

SOAP/TCP v1.0

May 15, 2007

This document specifies a simple, efficient and light weight transport over a TCP/IP connection that is suitable for the transmission of Web service messages (SOAP-based or XML Infoset-based).

Sun SOAP/TCP Non-Assertion Covenant

Sun Microsystems irrevocably covenants that, subject solely to the reciprocity requirement described below, it will not seek to enforce any of its enforceable U.S. or foreign patents against that portion of a product that implements the [SOAP/TCP v1.0](#) specification or any subsequent version of that specification in whose development Sun participates ("SOAP/TCP Implementation").

The foregoing covenant shall not apply and Sun makes no assurance, covenant or commitment not to assert or enforce any or all of its patent rights against any individual, corporation or other entity that asserts, threatens or seeks at any time to enforce its own or another party's U.S. or foreign patents or patent rights against any SOAP/TCP Implementation.

This statement is not an assurance either (i) that any of Sun's issued patents cover a SOAP/TCP Implementation or are enforceable, or (ii) that a SOAP/TCP Implementation would not infringe patents or other intellectual property rights of any third party.

No other rights except those expressly stated in this Non-Assertion Covenant shall be deemed granted, waived, or received by implication, or estoppel, or otherwise.

Table of Contents

1.Introduction.....	3
2.Terms and Definitions.....	3
3.Message frame.....	4
3.1.Message frame grammar.....	4
3.1.1.The frame.....	6
3.1.2.The header.....	6
3.1.3.The payload.....	6
3.1.4.The content-description.....	6
3.2.Message frame encoding.....	7
3.2.1.INTEGER4 type.....	7
3.2.2.INTEGER8 type.....	7
3.2.3.STRING type.....	7
3.2.4.OCTET type.....	7
3.2.5.NIBBLE type.....	8
4.Message.....	8
4.1.Message representation.....	8
4.2.Exchanging of messages.....	8
5.Message errors.....	9
5.1.Malformed message frames.....	9
5.2.Message frames with channel errors.....	9
5.3.Denial of service attacks.....	10
5.4.Error messages.....	10
5.4.1.Payload of the error message.....	10
5.4.2.The error message payload for malformed message frame errors.....	10
5.4.3.The error message payload for message frames with channel errors.....	11
6.Connection Management service.....	11
6.1.Connection Management service errors.....	11
7.Initiating a session.....	12
7.1.Protocol magic identifier.....	12
7.2.Protocol version.....	13
7.2.1.Protocol version grammar.....	13
8.Channels.....	13
8.1.Opening a channel.....	13
8.1.1.Content negotiation.....	13
8.2.Closing a channel.....	14
8.3.The service channel.....	14
9.Stated-based Fast Infoset.....	14
Appendix A. Connection Management web service description (WSDL).....	16
Appendix B. Connection Management web service type definitions (XSD).....	18
Appendix C. Data types encoding examples.....	20
I.INTEGER4.....	20
II.INTEGER8.....	20
Appendix D. References.....	21

1. Introduction

This document specifies a simple, efficient and light weight transport over a TCP/IP connection that is suitable for the transmission of Web service messages (SOAP-based or XML Infoset-based).

A client initiates a session (section 7) with a server by opening TCP/IP connection and negotiating the session requirements. Once initiated a client may open one or more channels (section 8), or “virtual connections”, for the transmission of messages over the channels to and from a server.

Messages are represented as one or more message frames (section 3). Transmission of a message over a channel is implemented by the transmission of the message frames over a session. The peers (client and server) are responsible for fragmenting messages into message frames. The frames associated with different channels cannot be interleaved.

Channel 0, or the service channel, is a specific channel that is implicitly opened on session initiation and is reserved for the transmission of messages associated with session and channel management (section 8.3). Such messages are SOAP messages, specified by the Connection Management service.

When opening a channel the client and server exchange messages on the service channel:

- the client sends the endpoint address that request messages on the channel are intended for; and
- the client and server negotiate the content types and parameters of the payload of messages that will be transmitted on the channel (section 8.1.1).

