Wireless Session Protocol 1.0

Candidate Version 1.0 – 20 Sept 2002

Open Mobile Alliance

OMA-WAP-TS-WSP-V1_0-20020920-C

Continues the Technical Activities
Originated in the WAP Forum

Contents

1 Scope

The Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope for the WAP Forum is to define a set of specifications to be used by service applications. The wireless market is growing very quickly, and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and fast/flexible service creation WAP Forum defines a set of protocols in transport, security, transaction, session and application layers. For additional information on the WAP architecture, please refer to "*Wireless Application Protocol Architecture Specification"* [ARCH].

The Session layer protocol family in the WAP architecture is called the Wireless Session Protocol, WSP. WSP provides the upper-level application layer of WAP with a consistent interface for two session services. The first is a connection-mode service that operates above a transaction layer protocol WTP, and the second is a connectionless service that operates above a secure or non-secure datagram transport service. For more information on the transaction and transport services, please refer to "*Wireless Application Protocol: Wireless Transaction Protocol Specification*" [WTP] and "*Wireless Application Protocol: Wireless Datagram Protocol Specification*" [WDP].

The Wireless Session Protocols currently offer services most suited for browsing applications (WSP). WSP provides HTTP 1.1 functionality and incorporates new features such as long-lived sessions, a common facility for data push, capability negotiation and session suspend/resume. The protocols in the WSP family are optimised for low-bandwidth bearer networks with relatively long latency.

2 Document Status

For information and comments on this specification, please visit http://www.openmobilealliance.org/.

3 References

3.1 Normative References

[ARCH] "WAP Architecture Specification, WAP Forum, 30-April-1998.
URL: http://www.openmobilealliance.org

[IOPProc] "OMA Interoperability Policy and Process". Open Mobile Alliance™. OMA-IOP-Process-v1_0.
URL:http//www.openmobilealliance.org/

[PROVCONT] "WAP Provisioning Content ", WAP Forum, 19-February-2000,
URL: http://www.openmobilealliance.org

[WCMP] "Wireless Control Message Protocol", Open Mobile Alliance™, WAP-202-WCMP,
http://www.openmobilealliance.org/.

[WDP] "Wireless Datagram Protocol", Open Mobile Alliance™, WAP-259-WDP,
http://www.openmobilealliance.org/.

[WTP] "Wireless Transaction Protocol", Open Mobile Alliance™, WAP-224-WTP,
http://www.openmobilealliance.org/.

[UAPROF] "User Agent ProfileUAPROF", Open Mobile Alliance™, OMA-UAProf-v2_0
URL: http://www.openmobilealliance.org

[UACACHE] "WAP Caching Model Specification", WAP Forum, 11-February-1999.
URL: http://www.openmobilealliance.org

[WBXML] "WAP Binary XML Content Format", WAP Forum, 4-November-1999.
URL: http://www.openmobilealliance.org

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", Bradner, S.,
March 1997, URL: ftp://ftp.isi.edu/in-notes/rfc2119.txt.

[RFC2616] "Hypertext Transfer Protocol -- HTTP/1.1", Fielding, R., et. al., June 1999,URL: ftp://ftp.isi.edu/in-notes/rfc2616.txt.

[RFC2045] "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies",
Borenstein, N., et. al., November 1996, URL: ftp://ftp.isi.edu/in-notes/rfc2045.txt.

[RFC2047] "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", Moore, K., November 1996, URL: ftp://ftp.isi.edu/in-notes/rfc2047.txt.

[RFC2145] "Use and Interpretation pf HTTP Version Numbers", Mogul J.C. & al, May 1997, URL:
ftp://ftp.isi.edu/in-notes/rfc2145.txt

[RFC822] "Standard for The Format of ARPA Internet Text Messages", Crocker, D.,
August 1982, URL: ftp://ftp.isi.edu/in-notes/rfc822.txt.

[TLE2E] "WAP Transport Layer E2E Security", Open Mobile Alliance™,
OMA-WAP-TransportE2ESec-v1_0, URL: http://www.openmobilealliance.org/

[RFC2183] "Communicating Presentation Information in Internet Messages: The Content-Disposition Header field",
R. Troost, S. Dorner, K. Moore, August 1997, URL:ftp://ftp.isi.edu/in-notes/rfc2183.txt

[RFC2388] "Returning Values from Forms: multipart/form-data", L. Masinter, August 1998,
URL:ftp://ftp.isi.edu/in-notes/rfc2388.txt

3.2 Informative References

[ISO7498] "Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model", ISO/IEC 7498-1:1994.

[ISO10731] "Information Technology - Open Systems Interconnection - Basic Reference Model - Conventions for the Definition of OSI Services", ISO/IEC 10731:1994.

[RFC1630] "Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", Berners-Lee, T., June 1994, URL:ftp://ftp.isi.edu/in-notes/rfc1630.txt.

[RFC1738] "Uniform Resource Locators (URL)", Berners-Lee, T., et. al., December 1994, URL:ftp://ftp.isi.edu/in-notes/rfc1738.txt.

[RFC1808] "Relative Uniform Resource Locators", Fielding, R., June 1995, URL:ftp://ftp.isi.edu/in-notes/rfc1808.txt.

[RFC1864] "The Content-MD5 Header Field", Meyers, J. and Rose, M., October 1995, URL:ftp://ftp.isi.edu/in-notes/rfc1864.txt.

[CCPPEx] "Composite Capability / Preferences Profiles: A user side framework for Content Negotiation," W3C Note, 27-July, 1999.
URL: http://www.w3.org/TR/1999/NOTE-CCPP-19990727

[MEDIATYPE] "XML Media Types", M. Murata & al, December 1999, URL:
http://www.simonstl.com/idxmlmime/draft-murata-xml-02.htm

4 Definitions and Abbreviations

4.1 Definitions

For the purposes of this specification the following definitions apply.

Bearer Network

A bearer network is used to carry the messages of a transport-layer protocol - and ultimately also of the session layer protocols - between physical devices. During the lifetime of a session, several bearer networks may be used.

Capability

Capability is a term introduced in section 6.3.2, "Capabilities", to refer to the session layer protocol facilities and configuration parameters that a client or server supports.

Capability Negotiation

Capability negotiation is the mechanism defined in section 6.3.2.1, "Capability Negotiation", for agreeing on session functionality and protocol options. Session capabilities are negotiated during session establishment. Capability negotiation allows a server application to determine whether a client can support certain protocol facilities and configurations.

Client and Server

The term client and server are used in order to map WSP to well known and existing systems. A client is a device (or application) which initiates a request for a session. The server is a device that passively waits for session requests from client devices. The server can either accept the request or reject it.

An implementation of the WSP protocol may include only client or server functions in order to minimise the footprint. A client or server may only support a subset of the protocol facilities, indicating this during protocol capability negotiation.

Connectionless Session Service

Connectionless session service (section 6.4) is an unreliable session service. In this mode, only the request primitive is available to service users, and only the indication primitive is available to the service provider.

Connection-Mode Session Service

Connection-mode session service (section 6.3) is a reliable session service. In this mode, both request and response primitives are available to service users, and both indication and confirm primitives are available to the service provider.

Content

The entity body sent with a request or response is referred to as content. It is encoded in a format and encoding defined by the entity-header fields.

Content Negotiation

Content negotiation is the mechanism the server uses to select the appropriate type and encoding of content when servicing a request. The type and encoding of content in any response can be negotiated. Content negotiation allows a server application to decide whether a client can support a certain form of content.

Default Session Context
The assumed session context based on WAP class conformance or implementation specific mechanisms.

Entity
An entity is the information transferred as the payload of a request or response. An entity consists of meta-information in the form of entity-header fields and content in the form of an entity-body.

Header
A header contains meta-information. Specifically, a session header contains general information about a session that remains constant over the lifetime of a session; an entity-header contains meta-information about a particular request, response or entity body (content).

Layer Entity
In the OSI architecture, the active elements within a layer that participate in providing layer service are called layer entities.

Method
Method is the *type* of client request as defined by HTTP/1.1 (eg, Get, Post, etc.). A WSP client uses methods and extended methods to invoke services on the server.

Null terminated string
A sequence of non-zero octets followed by a zero octet.

Peer Address Quadruplet
Sessions are associated with a particular client address, client port, server address and server port. This combination of four values is called the peer address quadruplet in the specification.

Proxy
An intermediary program that acts both as a server and a client for the purpose of making request on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other server.

Pull and Push Data Transfer
Push and pull are common vernacular in the Internet world to describe push transactions and method transactions respectively. A server "pushes" data to a client by invoking the WSP push service, whereas a client "pulls" data from a server by invoking the WSP method service.

Session
A long-lived communication context established between two programs for the purpose of transactions and typed data transfer.

