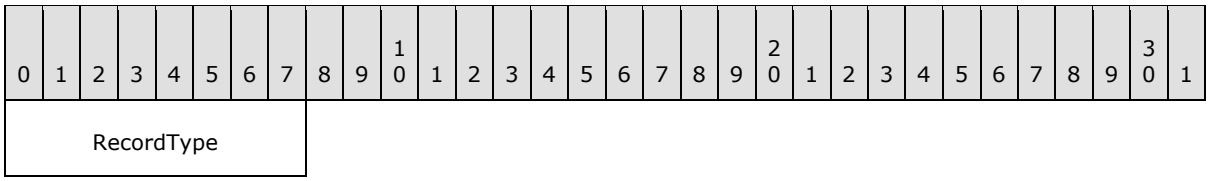
The Preamble End Record is a Property Record sent to indicate the end of message properties. Envelope Records follows this record. This record has no data.



RecordType (1 byte): This octet MUST be set to 0x0C to indicate this is a Preamble End Record.

2.2.3.8 Preamble Ack Record

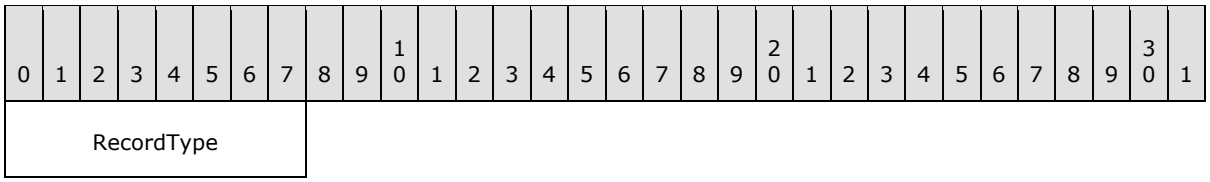
The Preamble Ack Record is a Property Record sent to indicate receipt of a Preamble End Record and that all message properties and stream upgrades have been successfully applied. The receiving end is now ready to receive the Envelope Records. This record has no data.



RecordType (1 byte): This octet MUST be set to 0x0B to indicate this is a Preamble Ack Record.

2.2.3.9 End Record

The End Record is a Property Record that indicates that communication over a connection has ended. This record has no data.



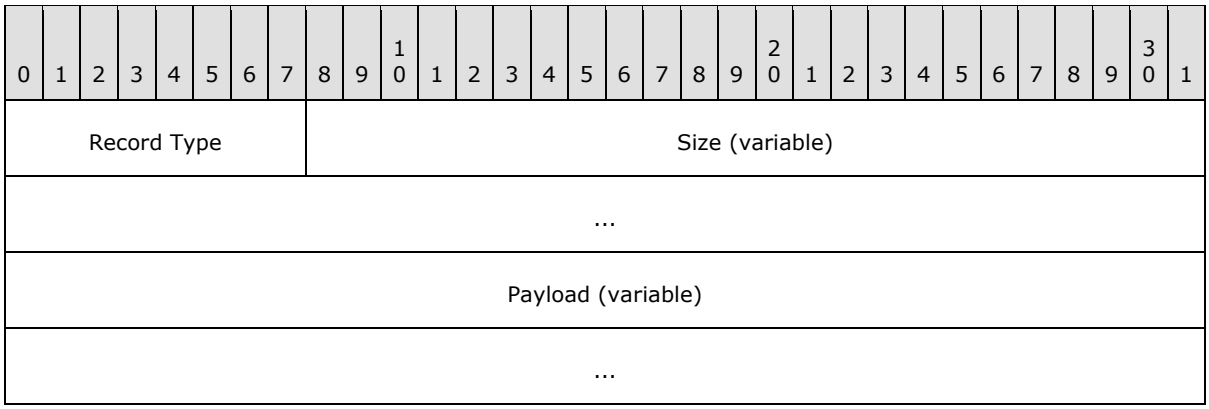
RecordType (1 byte): This octet MUST be set to 0x07 to indicate this is an End Record.

2.2.4 Envelope Records

An Envelope Record contains a message payload. There are two possible record types, depending on the message transfer mode.

2.2.4.1 Sized Envelope Record

A Sized Envelope Record contains a message of the specified size.



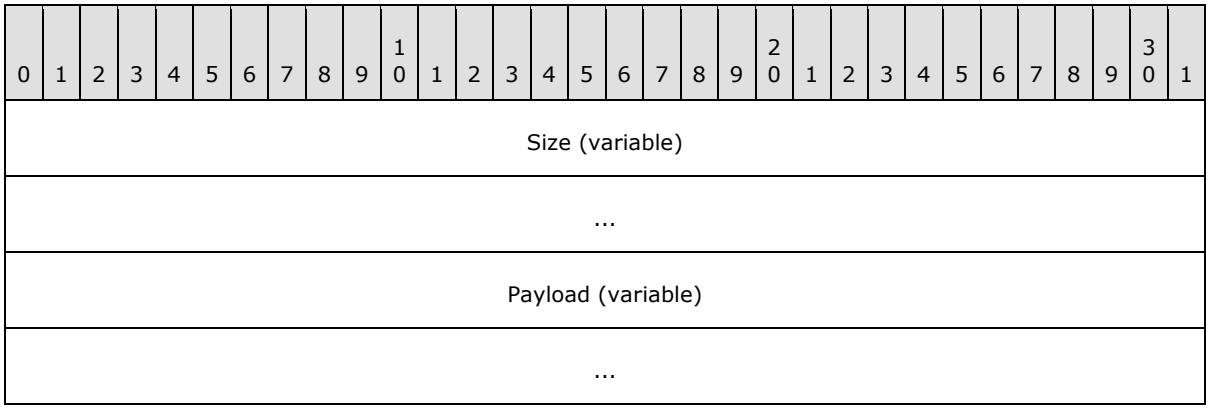
Record Type (1 byte): This octet MUST be set to 0x06 to indicate this is a Sized Envelope Record.

Size (variable): The value MUST be set to the size of the payload in octets, encoded based on the scheme described in section 2.2.2. The size MUST NOT be set to 0.

Payload (variable): Content of the message.

2.2.4.2 Data Chunk

A Data Chunk is used to transmit a portion of a message payload.

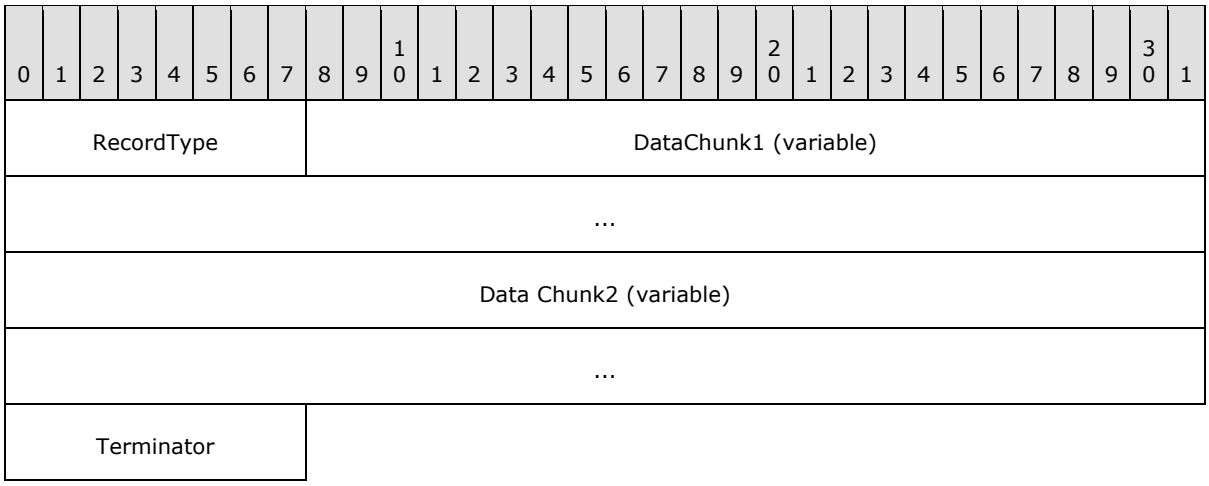


Size (variable): The value MUST be set to the size of the payload in octets, encoded based on the scheme described in section 2.2.2. The size MUST NOT be set to 0.