The endpoint address is an URI and the corresponding channel is utilized to transmit messages to and from that endpoint address (messages for other endpoint addresses are sent on other open channels). An endpoint address can be the address of a deployed SOAP-based Web service.

The content type is a MIME media type. A client will send a set of content types and and set of parameters it supports and the server will respond with subsets of the client sets that it supports. Messages sent on the channel will refer to a content type using a content identifier.

Since the endpoint address, content types and parameters are specified when a channel is opened the size in octets of an encoded message frame is small and the processing of a message frame is minimal.

NOTE – A 512 bytes of payload transmitted over channel 1 with a content identifier of 1 with no parameters is encoded in 516 bytes (a 4 byte overhead).

2. Terms and Definitions

Application message: A request or response message containing an application-defined payload that is sequence of one or more message frames of message, message-start-chunk, message-chunk and message-end-chunk (section 4.1).

Channel: The protocol agreed between a client and server for the transmission of messages in sequence.

Channel identifier: A non-negative integer that is uniquely associated with a channel.

Service channel: Channel 0 that is used for the transmission of messages to and from the Connection Management service.

Connection Management service: A Web service that is instantiated on session initiation and processes requests sent on service channel.

Client: A peer that initiates the creation of a session over a connection established with another peer (a server).

Connection: An established and open TCP/IP connection.

Content identifier: A non-negative integer that is uniquely associated with (within the scope of a channel) a content type.

Content type: Content specified by a MIME media type.

Error message: A response message containing a channel or session error associated with a request that is a sequence of one message frame of `error`.

Message: The basic unit of information transmitted over a channel. A message is represented as an ordered contiguous sequence of (one or more) message frames, with the same channel identifier, transmitted over a session.

Message frame: The basic unit of information transmitted over a session.

Null message: A request or response message containing no application-defined data that is a sequence of one message frame of `null`.

Peer: An entity capable of sending and receiving messages.

Request message: A message sent by a client to a server.

Response message: A message sent by a server to a client in response to a request message.

Server: A peer that responds to a client initiating the creation of a session to establish a created session.

Session: The protocol agreed between a client and server for management of channels and the transmission of message frames over a connection.

3. Message frame

The message frame is the basic unit of information that is transmitted over a session.

3.1. Message frame grammar

The message frame grammar is presented as follows in ABNF [1].

```

frame           = channel-id header payload
header         = message /

```

```

message-start-chunk /
message-chunk /
message-end-chunk /
error /
null

message = message-id
         content-description
message-start-chunk = message-start-chunk-id
                    content-description
message-chunk = message-chunk-id
message-end-chunk = message-end-chunk-id
error = error-id
null = null-id

content-description = content-id
                    number-of-parameters
                    *parameters
parameters = parameter-id
           parameter-value
payload = payload-length
        payload-octets

message-id = INTEGER4 (0)
message-start-chunk-id = INTEGER4 (1)
message-chunk-id = INTEGER4 (2)
message-end-chunk-id = INTEGER4 (3)
error-id = INTEGER4 (4)
null-id = INTEGER4 (5)

channel-id = INTEGER4
content-id = INTEGER4
number-of-parameters = INTEGER4
parameter-id = INTEGER4
parameter-value = STRING

payload-length = INTEGER8
payload-octets = *OCTET

```

3.1.1. The frame

The frame specifies the channel identifier, `channel-id`, of the message frame, the header (section [3.1.2](#)) and the payload (section [3.1.3](#)).

3.1.2. The header

The message frame grammar specifies six types of message frame.

NOTE – The relationship and constraints on sequence of one or more message frames that is a message is specified in [4.1](#).

If the message frame is `message` or `message-start-chunk` then a description of the content (section [3.1.4](#)), `content-description`, is required to be transmitted after the message identifier (`message-id`, `message-start-chunk-id`).

3.1.3. The payload

The payload of the message frame holds the all or some “chunk” of the application-defined data.