Session Context
The negotiated capabilities, transaction state and client/server addressing data as passed during session establishment.

Session Service Access Point (S-SAP)
Session Service Access Point is a conceptual point at which session service is provided to the upper layer.

Session Service Provider
A Session Service Provider is a layer entity that actively participates in providing the session service via an S-SAP.

Session Service User
A Session Service User is a layer entity that requests services from a Session Service Provider via an S-SAP.

Transaction

Two forms of transactions are specified herein. We do not use the term transaction to imply the semantics often associated with database transactions.

- A *method transaction* is a three-way request-response-acknowledge communication initiated by the client to invoke a method on the server.
- A *push transaction* is a two-way request-acknowledge communication initiated by the server to push data to the client.

4.2 Abbreviations

For the purposes of this specification the following abbreviations apply.

API	Application Programming Interface
A-SAP	Application Service Access Point
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
MOM	Maximum Outstanding Method requests
MOP	Maximum Outstanding Push requests
MRU	Maximum Receive Unit
OSI	Open System Interconnection
PDU	Protocol Data Unit
S-SAP	Session Service Access Point
SDU	Service Data Unit
SEC-SAP	Security Service Access Point
T-SAP	Transport Service Access Point
TID	Transaction Identifier
TR-SAP	Transaction Service Access Point
WDP	Wireless Datagram Protocol
WSP	Wireless Session Protocol
WTP	Wireless Transaction Protocol

4.3 Documentation Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119]. The following sections and appendices are normative:

Sections 3, 6, 7, 8
Appendix A, D

All other sections and appendices are informative.

5 WSP Architectural Overview

Wireless Session Protocol is a session-level protocol family for remote operations between a client and proxy or server.

5.1 Reference Model

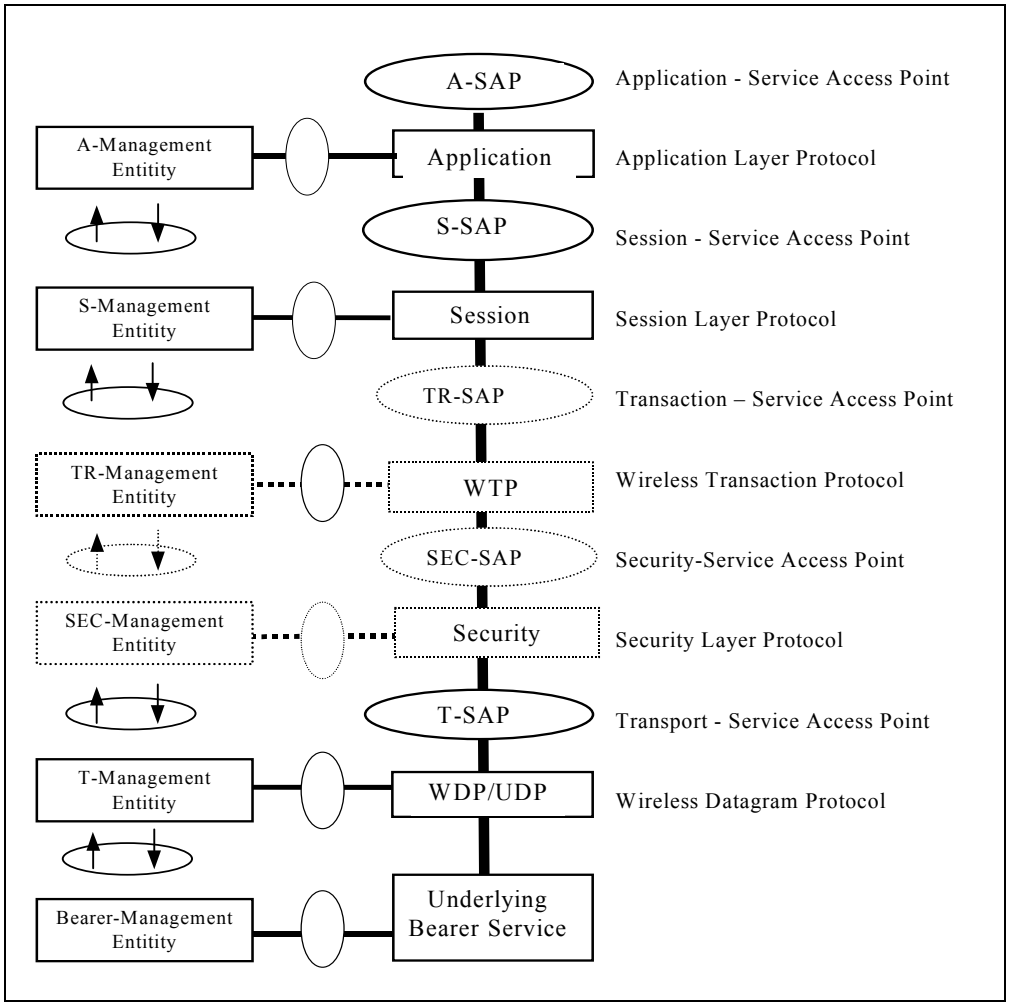

Figure 1. Wireless Application Protocol Reference Model

A model of layering the protocols in WAP is illustrated in Figure 1. WAP protocols and their functions are layered in a style resembling that of the ISO OSI Reference Model [ISO7498]. Layer Management Entities handle protocol initialisation, configuration and error conditions (such as loss of connectivity due to the mobile station roaming out of coverage) that are not handled by the protocol itself.

WSP is designed to function on the transaction and datagram services. Security is assumed to be an optional layer above the transport layer. The security layer preserves the transport service interfaces. The transaction, session or application management entities are assumed to provide the additional support that is required to establish security contexts and secure connections. This support is not provided by the WSP protocols directly. In this regard, the security layer is modular. WSP itself does not require a security layer; however, applications that use WSP may require it.

5.2 WSP Features

WSP provides a means for organised exchange of content between co-operating client/server applications. Specifically, it provides the applications means to:

a) establish a reliable session from client to server and release that session in an orderly manner;
b) agree on a common level of protocol functionality using capability negotiation;
c) exchange content between client and server using compact encoding;
d) suspend and resume the session.

The currently defined services and protocols (WSP) are most suited for browsing-type applications. WSP defines actually two protocols: one provides connection-mode session services over a transaction service, and another provides non-confirmed, connectionless services over a datagram transport service. The connectionless service is most suitable, when applications do not need reliable delivery of data and do not care about confirmation. It can be used without actually having to establish a session.

In addition to the general features, WSP offers means to:

a) provide HTTP/1.1 functionality:
   1) extensible request-reply methods,
   2) composite objects,
   3) content type negotiation;
b) exchange client and server session headers;
c) interrupt transactions in process;
d) push content from server to client in an unsynchronised manner;
e) negotiate support for multiple, simultaneous asynchronous transactions.

5.2.1 Basic Functionality

The core of the WSP design is a binary form of HTTP. Consequently the requests sent to a server and responses going to a client may include both headers (meta-information) and data. All the methods defined by HTTP/1.1 are supported. In addition, capability negotiation can be used to agree on a set of extended request methods, so that full compatibility to HTTP/1.1 applications can be retained.

WSP provides typed data transfer for the application layer. The HTTP/1.1 content headers are used to define content type, character set encoding, languages, etc, in an extensible manner. However, compact binary encodings are defined for the well-known headers to reduce protocol overhead. WSP also specifies a compact composite data format that provides content headers for each component within the composite data object. This is a semantically equivalent binary form of the MIME "multipart/mixed" format used by HTTP/1.1.

WSP itself does not interpret the header information in requests and replies. As part of the session creation process, request and reply headers that remain constant over the life of the session can be exchanged between service users in the client and the server. These may include acceptable content types, character sets, languages, device capabilities and other static parameters. WSP will pass through client and server session headers as well as request and response headers without additions or removals.

The lifecycle of a WSP session is not tied to the underlying transport. A session can be suspended while the session is idle to free up network resources or save battery. A lightweight session re-establishment protocol allows the session to be resumed without the overhead of full-blown session establishment. A session may be resumed over a different bearer network.

5.2.2 Extended Functionality

WSP allows extended capabilities to be negotiated between the peers. This allows for both high-performance, feature-full implementation as well as simple, basic and small implementations.

WSP provides an optional mechanism for attaching header information (meta-data) to the acknowledgement of a transaction. This allows the client application to communicate specific information about the completed transaction back to the server.

WSP provides both push and pull data transfer. Pull is done using the request/response mechanism from HTTP/1.1. In addition, WSP provides three push mechanisms for data transfer:

- Confirmed data push within an existing session context
- Non-confirmed data push within an existing session context
- Non-confirmed data push without an existing session

The confirmed data push mechanism allows the server to push data to the client at any time during a session. The server receives confirmation that the push was delivered.