Payload (variable): Content of the chunk.

2.2.4.3 Unsized Envelope Record

An Unsized Envelope Record contains a message broken into one or more Data Chunk(s). The end of this record is indicated by a single 0x00 octet in place of the start of the next Data Chunk.



RecordType (1 byte): This octet MUST be set to 0x05 to indicate this is an Unsized Envelope Record.

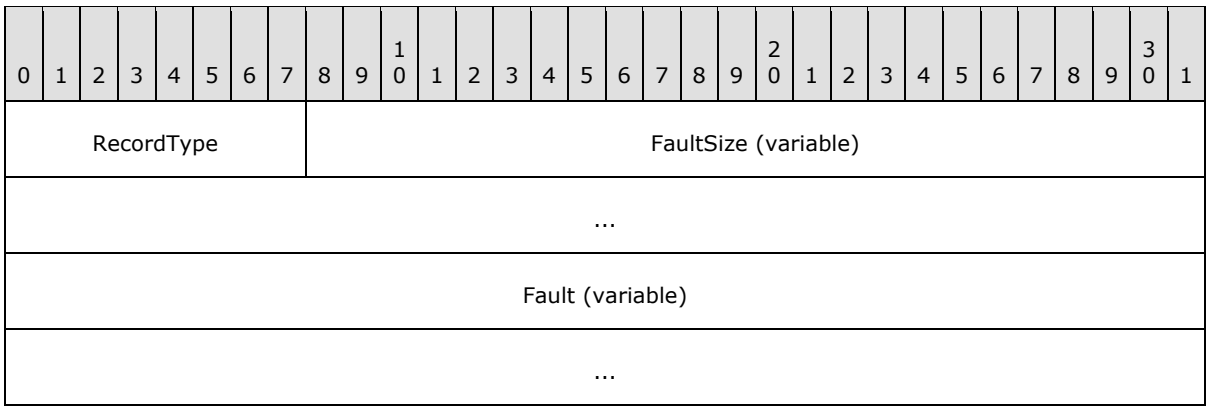
DataChunk1 (variable): The first chunk of message data. This MUST be present.

Data Chunk2 (variable): Successive chunks of message data. Additional chunks MAY be present if the message is split across multiple chunks.

Terminator (1 byte): This marks the end of chunks, and MUST be set to 0x00.

2.2.5 Fault Records

When an error is encountered while processing a message frame, a node MAY notify the sender of the error by sending a Fault Record. Generation of a Fault Record is purely for informational purposes.



RecordType (1 byte): This octet MUST be set to 0x08 to indicate this is a Fault record.

FaultSize (variable): The value MUST be set to the size of the fault in octets, encoded based on the scheme described in section 2.2.2. The size MUST NOT be set to 0.

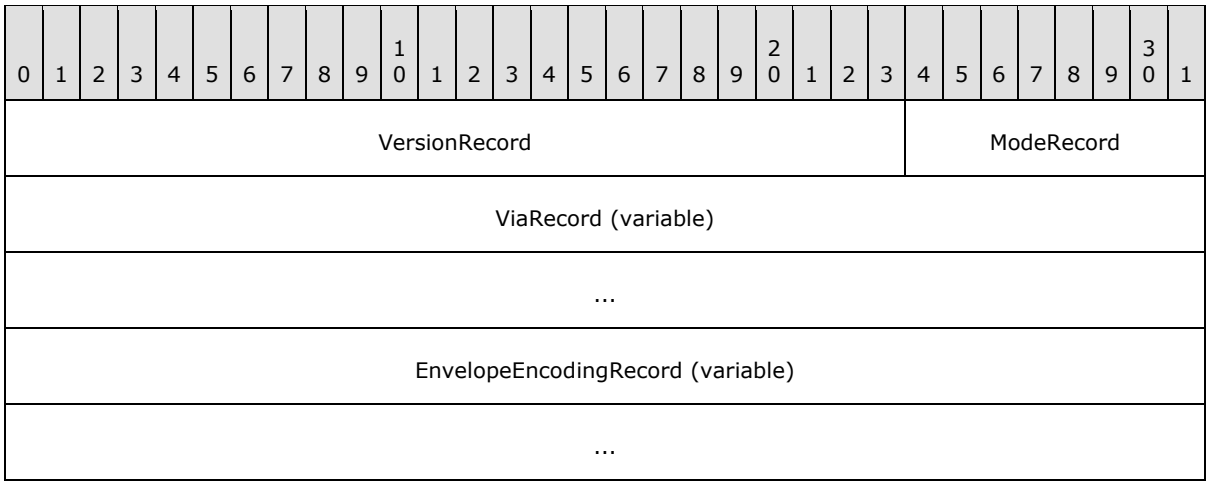
Fault (variable): A URI (as defined by [RFC2396] except that the 'escaped' construct is never used). The URI is encoded using UTF-8. The table below defines a collection of faults. An implementation MAY support these fault values, and MAY additionally define new ones.<4>

For convenience, the URI is broken into a namespace and fault name in this description. The namespace for faults in the table below is http://schemas.microsoft.com/ws/2006/05/framing/faults/. Any additional faults that are defined MUST NOT use this namespace.

Fault name	Meaning
ConnectionDispatchFailed	Endpoint referenced by the Via record exists, but the attempt to dispatch the message to the endpoint failed.
ContentTypeInvalid	Envelope Encoding Record sent is not supported by the endpoint.
ContentTypeTooLong	Receiver is enforcing a maximum content type size, and the Envelope Encoding Record exceeded that quota.
EndpointAccessDenied	Endpoint referenced by the Via record could not be accessed.
EndpointNotFound	Endpoint referenced by the Via record could not be found.
EndpointPaused	Endpoint referenced by the Via record exists, but the endpoint is currently paused.
EndpointUnavailable	Endpoint referenced by the Via record exists, but the endpoint is currently unavailable.
InvalidRecordSequence	The record sequence does not conform to the grammar outlined in section 3.1.1.2.
MaxMessageSizeExceededFault	Receiver is enforcing a maximum message size, and the incoming message has exceeded that quota.
ServerTooBusy	Endpoint does not have sufficient resources to process the connection.
ServiceActivationFailed	Endpoint is in a process that could not be activated.
UnsupportedMode	Mode record value is not supported by the destination.
UnsupportedVersion	Version record value is not supported by the destination.
UpgradeInvalid	Upgrade requested is not supported by the remote endpoint.
ViaTooLong	Receiver is enforcing a maximum Via size, and the Via record exceeded that quota.

2.2.6 Preamble Message

To aid description, a Preamble Message is defined for an initial record sequence. The Preamble Message may apply to multiple messages depending on the Mode specified.



3 Protocol Details

A node can participate in this protocol in one of the two roles: **initiator** or receiver. An initiator initiates the process by sending the preamble message. Subsequently, the initiator and receiver send messages respecting these properties. Before going into details for the initiator and receiver, common functionality is covered.

3.1 Common Details

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with what is described in this document.

3.1.1.1 Initiator – Receiver Interactions

This section describes some typical interactions between an Initiator and the Receiver.