NOTE – If the message frame is part of a message then the payload of the message frame will hold a chunk of the application data.

The payload consists of the length in OCTETs of the application-defined data, `payload-length`, and the `payload-value`, which is the application-defined data and represented as a sequence of zero or more OCTETs. The `payload-length` MUST be equal to the number of OCTETs in the sequence.

3.1.4. The content-description

The description of content will contain a content identifier (section [3.1.4.1](#)), `content-id`, and a sequence of zero or more parameters (section [3.1.4.2](#)). The number of parameters, `number-of-parameters`, MUST be equal to to the number of the parameters in the sequence.

3.1.4.1. The content identifier

The content identifier, `content-id`, is an INTEGER4 value that identifies the content type of the payload of the message.

NOTE – The set of content type associated with a channel are specified by the Connection Management service (section [6](#)) when the channel is opened (section [8.1](#)).

3.1.4.2. The parameter

The parameter consists of a parameter identifier, `parameter-id`, and a value, `parameter-value` that is a STRING.

The `parameter-id` is an INTEGER4 value that identifies a parameter.

NOTE – The set of parameters associated with a channel are specified by the Connection Management service (section [6](#)) when the channel is opened (section [8.1](#)).

3.2. Message frame encoding

Message frames are encoded according to their types specified in the message frame grammar (section 3.1). Each value is encoded, according to its type, in order (as specified by the message frame grammar) and the bits produced by encoding a value are written to the Connection in that same order.

3.2.1. INTEGER4 type

The `INTEGER4` type is a non-negative integer (with no upper bound) whose value is encoded as a sequence of one or more `NIBBLE` values.

The non-negative integer is encoded in the least significant 3 bits of each `NIBBLE` as a sequence of 3-bit-nibbles, ordered from right to left with the least significant 3-bit-nibble encoded first. The sequence **MUST** be terminated with a (terminating) `NIBBLE` whose most significant bit is set to '0'. A non-terminating `NIBBLE` **MUST** have a most significant bit set to '1'. The non-negative integer **SHOULD** be encoded in the minimum number of required `NIBBLES`.

NOTE – See [Appendix C](#) for an example.

A rule using `INTEGER4` proceeding with text that is a non-negative integer (n say) surrounded in brackets specifies that the non-negative integer value of `INTEGER4` is n.

NOTE – An `INTEGER4 (2)` specifies an `INTEGER4` value that is a non-negative integer whose value is 2.

3.2.2. INTEGER8 type

The `INTEGER8` type is a non-negative integer (with no upper bound) whose value is encoded as a sequence of one or more `OCTET` values.

The non-negative integer is encoded in the least significant 7 bits of each `OCTET` as a sequence of 7-bit-octets, ordered from right to left with the least significant 7-bit-octet encoded first. The sequence **MUST** be terminated with a (terminating) `OCTET` whose most significant bit is set to '0'. A non-terminating `OCTET` **MUST** have a most significant bit set to '1'. The non-negative integer **SHOULD** be encoded in the minimum number of required `OCTETS`.

NOTE – See [Appendix C](#) for an example.

3.2.3. STRING type

The `STRING` type is a sequence of Unicode characters.

The number of `OCTET` values produced by the application of the UTF-8 encoding to the sequence of Unicode characters is encoded as an `INTEGER4` value, followed by those `OCTET` values (that is the UTF-8 encoding of the sequence of Unicode characters).

3.2.4. OCTET type

The `OCTET` type is a non-negative integer whose value is in the range 0 to 255 and is

encoded in a field of 8 bits.

Zero to seven 0 (padding) bits are encoded (if required) before the encoding of the 8 bits that is the OCTET value. The number of 0 (padding) bits, *p* say, to be encoded is function of the total number of bits previously encoded, *n* say, and is specified as follows

$$p = (8 - (n \% 8)) \% 8$$

NOTE – The encoding of padding bits ensures that an encoded OCTET is byte-aligned

3.2.5. NIBBLE type

The NIBBLE type is a non-negative integer whose value is in the range 0 to 15 and is encoded in a field of 4 bits.