The non-confirmed data push within an existing session provides a similar function as reliable data push, but without confirmation. The non-confirmed data push can also be achieved without an existing session. In this case, a default session context is assumed. Non-confirmed out-of-session data push can be used to send one-way messages over an unreliable transport.

WSP optionally supports asynchronous requests, so that a client can submit multiple requests to the server simultaneously. This improves utilisation of airtime in that multiple requests and replies can be coalesced into fewer messages. This also improves latency as the results of each request can be sent to the client when it becomes available.

WSP partitions the space of well-known header field names into *header code pages.* Each code page can define only a fairly limited number of encodings for well-known field names, which permits them to be represented more compactly. Running out of identities for well-known field names on a certain code page is still not a problem, since WSP specifies a mechanism for shifting from one header code page to another.

6 WSP Elements of Layer-to-Layer Communication

The session layer in WAP provides both connection-mode and connectionless services. They are defined using an abstract description technique based on service primitives, which is borrowed from [ISO10731]. Some of the terms and concepts used to describe the communication mechanisms are borrowed from [ISO7498], whereas the terminology used for operations and the manipulated data objects is based on [RFC2616].

This service definition specifies the minimum functionality that the WAP session layer must be able to provide to support its users. Since this definition is abstract, it does not specify or constrain programming interfaces or implementations. In fact the same service could be delivered by different protocols.

6.1 Notations Used

6.1.1 Definition of Service Primitives and Parameters

Communications between layers and between entities within the session layer are accomplished by means of service primitives. Service primitives represent, in an abstract way, the logical exchange of information and control between the session layer and adjacent layers.

Service primitives consist of commands and their respective responses associated with the particular service provided. The general syntax of a primitive is:

X-Service.type (Parameters)

where *X* designates the layer providing the service. For this specification *X* is "S" for the Session Layer.

Service primitives are not the same as an application-programming interface (API) and are not meant to imply any specific method of implementing an API. Service primitives are an abstract means of illustrating the services provided by the protocol layer to the layer above. In particular, the service primitives and their parameters are not intended to include the information that an implementation might need to route the primitives to each implementation object, which corresponds to some abstract user or service provider entity instance. The mapping of these concepts to a real API and the semantics associated with a real API is an implementation issue and beyond the scope of this specification.

6.1.2 Time Sequence Charts

The behaviour of service primitives is illustrated using time sequence charts, which are described in [ISO10731].

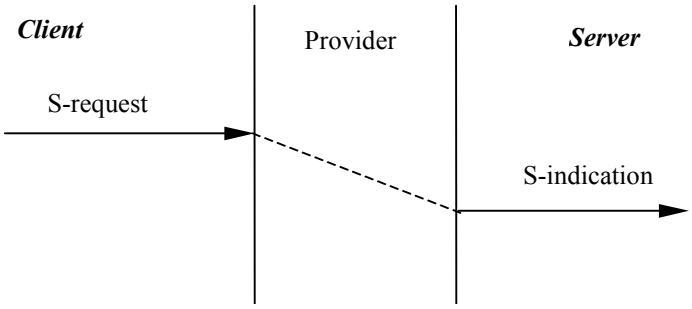

Figure 2: A Non-confirmed Service

Figure 2 illustrates a simple non-confirmed service, which is invoked using a request primitive and results in an indication primitive in the peer. The dashed line represents propagation through the provider over a period of time indicated by the vertical difference between the two arrows representing the primitives. If the labels ***Client*** and ***Server*** are included in the

diagram, this indicates that both peers cannot originate a primitive; if the labels are omitted, either peer can originate the primitive.

6.1.3 Primitives Types

The primitives types defined in this specification are

Type	Abbreviation	Description
Request	req	Used when a higher layer is requesting a service from the next lower layer
Indication	ind	A layer providing a service uses this primitive type to notify the next higher layer of activities related to the peer (such as the invocation of the request primitive) or to the provider of the service (such as a protocol generated event)
Response	res	A layer uses the response primitive type to acknowledge receipt of the indication primitive type from the next lower layer
Confirm	cnf	The layer providing the requested service uses the confirm primitive type to report that the activity has been completed successfully

6.1.4 Primitive Parameter Tables

The service primitives are defined using tables indicating which parameters are possible and how they are used with the different primitive types. If some primitive type is not possible, the column for it will be omitted.

The entries used in the primitive type columns are defined in the following table:

Table 1. Parameter Usage Legend

M	Presence of the parameter is mandatory - it MUST be present
C	Presence of the parameter is conditional depending on values of other parameters
O	Presence of the parameter is a user option - it MAY be omitted
P	Presence of the parameter is a service provider option - an implementation MAY not provide it
–	The parameter is absent
*	Presence of the parameter is determined by the lower layer protocol
(=)	When this primitive is generated as a result of invoking the preceding primitive by the peer service user, the value of the parameter shall be identical to the value of the corresponding parameter in that primitive. Otherwise the service provider selects an appropriate value.

For example, a simple confirmed primitive might be defined using the following:

Primitive / Parameter	S-PrimitiveX			
	req	***Ind***	***res***	***cnf***
Parameter 1	M	M(=)	–	–
Parameter 2	–	–	O	C(=)

In the example definition above, *Parameter 1* is always present in *S-PrimitiveX.request* and corresponding *S-PrimitiveX.indication*. *Parameter 2* MAY be specified in *S-PrimitiveX.response* and in that case it MUST be present and have the equivalent value also in the corresponding *S-PrimitiveX.confirm*; otherwise, it MUST NOT be present.

An example of a simpler primitive is:

Primitive / Parameter	S-PrimitiveY	
	req	***ind***
Parameter 2	–	M

In this example, *S-PrimitiveY.request* has no parameters, but the corresponding *S-PrimitiveX.indication* MUST always have *Parameter 2*. *S-PrimitiveX.response* and *S-PrimitiveX.confirm* are not defined and so can never occur.

6.2 Service Parameter Types

This section describes the types of the abstract parameters used subsequently in the service primitive definition. The actual format and encoding of these types is an implementation issue not addressed by this service definition.

In the primitive descriptions the types are used in the names of parameters, and they often have an additional qualifier indicating where or how the parameter is being used. For example, parameter *Push Body* is of the type *Body*, and parameter *Client Address* of type *Address*.

6.2.1 Address

The session layer uses directly the addressing scheme of the layer below. *Server Address* and *Client Address* together form the peer address quadruplet, which identifies the local lower-layer service access point to be used for communication. This access point has to be prepared for communication prior to invoking the session services; this is expected to be accomplished with interactions between the service user and management entities in a manner that is not a part of this specification.

6.2.2 Body and Headers

The *Body* type is equivalent to the HTTP entity-body [RFC2616]. The *Headers* type represents a list of attribute information items, which are equivalent to HTTP headers.

6.2.3 Capabilities

The *Capabilities* type represents a set of service facilities and parameter settings, which are related to the operation of the service provider. The predefined capabilities are described in section 6.3.2.2, but the service providers may recognise additional capabilities.

6.2.4 Push Identifier (Push Id)

The *Push Identifier* type represents an abstract value, which can be used to uniquely distinguish among the push transactions of a session that are pending on the service interface.

6.2.5 Reason

The service provider uses the *Reason* type to report the cause of a particular indication primitive. Each provider MAY define additional *Reason* values, but the service user MUST be prepared for the following ones:

Reason Value	Description
PROTOERR	The rules of the protocol prevented the peer from performing the operation in its current state. For example, the used PDU was not allowed.
DISCONNECT	The session was disconnected while the operation was still in progress.
SUSPEND	The session was suspended while the operation was still in progress.
RESUME	The session was resumed while the operation was still in progress.
CONGESTION	The peer implementation could not process the request due to lack of resources.
CONNECTERR	An error prevented session creation.
MRUEXCEEDED	The SDU size in a request was larger than the Maximum Receive Unit negotiated with the peer.
MOREXCEEDED	The negotiated upper limit on the number of simultaneously outstanding method or push requests was exceeded.
PEERREQ	The service peer requested the operation to be aborted.
NETERR	An underlying network error prevented completion of a request.
USERREQ	An action of the local service user was the cause of the indication.

6.2.6 Request URI

The *Request URI* parameter type is intended to have a similar use as the Request-URI in HTTP method requests [RFC2616]. However, the session user MAY use it as it sees fit, even leaving it empty or including binary data not compatible with the URI syntax.

6.2.7 Status

The *Status* parameter type has values equivalent to the HTTP/1.1 status codes [RFC2616].

6.2.8 Transaction Identifier (Transaction Id)

The *Transaction Identifier* type represents an abstract value, which can be used to uniquely distinguish among the method invocation transactions of a session that are pending on the service interface.