3.1.1.1.1 Singleton Unsized Mode

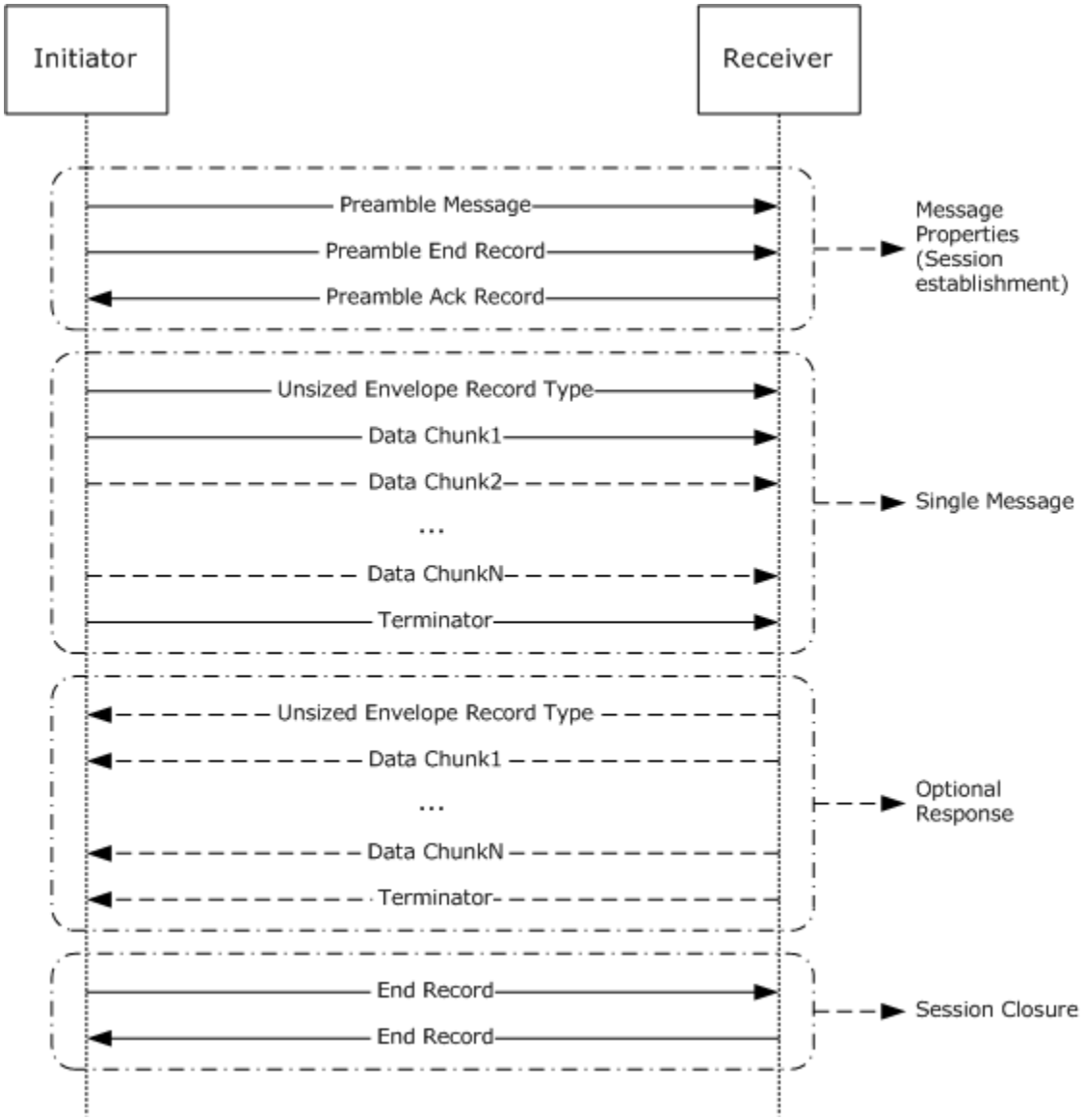


Figure 3: Singleton Unsized Mode

3.1.1.1.2 Duplex Mode

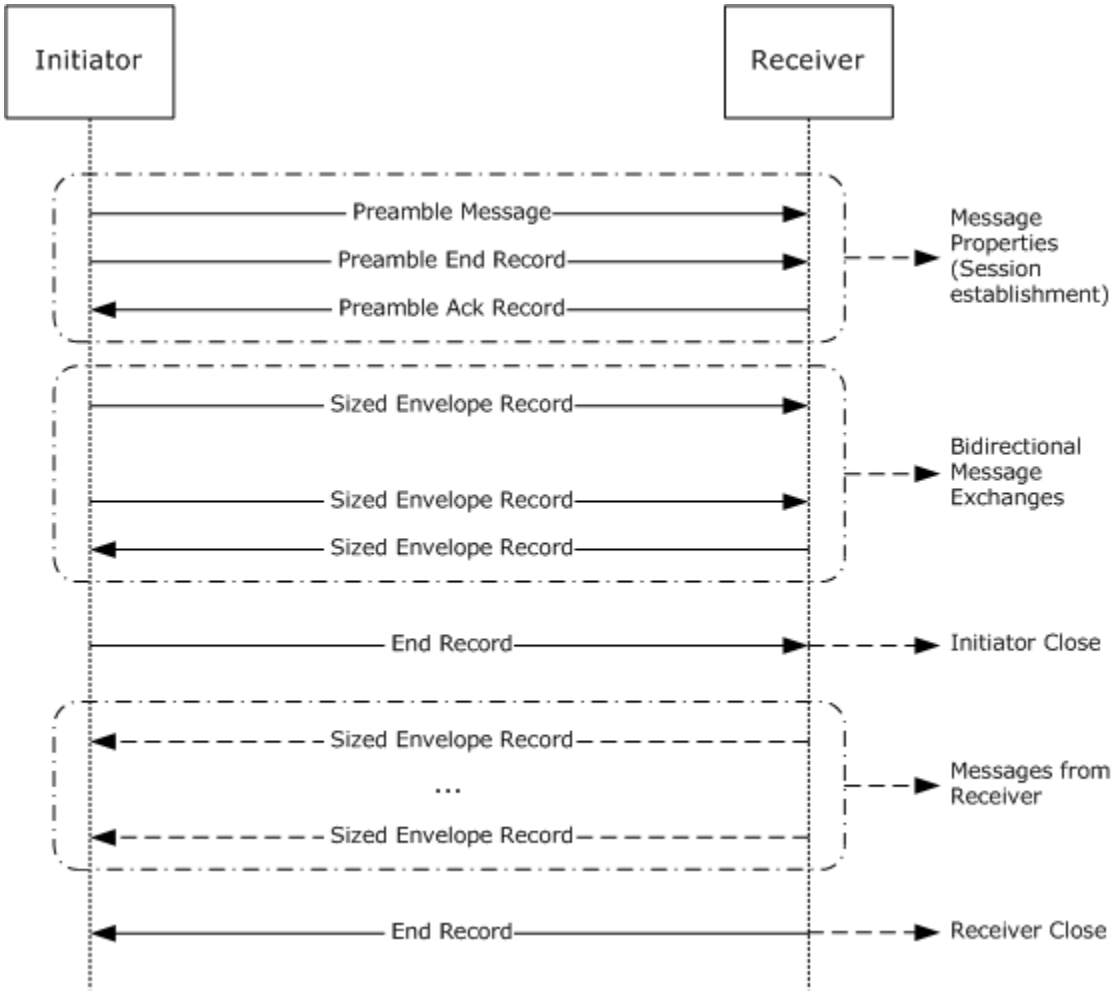


Figure 4: Duplex Mode

Note that in the case illustrated, the initiator sends the End Record first. The protocol allows either node to send the End Record first. After the End Record has been sent by a node, it can continue to receive messages.