Zero to three 0 (padding) bits are encoded (if required) before the encoding of the 4 bits that is the NIBBLE value. The number of 0 (padding) bits, *p* say, to be encoded is function of the total number of bits previously encoded, *n* say, and is specified as follows

$$p = (4 - (n \% 4)) \% 4$$

NOTE – The encoding of padding bits ensures that an encoded NIBBLE is byte-aligned

4. Message

The message is the basic unit of information transmitted over a channel.

4.1. Message representation

A message is an ordered sequence of one or more message frames that have the same channel identifier.

A message composed of one message frame **MUST** be a message frame of `message` (application message), `error` (error message) or `null` (null message).

A message (application message) composed of two or more message frames is an ordered sequence of message frames. The sequence **MUST** start with a message frame of `message-start-chunk`, **MUST** end with a message frame of `message-end-chunk`, and **MUST** contain (and only contain), in between the start and end of the sequence, zero or more message frames of `message-chunk`. All message frames **MUST** have the same channel identifier.

4.2. Exchanging of messages

Servers will receive request messages in the order they are sent by a client, and clients will receive response messages in the order they are sent by a server.

NOTE – The session and connection ensure message order. The session retains the order of message frames of a message. The (TCP/IP) connection retains the order of message frames.

A server **MUST** send a response message in response to receiving a request message from a client. The response message **MUST** be sent on the same channel as the request message.

NOTE – The correlation between a request and response is implicitly defined by the order of request and response messages.

Five request and response message exchanges are specified in [Table 1](#).

<i>Request</i>	<i>Response</i>
Application message	Application message
Application message	Null message
Null message	Null message
Application message	Error message
Null message	Error message

Table 1: Request/response exchanges

5. Message errors

5.1. Malformed message frames

A client or server MUST close the connection if a malformed message frame is detected. The server on detecting a malformed message frame sent by the client MAY respond with an error message before closing the connection (section [5.4.2](#)).

The following malformed message frames are detectable:

- a `message-id` greater than 5;
- an incorrect message frame sequence that does not conform to the sequence specified in section [4.1](#); and
- an incorrect request/response exchange that does not conform to the exchanges specified in section [4.2](#).

5.2. Message frames with channel errors

A client or server MUST ignore a message frame transmitted on a channel and not transmit further application messages on that channel if a message frame with a channel error is detected on that channel.

A client or server MAY transmit application messages on the channel if the channel error is resolved by transmission of message frames on a different channel or by out-of-band mechanisms not specified by this document.

The server on detecting a message frame with a channel error MUST respond with an error message (section [5.4.3](#)) on that channel.

The following are the set of possible channel errors:

- a general channel error;
 - NOTE – Such an error can occur if the channel is known and a message frame on that channel is valid but the message could not be processed by the client or server.
- an unknown channel identifier;
- an unknown content identifier; and

- an unknown parameter identifier.

5.3. Denial of service attacks

A client or server **MUST** close the connection if a denial of service attack is detected.

The following denial of service attacks are possible:

- An INTEGER4 or INTEGER8 value that is too large; and
- A message composed of too many message frames.

This document does not specify the constraints, for denial of service detection, on the size of INTEGER4 or INTEGER8 values or the number of message frames.

5.4. Error messages

An error message is sent (by a server) to in response to the detection of a malformed message frame (sent by the client) or a message frame with a channel error (sent by the client).

The payload grammar of the error message is specified in section [5.4.1](#).

Error message payloads are specified in sections [5.4.2](#) and [5.4.3](#) for malformed message frames and message frames with channel errors respectively.

5.4.1. Payload of the error message

The payload grammar of the error message is presented as follows in ABNF [1]:

```

error-message-payload = code sub-code description
code                  = INTEGER4
sub-code              = INTEGER4
description           = STRING

```

5.4.2. The error message payload for malformed message frame errors

The code of the error message payload is 0.