6.3 Connection-mode Session Service

6.3.1 Overview

The connection-mode session service is divided into facilities, some of which are optional. Most of the facilities are asymmetric so that the operations available for the client and the server connected by the session are different. The provided facilities are

- Session Management facility
- Method Invocation facility
- Exception Reporting facility
- Push facility
- Confirmed Push facility
- Session Resume facility

The Session Management and Exception reporting facilities are always available. The others are controlled by capability negotiation during session establishment.

Session Management allows a client to connect with a server and to agree on the facilities and protocol options to be used. A server can refuse the connection attempt, optionally redirecting the client to another server. During session establishment the client and server can also exchange attribute information, which is expected to remain valid for the duration of the session. Both the server and the client service user can also terminate the session, so that the peer is eventually notified about the

termination. The user is also notified if session termination occurs due to the action of the service provider or a management entity.

Method Invocation permits the client to ask the server to execute an operation and return the result. The available operations are the HTTP methods [RFC2616] or user-defined extension operations, which fit into the same request-reply or transaction pattern. The service users both in the client and the server are always notified about the completion of the transaction, whether it succeeded or failed. Failure can be caused by an abort initiated either by the service user or the service provider.

The *Exception Reporting* facility allows the service provider to notify the user about events that are related to no particular transaction and do not cause a change in the state of the session.

The *Push* facility permits the server to send unsolicited information to the client taking advantage of the session information shared by the client and the server. This facility is a non-confirmed one, so delivery of the information MAY be unreliable.

The *Confirmed Push* facility is similar to the *Push* facility, but the client confirms the receipt of the information. The client may also choose to abort the push, so that the server is notified.

The *Session Resume* facility includes means to suspend a session so that the state of the session is preserved, but both peers know that further communication is not possible until the client resumes the session. This mechanism is also used to handle the situations in which the service provider detects that further communication is no longer possible, until some corrective action is taken by the service user or management entities. It can also be used to switch the session to use an alternate bearer network, which has more appropriate properties than the one being used. This facility SHOULD be implemented to ensure reasonable behaviour in certain bearer network environments.

6.3.2 Capabilities

Information that is related to the operation of the session service provider is handled using *capabilities*. Capabilities are used for a wide variety of purposes, ranging from representing the selected set of service facilities and settings of particular protocol parameters, to establishing the code page and extension method names used by both peers.

6.3.2.1 Capability Negotiation

Capability negotiation is used between service peers to agree on a mutually acceptable level of service, and to optimise the operation of the service provider according to the actual requirements of the service user. Capability negotiation is to be applied only to *negotiable* capabilities; *informational* capabilities are to be communicated to the peer service user without modifications.

The peer which starts the capability negotiation process is called the *initiator*, and the other peer is called the *responder*. Only a *one-way capability negotiation* is defined, in which the initiator proposes a set of capabilities, and the responder replies to these. The capability negotiation process is under the control of the initiator, so that the responder MUST NOT ever reply with any capability setting, which implies a higher level of functionality than the one proposed by the initiator and supported by the service provider peers. Capability negotiation applies always to all the known capabilities. If a particular capability is omitted from the set of capabilities carried by a service primitive, this must be interpreted to mean that the originator of the primitive wants to use the current capability setting, either the default or the value agreed upon during capability negotiation process. However, the responder may still reply with a different capability value, as long as this does not imply a higher level of functionality.

The one-way capability negotiation proceeds as follows:

1. Service user in initiator proposes a set of capability values.
2. The service provider in the initiator modifies the capabilities, so that they do not imply a higher level of functionality than the provider actually can support.
3. The service provider in the responder further modifies the capabilities, so that they do not imply a higher level of functionality than the provider in the responder actually can support.

4. The service user in the responder receives this modified set of capabilities, and responds with a set of capabilities, which reflect the level of functionality it actually wishes to use. If a particular capability is omitted, this is interpreted to mean that the responding service user wants to use the proposed capability setting.
5. The capabilities selected by the service user in the responder are indicated to the service user in the initiator. They will become the default settings, which will be applicable in the next capability negotiation during the session.

If the operation implied by the service primitive that is used to convey the capability information fails, the capability settings that were in effect before the operation shall remain in effect.

If a negotiable capability value is a positive integer, the final capability setting shall be the minimum of the values, which the service users have proposed to use and which the service provider peers are capable of supporting.

If a negotiable capability value is a set, the final capability setting shall contain only those elements, which are all included in the subsets that the service users have proposed to use and which the service provider peers are capable of supporting.

6.3.2.2 Defined Capabilities

A service user and a service provider MUST recognise the following capabilities.

Capability Name	Class	Type	Description
Aliases	I	List of addresses	A service user can use this capability to indicate the alternate addresses the peer may use to access the same service user instance that is using the current session. The addresses are listed in a preference order, with the most preferred alias first. This information can, for example, be used to facilitate a switch to a new bearer, when a session is resumed.
Client SDU Size	N	Positive integer	The client and server use this capability to agree on the size of the largest transaction service data unit, which may be sent to the *client* during the session.
Extended Methods	N	Set of method names	This capability is used to agree on the set of extended methods (beyond those defined in HTTP/1.1), which are supported both by the client and the server peer, and may be used subsequently during the session.
Header Code Pages	N	Set of code page names	This capability is used to agree on the set of extension header code pages, which are supported both by the client and the server, and shall be used subsequently during the session.
Maximum Outstanding Method Requests	N	Positive integer	The client and server use this capability to agree on the maximum number of method invocations, which can be active at the same time during the session.
Maximum Outstanding Push Requests	N	Positive integer	The client and server use this capability to agree on the maximum number of confirmed push invocations, which can be active at the same time during the session.
Protocol Options	N	Set of facilities and features	This capability is used to enable the optional service facilities and features. It may contain elements from the list: Push, Confirmed Push, Session Resume, Acknowledgement Headers, Large Data Transfer. The presence of an element indicates that use of the specific facility or feature is enabled.
Server SDU Size	N	Positive integer	The client and server use this capability to agree on the size of the largest transaction service data unit, which may be sent to the *server* during the session.
Client Message Size	N	Positive integer	The client and server use this capability to agree on the size of the largest message, which may be sent to the *client* during the session. One message may consist of multiple transaction service data units.
Server Message Size	N	Positive integer	The client and server use this capability to agree on the size of the largest message, which may be sent to the *server* during the session. One message may consist of multiple transaction service data units.

In the *Class* column *N* stands for negotiable, *I* for informational.

6.3.3 Service Primitives

This section lists all the abstract service primitives provided by the service and defines their meaning.

6.3.3.1 S-Connect

This primitive is used to initiate session establishment and to notify of its success. It also provides one-way capability negotiation with the client being the initiator and the server being the responder. It is part of the *Session Management* facility.

Primitive / Parameter	S-Connect			
	req	***Ind***	***Res***	***cnf***
Server Address	M	M(=)	–	–
Client Address	M	M(=)	–	–
Client Headers	O	C(=)	–	–
Requested Capabilities	O	M	–	–
Server Headers	–	–	O	C(=)
Negotiated Capabilities	–	–	O	M(=)

Server Address identifies the peer with which the session is to be established.

Client Address identifies the originator of the session.

Client Headers and *Server Headers* represent attribute information compatible with HTTP message headers [RFC2616], which is communicated without modification between the service users. They can be used for application-level parameters or to cache request headers and response headers, respectively, that are constant throughout the session. However, the actual interpretation and use of this information are completely up to the service users. If these parameters are not provided, applications may rely on application-dependant default session headers to provide a static form of session-wide information.

Requested Capabilities and *Negotiated Capabilities* are used to implement the capability negotiation process described in section 6.3.2.1, "Capability Negotiation". If the rules for capability negotiation are violated, the appropriate action is to fail the session establishment.

The service user may during session establishment invoke some service primitives that will turn out not to be part of the finally selected session functionality. When session establishment and the associated capability negotiation completes, such service requests shall be aborted and the appropriate error shall be indicated to the service user. It is an error, if such primitives are invoked after the session has been established, and the appropriate action is a local implementation matter.

The following figure illustrates the primitives used in a successful session establishment. The service user MAY request a method invocation already while the session is being established. Primitives related to this are shown with dashed lines.