3.1.1.1.3 Simplex Mode

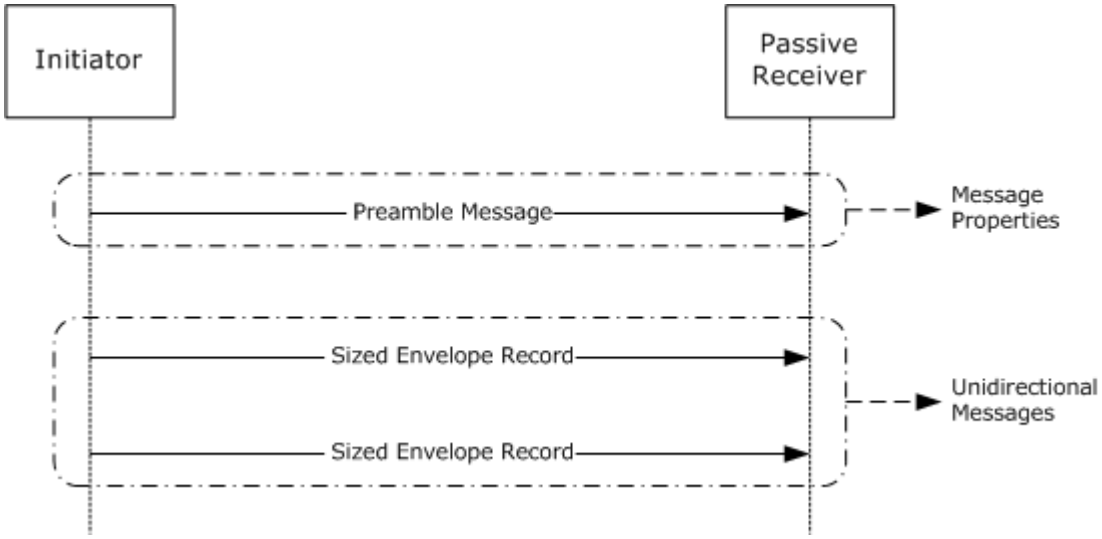


Figure 5: Simplex Mode

3.1.1.1.4 Singleton Sized Mode

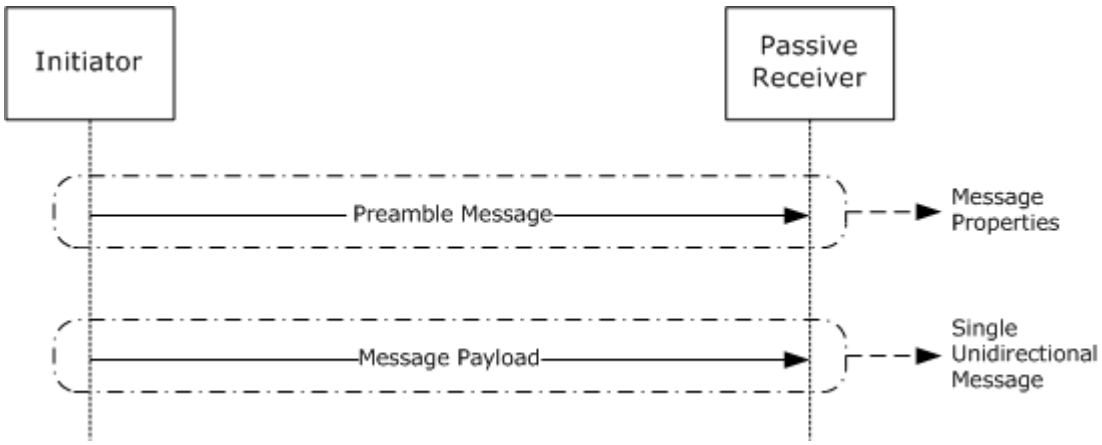


Figure 6: Singleton Sized Mode

3.1.1.1.5 Upgrades

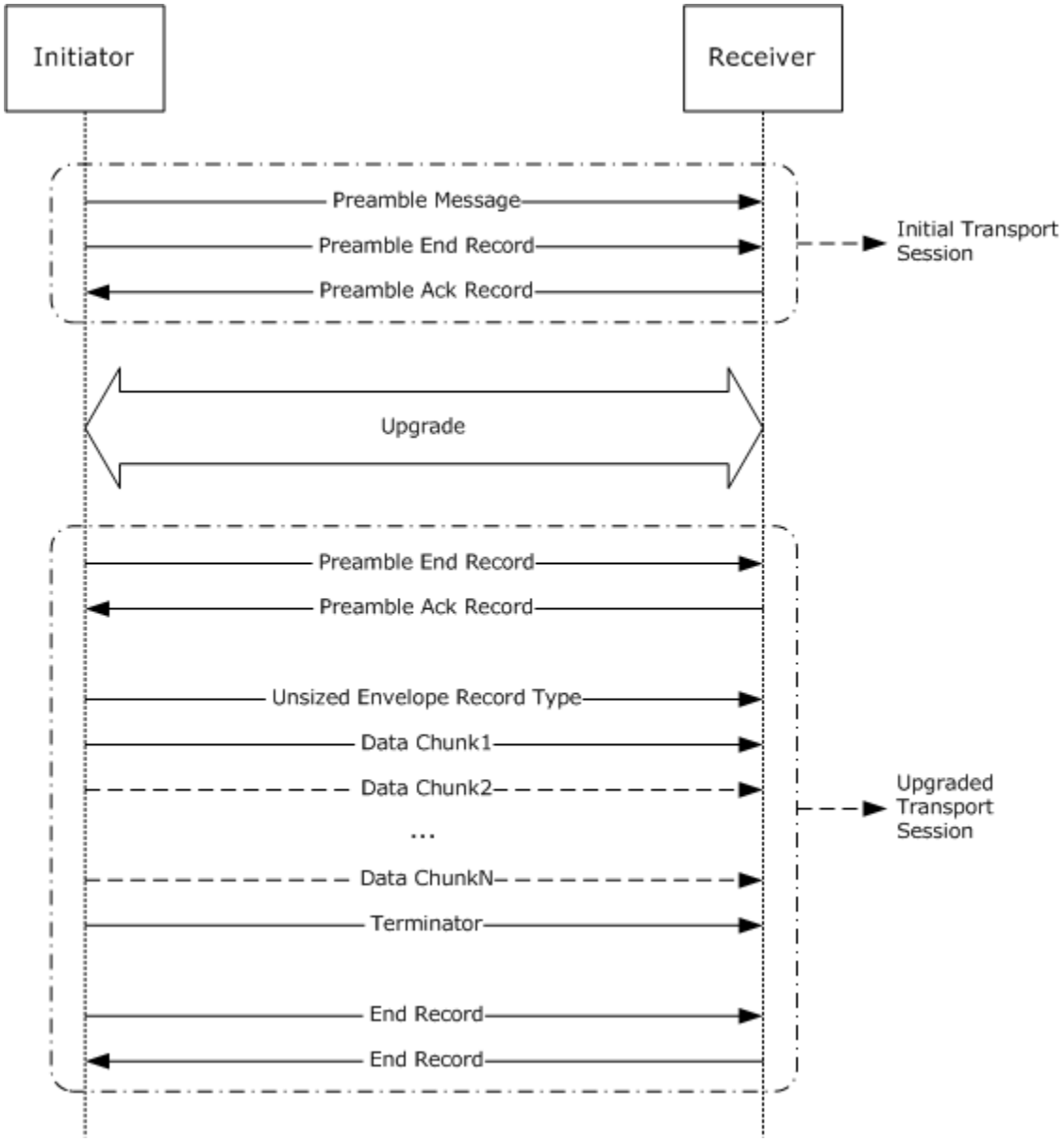


Figure 7: Upgrades

After the protocol upgrade, subsequent protocol exchange happens over the upgraded transport stream until a fault occurs or an End Record is received. Note that the stream upgrade using the Singleton Unsized mode is illustrated. The picture would look very similar when using the Duplex mode. Also, while the protocol allows for multiple upgrades, the above exchange only illustrates a single upgrade.