[Table 2](#) specifies the error payload with the sub-code in column 1, a non-normative description in column 2 and an detailed description of the error in column 3.

sub-code	description	comment
0	Malformed frame	A malformed frame is detected but the precise cause is unknown.
1	Unknown message identifier	The message-id is not recognized as part of the message frame grammar (section 3.1).
2	Incorrect message frame sequence	A message consisting of two or more message frames does not contain the correct sequence of message identifiers (section 4.1).

sub-code	description	comment
3	Interleaved message frame sequence	A message consisting of two or more message frames has a two or more frames with different channel identifies (section 4.1).
4	Unknown request/response pattern	The request/response pattern is not recognized as a valid request/response pattern (section 4.2)

Table 2: Malformed message frame sub-code errors

5.4.3. The error message payload for message frames with channel errors

The code of the error message payload is 1.

[Table 3](#) specifies the error payload with the sub-code in column 1, a non-normative description in column 2 and an detailed description of the error in column 3.

sub-code	description	comment
0	General channel error	A channel specific error has been detected but the precise cause is unknown.
1	Unknown channel identifier	The channel identifier is not recognized.
2	Unknown content identifier	The content identifier is not recognized.
3	Unknown parameter identifier	The parameter identifier is not recognized.

Table 3: Message frame with channel error sub-code errors

6. Connection Management service

The Connection Management service is a SOAP-based service, described by WSDL (see [Appendix A](#)), and operates on the service channel.

The service channel **MUST NOT** be used to transmit messages other than those specified by the operations of the Connection Management service.

The operations of the Connection Management service are described in the [table 4](#).

<i>Operations</i>	<i>Description</i>
initiateSession	Initiates the session (section 7)
openChannel	Requests the opening of new channel (section 8.1).
closeChannel	Requests the of closing of an opened channel (section 8.2).

Table 4: Connection Management service operations

6.1. Connection Management service errors

The Connection management service may return the SOAP fault, the ServiceChannelException (see [Appendix A](#)), if an operation failed.

The SOAP fault error codes for the initiateSession operation are specified in [table 5](#).

<i>Error code</i>	<i>Error description</i>
TOO_MANY_OPEN_SESSIONS	The server has reached the limit on the total number of open TCP connections..

Table 5: Connection Management service error codes for the *initiateSession* operation

If the server returns a SOAP fault response then the client **MUST** either close the connection or try resending another request for the *initiateSession* operation.

The SOAP fault error codes for the *openChannel* operation are specified in [table 6](#).

<i>Error code</i>	<i>Error description</i>
TOO_MANY_OPEN_CHANNELS_FOR_SESSION	The Server has reached the limit on the total number of open channels for a session.
UNKNOWN_ENDPOINT_ADDRESS	The Server cannot find an appropriate Web service that corresponds to the requested endpoint address.
CONTENT_NEGOTIATION_FAILED	The client and server cannot agree on a common set of content types.

Table 6: Connection Management service error codes for the *openChannel* operation

The SOAP fault error codes for the *closeChannel* operation are specified in [table 7](#).

<i>Error code</i>	<i>Error description</i>
UNKNOWN_CHANNEL_ID	The Server cannot find an appropriate channel and the context, which should correspond to the requested channel-id.

Table 7: Connection Management service error codes for the *closeChannel* operation

7. Initiating a session

A client initiates a session by performing the following actions (in order):

- open a TCP connection;
- send the protocol magic identifier (section [7.1](#))
- negotiate the version of the messaging framing and Connection Management service (section [7.2](#)); and
- invoke the *initiateSession* operation on the Connection Management service (section [6](#)).

Before the *initiateSession* operation is invoked the client and service **MUST** implicitly open a channel, the service channel, with the channel identifier 0 (section [8.3](#)).

7.1. Protocol magic identifier

After establishing a TCP connection the client **MUST** send the protocol magic identifier. The identifier **MUST** be the US-ACIII encoding of the character sequence “vnd.sun.ws.tcp”.