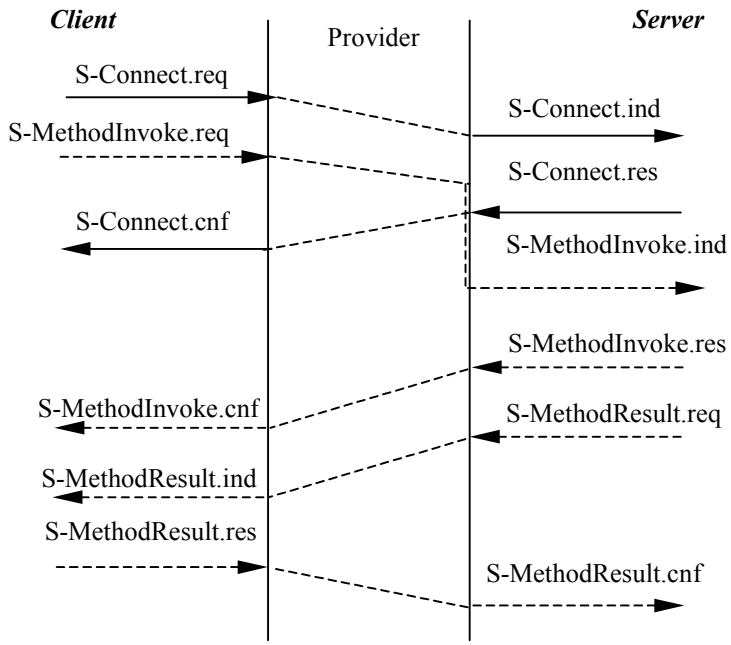

Figure 3: Successful Session Establishment

A disconnect indication generated by the service provider can occur also at any time during the session establishment.

6.3.3.2 S-Disconnect

This primitive is used to disconnect a session and to notify the session user that the session could not be established or has been disconnected. It is part of the *Session Management* facility. This primitive is always indicated when the session termination is detected, regardless of whether the disconnection was initiated by the local service user, the peer service user or the service provider. Before the disconnect indication, the session service provider MUST abort all incomplete method and push transactions. After the indication further primitives associated with the session MUST NOT occur.

Primitive / Parameter	S-Disconnect	
	req	*ind*
Reason Code	M	M
Redirect Security	C	C(=)
Redirect Addresses	C	C(=)
Error Headers	O	P(=)
Error Body	O	P(=)

The *Reason Code* parameter indicates the cause of disconnection. The possible values are a union of the values possible for the Reason and Status parameter types. In S-Disconnect.request only values of the Status type may be used.

If *Reason Code* indicates that the client is being redirected to contact a new server address, the *Redirect Security* and *Redirect Addresses* parameters MUST be present.

Redirect Security indicates whether or not the client MAY reuse the current secure session when redirecting to the new server or whether it MUST use a different secure session.

Redirect Addresses are the alternate addresses, which the client at the moment MUST use to establish a session with the same service it initially tried to contact. If *Reason Code* indicates that the client is being redirected temporarily, it SHOULD use the original *Server Address* in future attempts to establish a session with the service, once the subsequent session with one of

the redirect addresses has terminated. If *Reason* indicates that the client is being redirected permanently, it SHOULD use one of the *Redirect Addresses* in future attempts to establish a session with the service.

If *Reason Code* takes one of the values in the Status type, *Error Headers* and *Error Body* SHOULD be included to provide meaningful information about the error in addition to the *Reason Code*. The size of the headers and body MUST NOT cause the SDU to exceed the currently selected Maximum Receive Unit of the peer. The service provider MAY choose not to communicate the *Error Headers and Error Body* to the peer service user.

The following figure illustrates the primitives used, when the server rejects or redirects the session. The service user MAY request a method invocation already while the session is being established. Primitives related to this are shown with dashed lines.

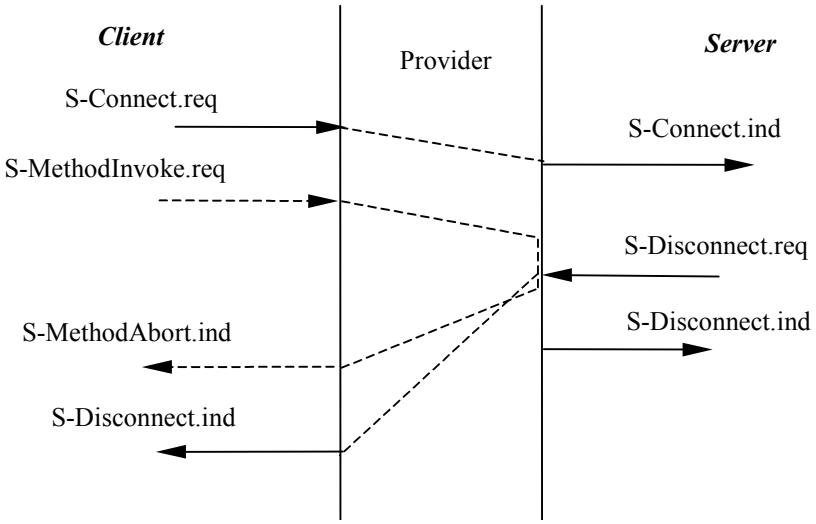

Figure 4: Refused Session Establishment

A disconnect indication generated by the service provider can occur at any time during the session.

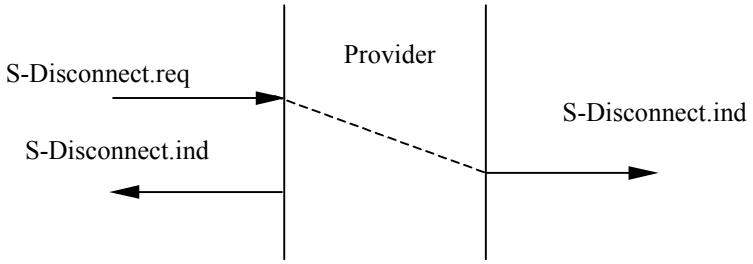

Figure 5: Active Session Termination

The primitive sequence for session termination of an active session is shown in Figure 5. The S-Disconnect.indication indicates that the session has been torn down, and cannot generate any further indications. The service provider shall abort all outstanding transactions prior to the S-Disconnect.indication.

The service user must be prepared for the session being disconnected at any time; if it wishes to continue communication, it has to establish the session again and retry the method invocations that may have been aborted.

6.3.3.3 S-Suspend

This primitive is used to request the session to be suspended, so that no other activity can occur on it, until it is either resumed or disconnected. Before the session becomes suspended, the session service provider MUST abort all incomplete method and push transactions. This primitive is part of the *Session Resume* facility.

Primitive	S-Suspend	
Parameter	***req***	***ind***
Reason	–	M

Reason provides the reason for the suspension. The service user may have requested it, or the service provider may have initiated it.

A possible flow of primitives is shown in the Figure 6:

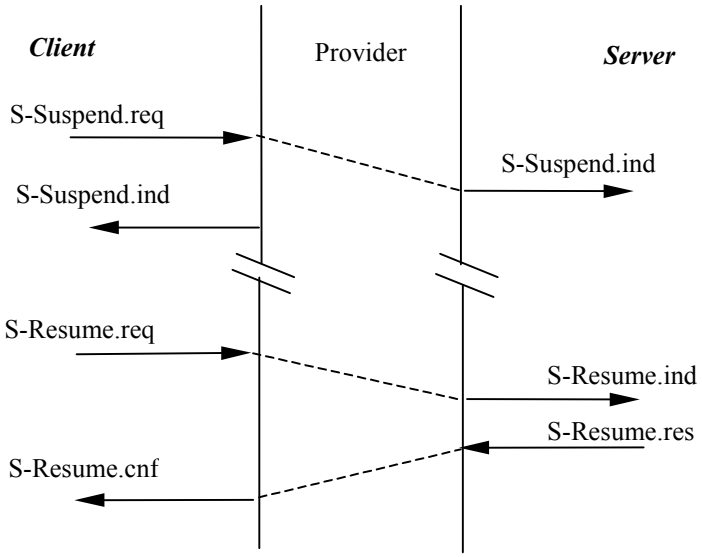

Figure 6: Session Suspension and Resume

Typically, the client would suspend a session, when it knows it will not be available to respond to data pushes, for example, because it will close a data circuit in the underlying bearer network. A side effect of S-Suspend.request is that all data transfer transactions are immediately aborted.

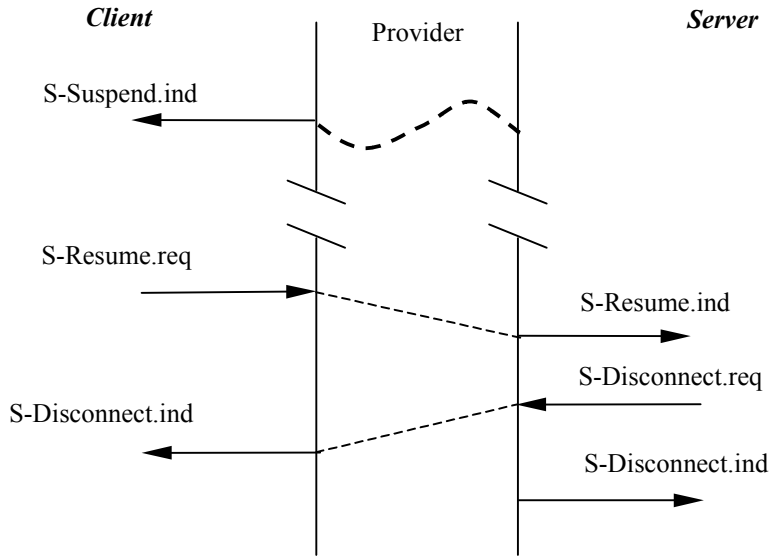

Figure 7: Suspend by Provider and Refused Resume

The service provider MAY also cause an established session to be suspended at any time, eg, due to the bearer network becoming unavailable. Figure 7 shows a scenario, in which only one of the peers - in this case the client - is notified about the suspension. When the client tries to resume the session, the server refuses the attempt by disconnecting the session. For example, the server may consider the used bearer network to be unsuitable.