3.1.1.1.6 Faults

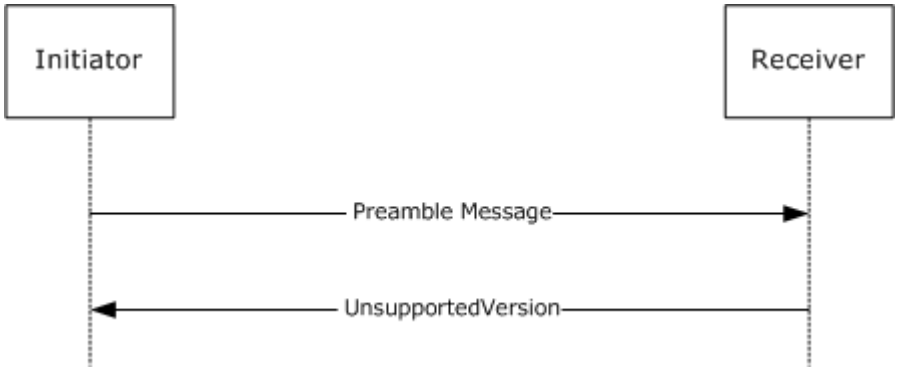


Figure 8: Unsupported Version

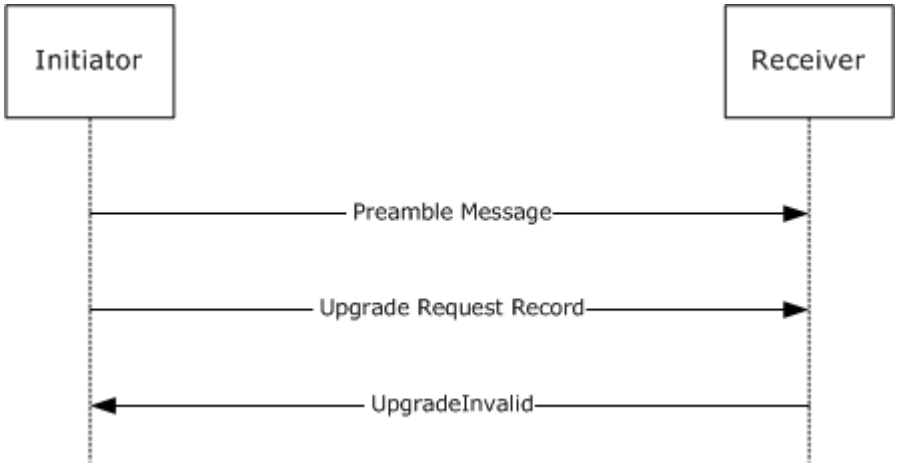


Figure 9: Upgrade Invalid

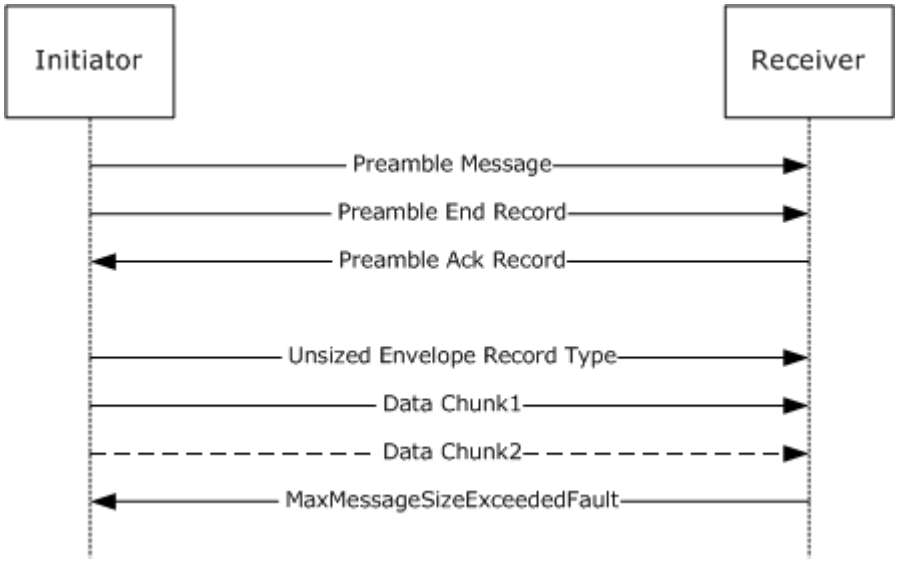


Figure 10: Maximum Message Size Exceeded

The above exchanges capture some of the scenarios where a Fault Record can be generated.

3.1.1.2 Protocol Grammar

This section uses the Augmented Backus-Naur Form (ABNF) notation specified in [RFC2234] to define the protocol stream grammar. ProtocolStream-a represents the stream of octets flowing from the initiator to the receiver, and ProtocolStream-b represents the stream of octets flowing from the receiver to the initiator.

```
ProtocolStream-a =
   1*(SingletonUnsizedStream-a / DuplexStream-a /
         SimplexStream-a / SingletonSizedStream-a)


ProtocolStream-b =
   1*(SingletonUnsizedStream-b / DuplexStream-b)
SingletonUnsizedStream-a =
   VersionRecord ModeRecordType SingletonUnsizedMode
   ViaRecord EncodingRecord
   *UpgradeRequest PreambleEndRecord
   UnsizedEnvelopeRecord
   EndRecord
DuplexStream-a =
   VersionRecord ModeRecordType DuplexMode
   ViaRecord EncodingRecord
   *UpgradeRequest PreambleEndRecord
   *SizedEnvelopeRecord
   EndRecord

SimplexStream-a =
   VersionRecord ModeRecordType SimplexMode
   ViaRecord EncodingRecord
   *SizedEnvelopeRecord
   EndRecord

SingletonSizedStream-a =
   VersionRecord ModeRecordType SingletonSizedMode
   ViaRecord EncodingRecord
   Octets
SingletonUnsizedStream-b =
(*UpgradeResponse FaultRecord) /
(*UpgradeResponse PreambleAckRecord *1(UnsizedEnvelopeRecord)
 (FaultRecord / EndRecord))
DuplexStream-b =
   (*UpgradeResponse FaultRecord) /
   (*UpgradeResponse PreambleAckRecord *SizedEnvelopeRecord
   (FaultRecord / EndRecord))
EncodingRecord = KnownEncodingRecord / ExtensibleEncodingRecord
UpgradeRequest = UpgradeRequestRecord Octets
UpgradeResponse = UpgradeResponseRecord Octets
VersionRecord = VersionRecordType MajorVersionNumber MinorVersionNumber
VersionRecordType = %x00
MajorVersionNumber = %x01
MinorVersionNumber = %x00
ModeRecordType = %x01
SingletonUnsizedMode = %x01
```

```
DuplexMode = %x02
SimplexMode = %x03
SingletonSizedMode = %x04
ViaRecord = ViaRecordType EncodedSize Utf8Octets
ViaRecordType = %x02
KnownEncodingRecord = KnownEncodingRecordType KnownEncodingType
KnownEncodingType = TextEncoding / BinaryEncoding / MtomEncoding
BinaryEncoding =
   BinarySessionlessEncoding /
   BinarySessionEncoding
TextEncoding =
   Soap11TextEncoding /
   Soap12TextEncoding
Soap11TextEncoding =
   Soap11Utf8Encoding /
   Soap11Utf16Encoding /
   Soap11UnicodeFFFEEncoding
Soap12TextEncoding =
   Soap12Utf8Encoding /
   Soap12Utf16Encoding /
   Soap12UnicodeFFFEEncoding
KnownEncodingRecordType = %x03
Soap11Utf8Encoding = %x00
Soap11Utf16Encoding = %x01
Soap11UnicodeFFFEEncoding = %x02
Soap12Utf8Encoding = %x03
Soap12Utf16Encoding = %x04
Soap12UnicodeFFFEEncoding = %x05
MtomEncoding = %x06
BinarySessionlessEncoding = %x07
BinarySessionEncoding = %x08

ExtensibleEncodingRecord =
   ExtensibleEncodingRecordType EncodedSize Utf8Octets
ExtensibleEncodingRecordType = %x04

UnsizedEnvelopeRecords =
   UnsizedEnvelopeRecordType 1*(EncodedSize Octets) Terminator
UnsizedEnvelopeRecordType = %x05
Terminator = %x00
SizedEnvelopeRecord = SizedEnvelopeRecordType EncodedSize Octets
SizedEnvelopeRecordType = %x06
EndRecord = EndRecordType
EndRecordType = %x07
FaultRecord = FaultRecordType EncodedSize Utf8Octets
FaultRecordType = %x08
UpgradeRequestRecord = UpgradeRequestRecordType EncodedSize Utf8Octets
UpgradeRequestRecordType = %x09
UpgradeResponseRecord = UpgradeResponseRecordType
UpgradeResponseRecordType = %x0A
PreambleAckRecord = PreambleAckRecordType
PreambleAckRecordType = %x0B
PreambleEndRecord = PreambleEndRecordType
PreambleEndRecordType = %x0C
Utf8Octets = 1*(Utf8Octet)
Utf8Octet =
   %x00-7F /
   %xC2-DF %x80-BF /
```

```
   %xE0-EF %x80-BF %x80-BF /
   %xF0-F4 %x80-BF %x80-BF %x80-BF
Octets = 1*(%x00-FF)
EncodedSize =
%x01-7F /
   %x80-FF %x01-7F /
   %x80-FF %x80-FF %x01-7F /
   %x80-FF %x80-FF %x80-FF %x01-7F /
   %x80-FF %x80-FF %x80-FF %x80-FF %x01-07
```

3.1.2 Timers

The protocol inherently doesn't need to support any timers.