If the identifier is not recognized by the server then the server **MUST** close the

connection.

7.2. Protocol version

After sending the protocol magic identifier the client MUST send the highest versions of the supported message framing and Connection Management service.

If the server supports the client's versions it MUST respond with the same versions as those that the client sent. If the server does not support the client's versions it MUST reply with closest versions to client's versions it supports and close the TCP connection.

A version consists of 2 parts: major; and minor.

NOTE – Version “1.0” will will have a major part of 1 and a minor part of 0.

The protocol version is encoded as specified in section [7.2.1](#).

This document specifies the message framing (section [3.1](#)) version to be 1.0 and the Connection Management service (section [6](#)) version to be 1.0.

7.2.1. Protocol version grammar

The protocol version grammar is presented as follows in ABNF [1]:

```

protocol-version           = message-frame-version
                           connection-management-version
message-frame-version      = major minor
connection-management-version = major minor
major                      = INTEGER4
minor                      = INTEGER4

```

8. Channels

A channel represents a “virtual connection” over a TCP connection.

8.1. Opening a channel

After a session is initiated a client can open a channel using the openChannel operation of the Connection Management service. The openChannel operation requires that the client and server negotiate the types of content that will be transmitted (section [8.1.1](#)). In addition it requires that the server respond with a channel identifier the client shall use to send requests on that channel.

8.1.1. Content negotiation

A client when sending a request to open a channel MUST send the sequence of content types and parameters it supports.

The server when responding to an open channel request MUST reply with a sub-set of client supported sequence of content types and parameters it supports.

If the server does not support any of the content types sent by the server then it MUST

reply with a SOAP fault as specified in [table 6](#).

The client and server MUST assign content identifiers to the content types declared in the sequence of content types returned by the server. The content identifier associated with a content type MUST be equivalent to the position of the content type in the sequence, where the first content type in the sequence has a position of 0.

The client and server MUST assign parameter identifiers to the parameters declared in the sequence of parameters returned by the server. The parameter identifier associated with a parameter MUST be equivalent to the position of the parameter in the sequence, where the first parameter in the sequence has a position of 0.

8.2. Closing a channel

A client may close a channel using the `closeChannel` operation of the Connection Management service. After a channel is closed a client and server may release all resources associated with that channel.

8.3. The service channel

The service channel specifies predefined content identifiers and parameters identifiers that are supported by the client and server transmitting messages on the service channel.

[Table 8](#) specifies the content identifiers and [Table 9](#) specifies the parameter identifiers.

<i>Mime type</i>	<i>Content-id</i>	<i>Description</i>
text/xml	0	SOAP 1.1
application/fastinfoset	1	Fast Infoset for SOAP 1.1

Table 8: Predefined content identifier values

<i>Mime parameter name</i>	<i>Parameter-id</i>	<i>Description</i>
charset	0	Charset parameter
SOAPAction	1	SOAP action parameter

Table 9: Predefined parameter identifier values

9. Stated-based Fast Infoset

A client and server MAY, when opening a channel, support state-based Fast Infoset for the encoding/decoding of messages.

The state-based Fast Infoset encoding/decoding is an extension to the Fast Infoset encoding/decoding where the final vocabulary obtained from the previous encoding/decoding of a message is utilized to encode/decode the current message.

NOTE – When a channel is used for a SOAP-based Web service it is likely that similar XML infoset, element/attribute names and namespace names, will be repeatedly sent for each message. The application of state-based Fast Infoset reduces the size of messages and increases the

processing efficiency because the redundancy of common repeating information over an exchange of messages is reduced.

The MIME media type for state-based Fast Infoset MUST be either “application/vnd.sun.stateful.fastinfoset” for Fast Infoset documents or SOAP 1.1 messages that are Fast Infoset documents, and “application/vnd.sun.stateful.soap+fastinfoset” for SOAP 1.2 messages that are Fast Infoset documents.