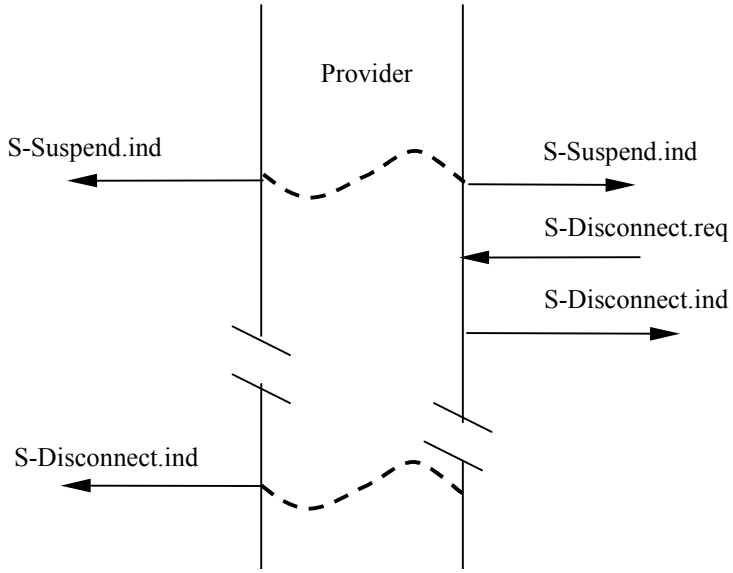

Figure 8: Suspended Session Termination

Figure 8 shows a sequence of events, in which both service users happen to be notified about the suspended session. However, in this case one service user decides to disconnect instead of trying to resume the session. The service user may tear down one half of the session at any time by invoking the S-Disconnect.request primitive. However, the other half of the session will not be notified of this, since the communication path between the service peers is not available. As shown in the figure, the service provider SHOULD eventually terminate a suspended session. The time a suspended session is retained is a local implementation matter.

6.3.3.4 S-Resume

This primitive is used to request the session to be resumed using the new service access point identified by the addresses. It is part of the *Session Resume* facility.

Primitive / Parameter	S-Resume			
	req	***ind***	***res***	***cnf***
Server Address	M	M(=)	–	–
Client Address	M	M(=)	–	–
Client Headers	O	C(=)		
Server Headers			O	C(=)

Server Address identifies the peer with which the session is to be resumed.

Client Address identifies the current origin of the session.

Both the *Server Address and Client Address* MAY be different than the one that was in effect before the session was suspended. If the *Server Address* is different than before suspension, the service user is responsible for providing an address, which will contact the same server instance that was previously in use.

Client Headers and *Server Headers* represent attribute information compatible with HTTP message headers [RFC2616], which is communicated without modification between the service users. They can be used for application-level parameters or to cache request headers and response headers, respectively, that are constant throughout the session.

6.3.3.5 S-Exception

This primitive is used to report events that neither are related to a particular transaction nor cause the session to be disconnected or suspended. It is part of the *Exception Reporting* facility.

Primitive / Parameter	S-Exception
	ind
Exception Data	M

Exception Data includes information from the service provider. Exceptions may occur for many reasons:

- Changes to the underlying transport (eg, roaming out of coverage)
- Changes to quality of service
- Changes or problems in the security layer

6.3.3.6 S-MethodInvoke

This primitive is used to request an operation to be executed by the server. It can be used only together with the S-MethodResult primitive. This primitive is part of the *Method Invocation* facility.

Primitive / Parameter	S-MethodInvoke			
	req	***ind***	***res***	***cnf***
Client Transaction Id	M	–	–	M(=)
Server Transaction Id	–	M	M(=)	–
Method	M	M(=)	–	–
Request URI	M	M(=)	–	–
Request Headers	O	C(=)	–	–
Request Body	C	C(=)	–	–
More Data	M	M(=)	–	–

The service user in the client can use *Client Transaction Id* to distinguish between pending transactions.

The service user in the server can use *Server Transaction Id* to distinguish between pending transactions.

Method identifies the requested operation: either an HTTP method [RFC2616] or one of the extension methods established during capability negotiation.

Request URI specifies the entity to which the operation applies.

Request Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Request Body is the data associated with the request, which is semantically equivalent to HTTP entity body. If the request *Method* is not defined to allow an entity-body, *Request Body* MUST NOT be provided [RFC2616].

More Data is a Boolean flag that specifies whether further S-MethodInvokeData primitives will be following for the same transaction.

6.3.3.7 S-MethodInvokeData

This primitive is used to send more data as a continuation of the S-MethodInvoke primitive. It can be invoked only after a preceding S-MethodInvoke primitive has occurred. This primitive is part of the *Method Invocation* facility.

Primitive / Parameter	S-MethodInvokeData			
	req	***ind***	***res***	***cnf***
Client Transaction Id	M	-	-	M
Server Transaction Id	-	M	M	-
Request Body	C	C(=)	-	-
Request Headers	C	C(=)	-	-
More Data	M	M(=)	-	-

The service user in the client can use *Client Transaction Id* to distinguish between pending transactions. It MUST match the Client Transaction Id of a previous S-MethodInvoke.request.

The service user in the server can use *Server Transaction Id* to distinguish between pending transactions. It MUST match the Server Transaction Id of a previous S-MethodInvoke.ind.

Request Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Request Body is the data associated with the request, which is semantically equivalent to HTTP entity body. If the request *Method* (specified in the S-MethodInvoke) is not defined to allow an entity-body, *Request Body* MUST NOT be provided [RFC2616].

More Data is a Boolean flag that specifies whether additional invocations of the primitive will be following for the same transaction.

6.3.3.8 S-MethodResult

This primitive is used to return a response to an operation request. It can be invoked only after a preceding S-MethodInvoke primitive has occurred. This primitive is part of the *Method Invocation* facility.

Primitive / Parameter	S-MethodResult			
	req	***ind***	***res***	***cnf***
Server Transaction Id	M	–	–	M(=)
Client Transaction Id	–	M	M(=)	–
Status	M	M(=)	–	–
Response Headers	O	C(=)	–	–
Response Body	C	C(=)	–	–
Acknowledgement Headers	–	–	O	P(=)
More Data	M	M(=)	–	–

The service user in the client can use *Client Transaction Id* to distinguish between pending transactions. It MUST match the Client Transaction Id of a previous S-MethodInvoke.request, for which S-MethodResult.indication has not yet occurred.

The service user in the server can use *Server Transaction Id* to distinguish between pending transactions. It MUST match the Server Transaction Id of a previous S-MethodInvoke.response, for which S-MethodResult.request has not yet occurred.

Status is semantically equivalent to an HTTP status code [RFC2616].

Response Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Response Body is the data associated with the response, which is semantically equivalent to an HTTP entity body. If *Status* indicates an error, *Response Body* SHOULD provide additional information about the error in a form, which can be shown to the human user.

Acknowledgement Headers MAY be used to return some information back to the server. However, the provider MAY ignore this parameter or support the transfer of a very limited amount of data.

More Data is a Boolean flag that specifies whether additional primitives-MethodResultData primitives will be following for the same transaction.

The following figure illustrates the flow of primitives in a complete transaction.

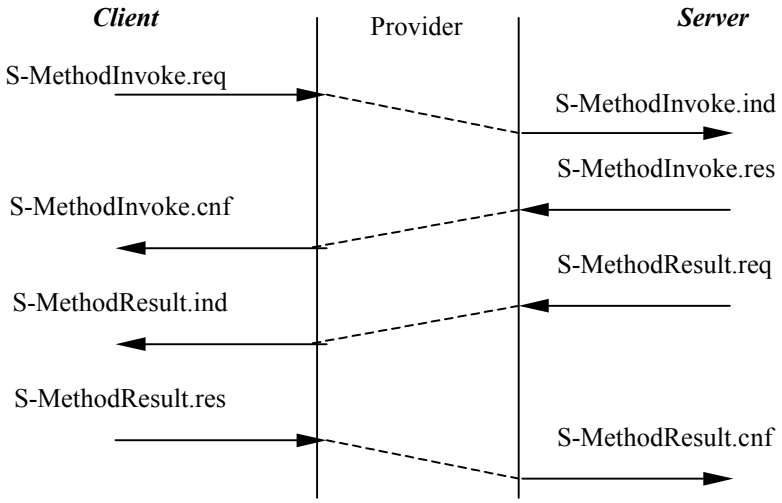

Figure 9: Completed Transaction

If the transaction is aborted for any reason, an S-MethodAbort.indication will be delivered to the service user. It can occur instead of one of the shown indication or confirm primitives or after one of them. Once the abort indication is delivered, no further primitives related to the transaction can occur.