3.1.3 Initialization

There is no global state that the protocol needs to initialize.

The protocol maintains the following state on a per session basis:

- Protocol Configuration Object – Determines the specific transport, protocol version, mode, via, and message encoding scheme to be used for this session. How the PCO is configured is not addressed in this protocol. It is made available to the protocol as part of a higher-layer triggered event.
- Send Allowed – A Boolean value that can be set to TRUE or FALSE to indicate whether messages can be sent on this session or not. It is initialized to FALSE.
- Receive Allowed – A Boolean value that can be set to TRUE or FALSE to indicate whether messages can be received on this session or not. It is initialized to FALSE.

3.1.4 Higher-Layer Triggered Events

This section covers reading record types from the underlying transport. The higher-layer triggered events and related processing is role specific.

The following stipulations apply throughout the remaining sections.

Wherever it's mentioned that session MUST be closed, it refers to the following actions being taken by the protocol:

- Any session-related state MUST be discarded.
- The protocol MUST notify the higher layer of the error.

Wherever it is mentioned that a Fault Record MAY (or SHOULD) be sent, it refers to the following action being taken by the protocol:

- If the mode is Singleton, Unsized, or Duplex mode, a Fault Record MAY (or SHOULD) be sent, as described in section 2.2.5.

3.1.4.1 Reading Variable-Sized Records

When a variable-sized record is received, the protocol MUST follow the below algorithm to decode the size and read the payload. This section assumes that the record type has already been read.

The algorithm takes the MaxSize (the maximum supported size for this record) as input. If the encoded size is 0, a Fault Record MAY <5> be sent, indicating that the size is 0, and the session MUST be closed. The decoded size is returned in little-endian format.

```
Size = 0
Read the first octet
If (first octet is 0)
{
    Fault
}
While Most Significant Bit in octet is 1
{
    Append the last 7 bits of the octet to Size
    If the Size exceeds MaxSize
    {
        Fault
    }
    Read the next octet
}
Read n octets where n = Size
```

Figure 11: Algorithm to Decode the Size and Read the Payload

3.1.4.2 Handling Receipt of an Unexpected Record Type

If the protocol receives an unexpected record type, it MUST be handled as follows:

- If the Record Type is not Fault Record, a Fault Record MAY be sent indicating that an unexpected record type has been received.
- The session MUST be closed.

3.1.4.3 Version Record

- If the Record Type is not Version Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST read the next two octets, which contain the Major and Minor Versions of the protocol being used.
- If the protocol does not understand the Version, a Fault Record MAY <6> be sent indicating that an incorrect mode has been specified, and the session MUST be closed.

3.1.4.4 Mode Record

- If the Record Type is not Mode Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST read the next octet, which contains the mode.
- If the mode is incorrect for the session, a Fault Record MAY <7> be sent indicating that an incorrect mode has been specified, and the session MUST be closed.

3.1.4.5 Via Record

- If the Record Type is not Via Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST obtain the Via, as detailed in section 3.1.4.1. The protocol SHOULD use a MaxViaSize<8>. If the Via is too long, a Fault Record MAY<9> be sent, and the session MUST be closed.

- If the protocol cannot locate an endpoint matching the Via, a Fault Record MAY<10> be sent, and the session MUST be closed.

3.1.4.6 Encoding Record

- If the Record Type is not Known Encoding Record or Extensible Encoding Record, it MUST be handled as described in section 3.1.4.2.
- If the Record Type is Known Encoding Record, the protocol MUST read the next octet, which contains the message encoding scheme.
- If the Record Type is Extensible Encoding Record, the protocol MUST obtain the encoding scheme, as detailed in section 3.1.4.1. The protocol SHOULD use a MaxContentTypeSize<11> . If the content type is too long, a Fault Record MAY<12> be sent, and the session MUST be closed.
- If the encoding is not supported, a Fault Record MAY<13> be sent, and the session MUST be closed.

3.1.4.7 Upgrade Request Record

- If the Record Type is not Upgrade Request Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST read the Upgrade Protocol, as detailed in section 3.1.4.1. The protocol SHOULD use a MaxUpgradeProtocolSize.<14> If the upgrade name is too long, a Fault Record MAY<15> be sent, and the session MUST be closed.
- If the upgrade is not supported, a Fault Record MAY<16> be sent, and the session MUST be closed.
- If the upgrade is supported, the protocol MUST send an Upgrade Response Record, as described in section 2.2.3.6. The protocol MUST invoke the appropriate upgrade handler. How the upgrade handler achieves the upgrade is not addressed in this document.

3.1.4.8 Upgrade Response Record

- If the Record Type is not Upgrade Response Record, it MUST be handled as described in section 3.1.4.2.
- If the upgrade is supported, the protocol MUST invoke the appropriate upgrade handler. How the upgrade handler achieves the upgrade is outside the scope of this document.

3.1.4.9 Preamble End Record

- If the Record Type is not Preamble End Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST send a Preamble Ack Record as described in section 2.2.3.8.

3.1.4.10 Preamble Ack Record

If the Record Type is not Preamble Ack Record, it MUST be handled as described in section 3.1.4.2.

3.1.4.11 Sized Envelope Record

- If the Record Type is not Sized Envelope Record, it MUST be handled as follows:

- If the Record Type is End Record, the protocol MUST notify the higher layer of the receipt of End Record, and set Receive Allowed to FALSE.
- If the Record Type is a Fault Record, the session MUST be closed.
- Otherwise, it MUST be handled as described in section 3.1.4.2.

- The protocol MUST obtain the message as detailed in section 3.1.4.1. The protocol SHOULD use a MaxEnvelopeSize.<17>

  If the message is too large, a Fault Record MAY<18> be sent, and the session MUST be closed.

3.1.4.12 Unsized Envelope Record

- If the Record Type is not Unsized Envelope Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST then process the first chunk and any additional chunks as described in section 3.1.4.1 until the Terminator marker (octet 0x00) is read. To achieve streaming, reading chunks SHOULD be correlated with consumption of chunks by the higher layer. The protocol SHOULD use a MaxChunkSize<19> . If the chunk size is too large, a Fault Record MAY<20> be sent, and the session MUST be closed.

3.1.4.13 End Record

- If the Record Type is not End Record, it MUST be handled as described in section 3.1.4.2.
- The protocol MUST set Receive Allowed to FALSE.

3.1.5 Message Processing Events and Sequencing Rules

This document assumes that processing of received octets is deferred until initiated by a higher-layer triggered event or a required response in the protocol. All message processing events and sequencing rules are explained in the context of higher-layer triggered events.

3.1.6 Timer Events

As the protocol does not support any timers, there are no timer events.

3.1.7 Other Local Events

3.1.7.1 Underlying Transport Session Is Closed

If, at any point, the underlying network transport session is closed, the Protocol Stream is closed. The protocol MUST discard any session-related state.

3.2 Initiator Details

3.2.1 Abstract Data Model

The details are covered in section 3.1.1.

3.2.2 Timers

3.2.3 Initialization

The details are covered in section 3.1.3.

3.2.4 Higher-Layer Triggered Events

The operation of the protocol is driven by the following higher-layer triggered events.

3.2.4.1 Initialize Session

A new session state MUST be created, and session properties initialized as described in section 3.1.3.

3.2.4.2 Send Preamble

- The protocol MUST send the Preamble Message as described in section 2.2.6.
- In the case of Singleton, Unsized, and Duplex modes, the protocol MUST perform the following additional steps:
  - If an upgrade is required, send the Upgrade Request Record as described in section 2.2.3.5.
  - If an upgrade is sent, read the Upgrade Response Record as described in section 3.1.4.8.
  - Send the Preamble End Record as described in section 2.2.3.7.
  - Read the Preamble Ack Record as described in section 3.1.4.10.
- The protocol MUST set Send Allowed to TRUE.
- If the Mode is Duplex, the protocol MUST set Receive Allowed to TRUE.

3.2.4.3 Send Message

If Send Allowed is set to FALSE, an error MUST be propagated to the higher layer, and no further processing done. Otherwise, the protocol MUST do the following based on the Mode.