After opening a state-based Fast Infoset capable channel the client and server initialize state such that each maintains two Fast Infoset vocabularies: the state-based encoding vocabulary for encoding; and the state-based decoding vocabulary for decoding. Initially both vocabularies MUST be empty. The vocabularies MUST be retained for the life-time of the channel.

A client or server encoding a message using Fast Infoset MUST use the state-based encoding vocabulary in an equivalent manner to it being an external vocabulary, except that the external vocabulary URI of the Fast Infoset document (that is the encoded message) MUST not be encoded. After encoding, the state-based encoding vocabulary MUST be set to the final vocabulary produced from the Fast Infoset encoding process.

A client or server decoding an encoded message using Fast Infoset MUST use the state-based decoding vocabulary in an equivalent manner to it being the external vocabulary of the Fast Infoset document (that is the encoded message). After decoding, the state-based decoding vocabulary MUST be set to the final vocabulary produced from the Fast Infoset decoding process.

NOTE – The state-based encoding and decoding vocabularies on the client and server will be consistent because messages are received in the order they are sent.

Appendix A. Connection Management web service description (WSDL)

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://servicechannel.tcp.transport.ws.xml.sun.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://servicechannel.tcp.transport.ws.xml.sun.com/"
  name="ServiceChannelWSImplService">
  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://servicechannel.tcp.transport.ws.xml.sun.com/"
        schemaLocation="ServiceChannelWSImplService_schema1.xsd"/>
      </xsd:schema>
    </types>
    <message name="initiateSession">
      <part name="parameters" element="tns:initiateSession"/>
    </message>
    <message name="initiateSessionResponse">
      <part name="parameters" element="tns:initiateSessionResponse"/>
    </message>
    <message name="ServiceChannelException">
      <part name="fault" element="tns:ServiceChannelException"/>
    </message>
    <message name="openChannel">
      <part name="parameters" element="tns:openChannel"/>
    </message>
    <message name="openChannelResponse">
      <part name="parameters" element="tns:openChannelResponse"/>
    </message>
    <message name="closeChannel">
      <part name="parameters" element="tns:closeChannel"/>
    </message>
    <message name="closeChannelResponse">
      <part name="parameters" element="tns:closeChannelResponse"/>
    </message>
    <portType name="ServiceChannelWSImpl">
      <operation name="initiateSession">
        <input message="tns:initiateSession"/>
        <output message="tns:initiateSessionResponse"/>
        <fault message="tns:ServiceChannelException"
          name="ServiceChannelException"/>
      </operation>
      <operation name="openChannel">
        <input message="tns:openChannel"/>
        <output message="tns:openChannelResponse"/>
        <fault message="tns:ServiceChannelException"
          name="ServiceChannelException"/>
      </operation>
      <operation name="closeChannel">
        <input message="tns:closeChannel"/>
        <output message="tns:closeChannelResponse"/>
        <fault message="tns:ServiceChannelException"
          name="ServiceChannelException"/>
      </operation>
    </portType>
    <binding name="ServiceChannelWSImplPortBinding" type="tns:ServiceChannelWSImpl">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
      <operation name="initiateSession">
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
  </definitions>

```