The session layer does not provide any sequencing between multiple overlapping method invocations, so the indications may be delivered in a different order than the corresponding requests. The same applies also to the responses and confirmations, as well as to the corresponding S-MethodResult primitives. The end result is that the results of method invocations may be delivered in an order different from the original order of the requests. The following figure illustrates this (omitting the responses and confirmations for clarity).

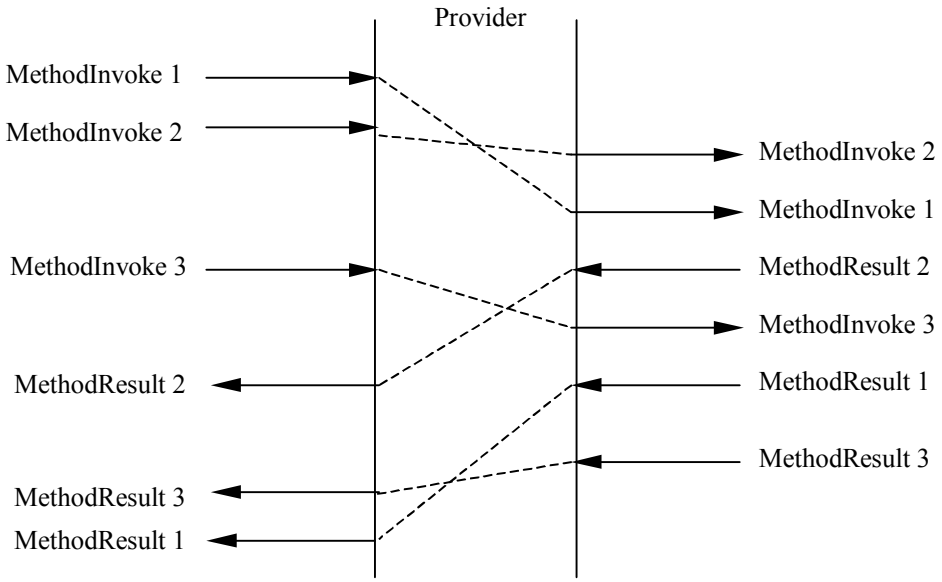

Figure 10: Unordered Asynchronous Requests

6.3.3.9 S-MethodResultData

This primitive is used to return additional data as continuation of the response to an operation request. It can be invoked only after a preceding S-MethodResult primitive has occurred. This primitive is part of the *Method Invocation* facility.

Primitive / Parameter	S-MethodResultData			
	req	*ind*	*res*	*cnf*
Server Transaction Id	M	–	-	M
Client Transaction Id	–	M	M	
Response Body	C	C(=)	-	
Response Headers	C	C(=)	-	
Acknowledgement Headers	–	–	O	P(=)
More Data	M	M(=)	-	

The service user in the client can use *Client Transaction Id* to distinguish between pending transactions. It MUST match the Client Transaction Id of a previous S-MethodResult.indication.

The service user in the server can use *Server Transaction Id* to distinguish between pending transactions. It MUST match the Server Transaction Id of a previous S-MethodResult.request.

Response Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Response Body is the data associated with the response, which is semantically equivalent to an HTTP entity body. If *Status* indicates an error, *Response Body* SHOULD provide additional information about the error in a form, which can be shown to the human user.

Acknowledgement Headers MAY be used to return some information back to the server. It can be present only if the More Data flag was set to False in the request (and indication). However, the provider MAY ignore this parameter or support the transfer of a very limited amount of data.

More Data is a Boolean flag that specifies whether additional invocations of the primitive will be following for the same transaction.

6.3.3.10 S-MethodAbort

This primitive is used to abort an operation request, which is not yet complete. It can be invoked only after a preceding S-MethodInvoke primitive has occurred. It is part of the *Method Invocation* facility.

Primitive / Parameter	S-MethodAbort	
	req	*ind*
Transaction Id	M	M
Reason	–	M

The service user in the client uses *Transaction Id* to distinguish between pending transactions, when invoking S-MethodAbort.request. It MUST match the *Client Transaction Id* of a previous S-MethodInvoke.request, for which S-MethodResult.response has not yet occurred. The *Transaction Id* of the S-MethodAbort.indication in the server will in this case match the *Server Transaction Id* of that transaction.

The service user in the server uses *Transaction Id* to distinguish between pending transactions, when invoking S-MethodAbort.request. It MUST match the *Server Transaction Id* of a previous S-MethodInvoke.indication, for which S-MethodResult.confirm has not yet occurred. The *Transaction Id* of the S-MethodAbort.indication in the client will in this case match the *Client Transaction Id* of that transaction.

Reason is the reason for aborting the transaction. It will be PEERREQ, if the peer invoked S-MethodAbort.request.

There are two scenarios depending on the timing of the primitives.

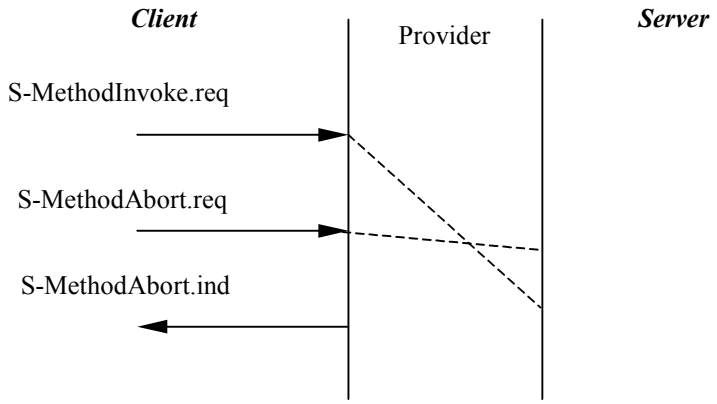

Figure 11: Abort before S-MethodInvoke.indication

The first scenario is shown in Figure 11. The abort request is submitted, while the method invocation is still being communicated to the provider peer, before the S-MethodInvoke.indication has occurred. In this case, the transaction is aborted without the peer user ever being notified about the transaction.

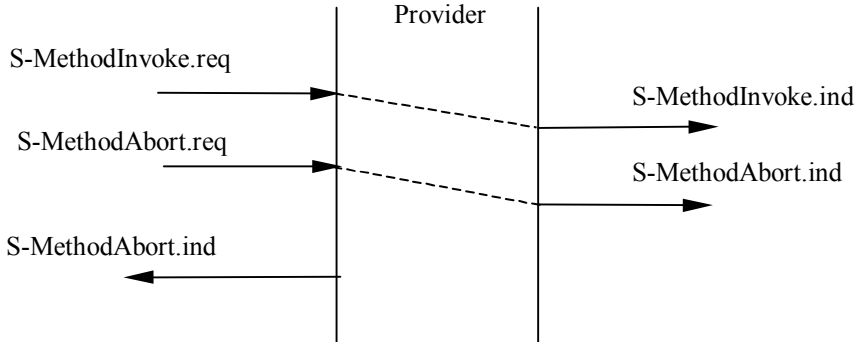

Figure 12: Abort after S-MethodInvoke.indication

The second scenario is shown in Figure 12. The abort request is communicated to the provider peer *after* the S-MethodInvoke.indication has occurred. In this case, the S-MethodAbort.indication will occur as well, and the application MUST NOT invoke any further S-MethodInvoke or S-MethodResult primitives applying to the aborted transaction.

The S-MethodAbort primitive may be invoked in the client at any time between S-MethodInvoke.request and S-MethodResult.response for the transaction to be aborted. Likewise, S-MethodAbort may be invoked in the server at any time between S-MethodInvoke.indication and S-MethodResult.confirm.

6.3.3.11 S-Push

This primitive is used to send unsolicited information from the server within the session context in a non-confirmed manner. This primitive is part of the *Push* facility.

Primitive / Parameter	S-Push	
	Req	***ind***
Push Headers	O	C(=)
Push Body	O	C(=)

If the location of the pushed entity needs to be indicated, the Content-Location header [RFC2616] SHOULD be included in *Push Headers* to ensure interoperability.