3.2.4.3.1 Singleton Unsized Mode

- The protocol MUST send an Unsized Envelope Record containing the message as described in section 2.2.4.3.
- The protocol MUST send an End Record as described in section 2.2.3.9.
- The protocol MUST set Send Allowed to FALSE.

3.2.4.3.2 Duplex or Simplex Mode

The protocol MUST send a Sized Envelope Record containing the message as described in section 2.2.4.1.

3.2.4.3.3 Singleton Sized Mode

The protocol MUST send the message and set Send Allowed to FALSE.

3.2.4.4 Receive Message

If Receive Allowed is set to FALSE, an error MUST be propagated to the higher layer and no further processing done. Otherwise, the protocol MUST read a Sized Envelope Record as described in section 3.1.4.11, and propagate the contained message to a higher layer.

3.2.4.5 Send End Record

If Mode is not Duplex or Simplex, an error MUST be propagated to the higher layer and no further processing done. Otherwise, the protocol MUST send an End Record as described in section 2.2.3.9. The protocol MUST set Send Allowed to FALSE.

3.2.4.6 Session Close

The protocol MUST discard any session-related state and no further processing done.

3.2.5 Message Processing Events and Sequencing Rules

The details are covered in section 3.1.5.

3.2.6 Timer Events

3.2.7 Other Local Events

The details are covered in section 3.1.7.

3.3 Receiver Details

3.3.1 Abstract Data Model

The details are covered in section 3.1.1.

3.3.2 Timers

3.3.3 Initialization

The details are covered in section 3.1.3.

3.3.4 Higher-Layer Triggered Events

The operation of the protocol is driven by the following higher-layer triggered events.

3.3.4.1 Initialize Session

A new session state MUST be created and session properties initialized as described in section 3.1.3.

3.3.4.2 Receive Preamble

- The protocol MUST read the Version Record, as described in section 3.1.4.3.
- The protocol MUST read the Mode Record, as described in section 3.1.4.4.
- The protocol MUST read the Via Record, as described in section 3.1.4.5.

- The protocol MUST read the Encoding Record, as described in section 3.1.4.6.
- If the Mode is Singleton, Unsized, or Duplex, the protocol MUST perform these additional steps:
  - If an upgrade is required, read the Upgrade Request Record, as described in section 3.1.4.7.
  - Read the Preamble End Record, as described in section 3.1.4.9.
- The protocol MUST set Receive Allowed to TRUE.
- If the Mode is Duplex, the protocol MUST set Send Allowed to TRUE.

3.3.4.3 Send Message

If Send Allowed is set to FALSE, an error MUST be propagated to the higher layer and no further processing done. Otherwise, the protocol MUST send a Sized Envelope Record containing the message as described in section 2.2.4.1.

3.3.4.4 Receive Message

If Receive Allowed is set to FALSE, an error MUST be propagated to the higher layer and no further processing done. Otherwise, the protocol MUST do the following based on the Mode.

3.3.4.4.1 Singleton Unsized Mode

- The protocol MUST read an Unsized Envelope Record as described in section 3.1.4.12, and propagate the contained message to a higher layer.
- The protocol MUST read an End Record as described in section 3.1.4.13.
- The protocol MUST set Receive Allowed to FALSE.

3.3.4.4.2 Duplex or Simplex Mode

The protocol MUST read a Sized Envelope Record as described in section 3.1.4.11, and propagate the contained message to a higher layer.

3.3.4.4.3 Singleton Sized Mode

The protocol MUST read the message and propagate it to a higher layer. The protocol MUST set Receive Allowed to FALSE.

3.3.4.5 Send End Record

If mode is not Duplex, an error MUST be propagated to the higher layer and no further processing done. Otherwise, the protocol MUST send an End Record as described in section 2.2.3.9. The protocol MUST set Send Allowed to FALSE.

3.3.4.6 Session Close

The protocol MUST discard any session-related state and no further processing done.

3.3.5 Message Processing Events and Sequencing Rules

The details are covered in section 3.1.5.

3.3.6 Timer Events

3.3.7 Other Local Events

The details are covered in section 3.1.7.

4 Protocol Examples

4.1 Duplex Mode

The protocol exchange involving a duplex mode session is illustrated in this section. The initiator first establishes a session with the receiver. The initiator then sends a message, and the receiver replies. Finally, the session is closed. The Protocol Configuration Object for this session has been configured as follows:

- Transport – The specifics of network transport are excluded from this example. The packet captures below demonstrates only the .NET Message Framing Protocol and message payload.
- Version – This exchange happened over Major Version 1 and Minor Version 0 of this protocol.
- Mode – Duplex mode was used.
- Via – The receiver was identified by the URI net.tcp://SampleServer/SampleApp/.
- Encoding – Binary Session Encoding was used to encode the messages.

4.1.1 Initiator Receiver : Preamble Message

Raw Message

```
00000030                                                 00 01  ..
00000040  00 01 02 02 21 6E 65 74 2E 74 63 70 3A 2F 2F 53  ....!net.tcp://S
00000050  61 6D 70 6C 65 53 65 72 76 65 72 2F 53 61 6D 70 ampleServer/Samp
00000060  6C 65 41 70 70 2F 03 08                          leApp/..
```

Parsed Message

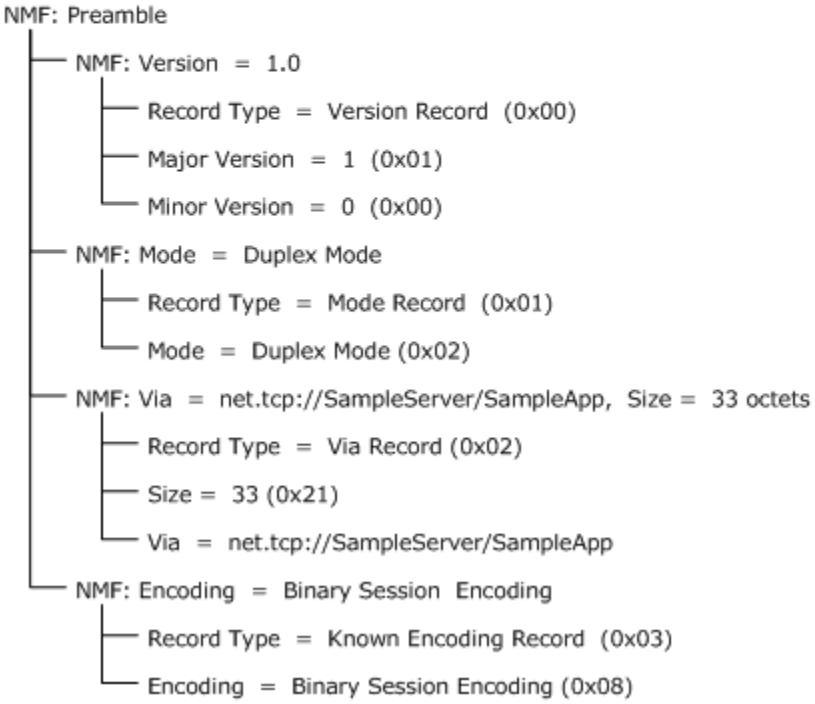


Figure 12: Initiator Receiver : Preamble Message

4.1.2 Initiator Receiver : Preamble End Message

Raw Message

```
00000030                                          0C            .
```

Parsed Message

```
NMF: Preamble End
  └── NMF: Record Type  =  Preamble End Record  (0x0C)
```

Figure 13: Initiator Receiver : Preamble End Message

4.1.3 Receiver Initiator : Preamble Ack Message

Raw Message

```
00000030                                          0B            .
```

Parsed Message

```
NMF: Preamble Ack Message
  └── NMF: Record Type = Preamble Ack Record (0x0B)
```