```
</output>
<fault name="ServiceChannelException">
  <soap:fault name="ServiceChannelException" use="literal"/>
</fault>
</operation>
<operation name="openChannel">
  <soap:operation soapAction=""/>
  <input>
    <soap:body use="literal"/>
  </input>
  <output>
    <soap:body use="literal"/>
  </output>
  <fault name="ServiceChannelException">
    <soap:fault name="ServiceChannelException" use="literal"/>
  </fault>
</operation>
<operation name="closeChannel">
  <soap:operation soapAction=""/>
  <input>
    <soap:body use="literal"/>
  </input>
  <output>
    <soap:body use="literal"/>
  </output>
  <fault name="ServiceChannelException">
    <soap:fault name="ServiceChannelException" use="literal"/>
  </fault>
</operation>
</binding>
<service name="ServiceChannelWSImplService">
  <port name="ServiceChannelWSImplPort"
    binding="tns:ServiceChannelWSImplPortBinding">
    <soap:address location="vnd.sun.ws.tcp://CHANGED_BY_RUNTIME"/>
  </port>
</service>
</definitions>
```

Appendix B. Connection Management web service type definitions (XSD)

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://servicechannel.tcp.transport.ws.xml.sun.com/"
  version="1.0"
  targetNamespace="http://servicechannel.tcp.transport.ws.xml.sun.com/">
  <xs:element name="ServiceChannelException"
    nillable="true"
    type="tns:serviceChannelExceptionBean"/>
  <xs:element name="closeChannel"
    type="tns:closeChannel"/>
  <xs:element name="closeChannelResponse"
    type="tns:closeChannelResponse"/>
  <xs:element name="initiateSession"
    type="tns:initiateSession"/>
  <xs:element name="initiateSessionResponse"
    type="tns:initiateSessionResponse"/>
  <xs:element name="openChannel"
    type="tns:openChannel"/>
  <xs:element name="openChannelResponse"
    type="tns:openChannelResponse"/>
  <xs:complexType name="initiateSession">
    <xs:sequence/>
  </xs:complexType>
  <xs:complexType name="initiateSessionResponse">
    <xs:sequence/>
  </xs:complexType>
  <xs:complexType name="serviceChannelExceptionBean">
    <xs:sequence>
      <xs:element name="errorCode" type="tns:serviceChannelErrorCode"/>
      <xs:element name="message" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="closeChannel">
    <xs:sequence>
      <xs:element name="channelId" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="closeChannelResponse">
    <xs:sequence/>
  </xs:complexType>
  <xs:complexType name="openChannel">
    <xs:sequence>
      <xs:element name="targetWSURI"
        type="xs:string"/>
      <xs:element name="negotiatedMimeTypes"
        type="xs:string" maxOccurs="unbounded"/>
      <xs:element name="negotiatedParams"
        type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="openChannelResponse">
    <xs:sequence>
      <xs:element name="channelId" type="xs:int"/>
      <xs:element name="negotiatedMimeTypes"
        type="xs:string" maxOccurs="unbounded"/>
      <xs:element name="negotiatedParams"
        type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="serviceChannelErrorCode">
    <xs:restriction base="xs:string">
      <xs:enumeration value="TOO_MANY_OPEN_SESSIONS"/>
      <xs:enumeration value="TOO_MANY_OPEN_CHANNELS_FOR_SESSION"/>
      <xs:enumeration value="UNKNOWN_ENDPOINT_ADDRESS"/>
      <xs:enumeration value="CONTENT_NEGOTIATION_FAILED"/>
      <xs:enumeration value="UNKNOWN_CHANNEL_ID"/>
    </xs:restriction>
  </xs:simpleType>

```

```
    </xs:restriction>  
  </xs:simpleType>  
</xs:schema>
```

Appendix C. Data types encoding examples

I. INTEGER4

Integer value: 7554(10) = **1**110**1**10**0**00**0**1**0**(2)

Sequence of NIBBLES: **1**0**1**0**1**0**0**0**1**1**1**0**1**110**0**00**1**(2)

(The most significant bit of each NIBBLE is highlighted in bold.)

II. INTEGER8

Integer value: 7554(10) = **1**11**0**1**1**00000**1**0(2)

Sequence of OCTETs: **1**100000**1**0**0**000**1**1**1**0**1**(2)

(The most significant bit of each OCTET is highlighted in bold.)

Appendix D. References

- [1] [Crocker, D.](#) and [P. Overell](#), "[Augmented BNF for Syntax Specifications: ABNF](#)", RFC 2234, November 1997.
- [2] Generic Applications of ASN.1, Fast Infoset, [ITU-T Rec. X.891 \(2005\)](#) | [ISO/IEC 24824-1](#).