Figure 14: Receiver Initiator : Preamble Ack Message

4.1.4 Initiator Receiver : Sized Envelope Message

Raw Message

```
00000030                                                06 AA                 ..
00000040  01 74 2A 68 74 74 70 3A 2F 2F 74 65 6D 70 75 72  .t*http://tempur
00000050  69 2E 6F 72 67 2F 49 4F 6E 65 57 61 79 43 6F 6E  i.org/IOneWayCon
00000060  74 72 61 63 74 2F 45 78 65 63 75 74 65 21 6E 65  tract/Execute!ne
00000070  74 2E 74 63 70 3A 2F 2F 53 61 6D 70 6C 65 53 65  t.tcp://SampleSe
00000080  72 76 65 72 2F 53 61 6D 70 6C 65 41 70 70 2F 07  rver/SampleApp/.
00000090  45 78 65 63 75 74 65 13 68 74 74 70 3A 2F 2F 74   Execute.http://t
000000A0  65 6D 70 75 72 69 2E 6F 72 67 2F 0A 73 65 6E 64  empuri.org/.send
000000B0  53 74 72 69 6E 67 56 02 0B 01 73 04 0B 01 61 06  StringV...s...a.
000000C0  56 08 44 0A 1E 00 82 AB 01 44 0C 1E 00 82 AB 03  V.D......D......
000000D0  01 56 0E 42 05 0A 07 42 09 99 0D 54 65 73 74 20  .V.B...B...Test.
000000E0  6D 65 73 73 61 67 65 31 01 01 01                 message1..
```

Parsed Message

```
NMF: Sized Envelope Message, Message Size = 170
  ├── NMF: Record Type  =  Sized Envelope Record  (0x06)
  ├── NMF: Size  = 170  (0xAA 0x01)
  └── +NMF: Payload: Number of payload octets remaining = 170
```

Figure 15: Initiator Receiver : Sized Envelope Message

4.1.5 Receiver Initiator : Sized Envelope Message

Raw Message

```
00000030                                                 06 36              .6
00000040  00 56 02 0B 01 73 04 0B 01 61 06 56 08 44 0A 1E .V...s...a.V.D..
00000050  00 82 AB 01 44 0C 1E 00 82 AB 03 01 56 0E 42 05 ....D.......V.B.
00000060  0A 07 42 09 99 0D 54 65 73 74 20 6D 65 73 73 61 ..B...Test.messa
00000070  67 65 32 01 01 01                                ge2..
```

Parsed Message

```
NMF: Sized Envelope Message, Message Size = 54
  ├─ NMF: Record Type  =  Sized Envelope Record  (0x06)
  ├─ NMF: Size  =  54  (0x36)
  └─ +NMF: Payload: Number of payload octets remaining = 54
```

Figure 16: Receiver Initiator : Sized Envelope

4.1.6 Initiator Receiver : End Message

Raw Message

```
00000030                                           07           .
```

Parsed Message

```
NMF: End Message
  └─ NMF: Record Type  =  End Record  (0x07)
```

Figure 17: Initiator Receiver : End Message

4.1.7 Receiver Initiator : End Message

Raw Message

```
00000030                                           07           .
```

Parsed Message

```
NMF: End Message
  └─ NMF: Record Type = End Record (0x07)
```

Figure 18: Receiver Initiator : End Message

5 Security

5.1 Security Considerations for Implementers

To minimize the risk of a denial-or-service attack, an implementation of this protocol SHOULD limit the size of variable-length records, including Via, Extensible Encoding, Upgrade Protocol, Sized Envelope, and Unsized Envelope Record chunks. Note that Via, Extensible Encoding, and Upgrade Protocol records are exchanged before a stream upgrade can supply transport level security. Therefore, particular care should be taken to bound these records to a reasonable size if security is not available.

5.2 Index of Security Parameters

There are no security parameters.

6 Appendix A: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows XP
- Windows Storage Server 2003
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 1.3: The Windows implementation of this protocol is exercised through the use of the following Windows Communication Framework Bindings [MSDN-WCF].

1. NetTcpBinding [MSDN-NETTcp] – If the TransferMode property on the binding is set to Buffered, the mode will be set to Duplex. Otherwise, the mode will be set to Singleton Unsized. If the Security.Transport.ClientCredentialType property on the binding is set to Certificate, the "SSL/TLS" upgrade protocol will be used. Otherwise, if it is set to Windows, the "Negotiate" upgrade protocol will be used.

2. NetNamedPipeBinding [MSDN-NETNamedPipe] – If the TransferMode property on the binding is set to Buffered, mode will be set to Duplex. Otherwise, the mode will be set to Singleton Unsized. If the Security.Mode property on the binding is set to Transport, the "Negotiate" upgrade protocol will be used.

3. NetMsmqBinding [MSDN-NETMsmq] – If a TransactionScope is being used, the mode will be set to Simplex. Otherwise, the mode will be set to Singleton Sized. If the Security.Transport.MsmqAuthenticationMode property on the binding is set to Certificate, the "SSL/TLS" upgrade protocol will be used. Otherwise, if it is set to WindowsDomain, the "Negotiate" upgrade protocol will be used.

The Windows implementation of this protocol is also exercised through a custom Windows Communication Framework Binding which uses the TcpTransportBindingElement [MSDN-NETTcpBE] or the NamedPipeTransportBindingElement [MSDN-NETNamedPipeBE] or the MsmqTransportBindingElement [MSDN-NETMsmqBE].

<2> Section 2.2.3.4.1: The Windows implementation of this protocol supports all of the known encoding schemes.

<3> Section 2.2.3.5: The Windows implementation of this protocol supports only SSL/TLS and Negotiate upgrade protocols.

<4> Section 2.2.5: TheWindows implementation of this protocol supports the following set of faults – ContentTypeInvalid, ContentTypeTooLong, ConnectionDispatchFailed, EndpointNotFound, EndpointUnavailable, MaxMessageSizeExceededFault, ServerTooBusy, ServiceActivationFailed, UnsupportedMode, UnsupportedVersion, UpgradeInvalid and ViaTooLong.

<5> Section 3.1.4.1: The Windows implementation of this protocol doesn't send a Fault Record if the size of a variable sized record is 0.

<6> Section 3.1.4.3: The Windows implementation of this protocol sends a Fault Record (UnsupportedMode) if an incorrect Mode is specified in the received Mode Record.

<7> Section 3.1.4.4: The Windows implementation of this protocol sends a Fault Record (UnsupportedMode) if an incorrect Mode is specified in the received Mode Record.

<8> Section 3.1.4.5: The Windows implementation of this protocol defines a MaxViaSize of 2048 bytes.

<9> Section 3.1.4.5: The Windows implementation of this protocol does not send a Fault Record if the size of Via in the received Via Record exceeds MaxViaSize.

<10> Section 3.1.4.5: The Windows implementation of this protocol sends a Fault Record (EndpointNotFound) if the endpoint can't be located for the specified Via in the received Via Record.

<11> Section 3.1.4.6: The Windows implementation of this protocol defines a MaxContentTypeSize of 256 bytes.

<12> Section 3.1.4.6: The Windows implementation of this protocol doesn't send a Fault Record if the size of extensible encoding in the received Extensible Encoding Record exceeds MaxContentTypeSize.

<13> Section 3.1.4.6: The Windows implementation of this protocol sends a Fault Record (ContentTypeInvalid) if an unsupported content type is specified in the received Encoding Record.

<14> Section 3.1.4.7: The Windows implementation of this protocol defines a MaxUpgradeProtocolSize of 256 bytes.

<15> Section 3.1.4.7: The Windows implementation of this protocol doesn't send a Fault Record if the size of an upgrade protocol name in the received Upgrade Request Record exceeds MaxUpgradeProtocolSize.

<16> Section 3.1.4.7: The Windows implementation of this protocol sends a Fault Record (UpgradeInvalid) if an unsupported upgrade protocol name is specified in an Upgrade Request Record.

<17> Section 3.1.4.11: The Windows implementation of this protocol uses a MaxEnvelopeSize as configured externally.

<18> Section 3.1.4.11: The Windows implementation of this protocol sends a Fault Record (MaxMessageSizeExceededFault) if the size of the received Sized Envelope Record exceeds MaxEnvelopeSize but is not greater than 0xffffffff. No Fault Record is sent if the size of the received Sized Envelope Record exceeds 0xffffffff.

<19> Section 3.1.4.12: The Windows implementation of this protocol defines a MaxChunkSize of 0xfffffffa.

<20> Section 3.1.4.12: The Windows implementation of this protocol doesn't send a Fault Record if the size of a chunk in the received Unsized Envelope Record exceeds MaxChunkSize.

7 Index

S

T

U

V

W