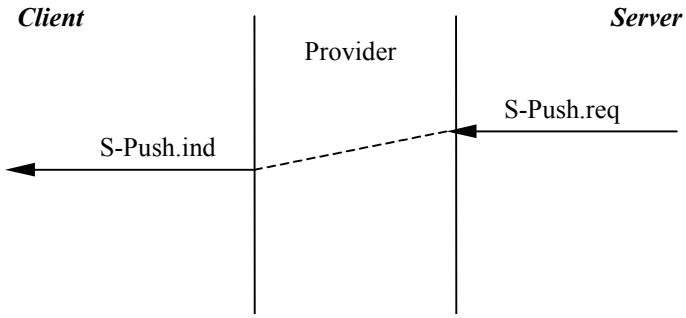

Figure 13: Non-confirmed Data Push

Delivery of information to the peer is not assured, so the following scenario is also permitted:

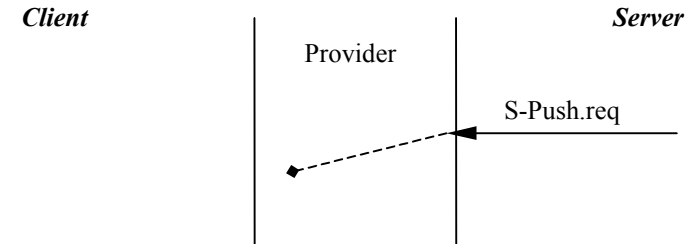

Figure 14: Failed Non-confirmed Data Push

6.3.3.12 S-ConfirmedPush

This primitive is used send unsolicited information from the server within the session context in a confirmed manner. It is part of the *Confirmed Push* facility.

Primitive / Parameter	S-ConfirmedPush			
	req	*ind*	*res*	*cnf*
Server Push Id	M	–	–	M(=)
Client Push Id	–	M	M(=)	–
Push Headers	O	C(=)	–	–
Push Body	O	C(=)	–	–
Acknowledgement Headers	–	–	O	P(=)

The service user in the server can use *Server Push Id* to distinguish between pending pushes.

The service user in the client can use *Client Push Id* to distinguish between pending pushes.

If the location of the pushed entity needs to be indicated, the Content-Location header [RFC2616] SHOULD be included in *Push Headers* to ensure interoperability.

Acknowledgement Headers MAY be used to return some information back to the server. However, the provider MAY ignore this parameter or support the transfer of a very limited amount of data.

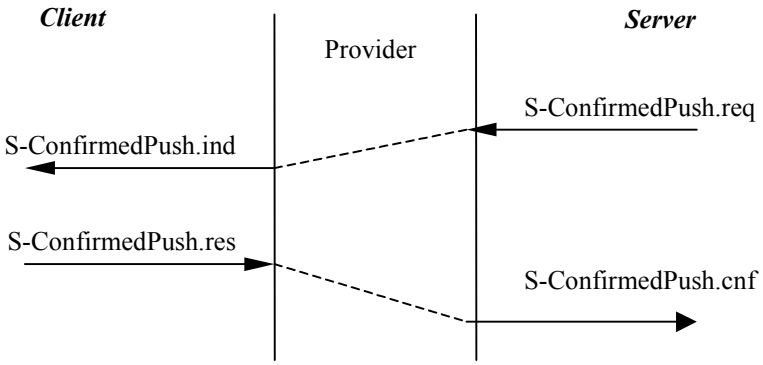

Figure 15: Confirmed Data Push

6.3.3.13 S-PushAbort

This primitive is used to reject a push operation. It is part of the *Confirmed Push* facility.

Primitive / Parameter	S-PushAbort	
	req	*ind*
Push Id	M	M
Reason	M	M

The service user in the client uses *Push Id* to distinguish between pending transactions, when invoking S-PushAbort.request. It MUST match the *Client Push Id* of a previous S-ConfirmedPush.indication. The *Push Id* of the S-PushAbort.indication in the server will in this case match the *Server Push Id* of a previous ConfirmedPush.request, which has not yet been confirmed or indicated as aborted.

Reason is the reason for aborting the push. It will either be the value provided by the peer service user, or a reason code from the service provider.

The following figure shows the behaviour of S-PushAbort. It can be requested only after an S-ConfirmedPush.indication, replacing an S-ConfirmedPush.response.

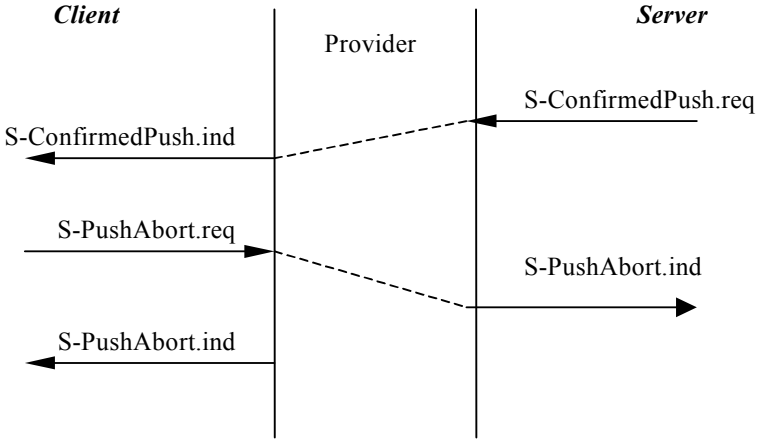

Figure 16: Aborted Confirmed Data Push

S-PushAbort.indication can also occur without the user's request as the result of a provider-initiated abort. In this case, the service user in the client uses *Push Id* to distinguish between pending transactions. It MUST match the *Client Push Id* of a previous S-ConfirmedPush.indication

6.3.4 Constraints on Using the Service Primitives

The following tables define the permitted primitive sequences on the service interface. The client and server have separate tables, since the service is asymmetric.

Only the permitted primitives are listed on the rows; the layer prefix is omitted for brevity. The table entries are interpreted as follows:

Table 2: Table Entry Legend

Entry:	Description
–	The indication or confirm primitive cannot occur.
N/A	Invoking this primitive is an error. The appropriate action is a local implementation matter.
STATE_NAME	Primitive is permitted and moves the service interface view to the named state.
[1]	If the number of outstanding transactions is equal to the selected Maximum Outstanding Method Requests value, invoking this primitive is an error. The appropriate action is a local implementation matter: delivery of the primitive might be delayed, until it is permitted.
[2]	If there is no outstanding transaction with a matching Transaction Id, invoking this primitive is an error. The appropriate action is a local implementation matter.
[3]	If the *Confirmed Push* facility has not been selected during capability negotiation, invoking this primitive is an error. Likewise, if there is no outstanding push with a matching Push Id. The appropriate action is a local implementation matter.
[4]	Possible only if the *Push* facility has been selected during capability negotiation.
[5]	Possible only if the *Confirmed Push* facility has been selected during capability negotiation.
[6]	If the *Push* facility has not been selected during capability negotiation, invoking this primitive is an error. The appropriate action is a local implementation matter.
[7]	If the *Confirmed Push* facility has not been selected during capability negotiation, invoking this primitive is an error. The appropriate action is a local implementation matter.
[8]	If the *Confirmed Push* facility has not been selected during capability negotiation, invoking this primitive is an error. The appropriate action is a local implementation matter. Also if the number of outstanding pushes is equal to the selected Maximum Outstanding Push Requests value, invoking this primitive is an error. The appropriate action is a local implementation matter: delivery of the primitive might be delayed, until it is permitted.
[9]	If the *Session Resume* facility has not been selected during capability negotiation, invoking this primitive is an error. The appropriate action is a local implementation matter.
[10]	Possible only if the *Session Resume* facility has been selected during capability negotiation.

Table 3: Permitted Client Session Layer Primitives

CLIENT	Session States						
S-Primitive	NULL	CONNECTING	CONNECTED	CLOSING	SUSPENDING	SUSPENDED	RESUMING
Connect.req	CONNECTING	N/A	N/A	N/A	N/A	N/A	N/A
Disconnect.req	N/A	CLOSING	CLOSING	N/A	CLOSING	CLOSING	CLOSING
MethodInvoke.req	N/A	[1]	[1]	N/A	N/A	N/A	[1]
MethodResult.res	N/A	N/A	[2]	N/A	N/A	N/A	N/A
MethodAbort.req	N/A	[2]	[2]	N/A	N/A	N/A	[2]
ConfirmedPush.res	N/A	N/A	[3]	N/A	N/A	N/A	N/A
PushAbort.req	N/A	N/A	[3]	N/A	N/A	N/A	N/A
Suspend.req	N/A	N/A	SUSPENDING [9]	N/A	N/A	N/A	SUSPENDING [9]
Resume.req	N/A	N/A	RESUMING [9]	N/A	RESUMING [9]	RESUMING [9]	N/A
Connect.cnf	–	CONNECTED	–	–	–	–	–
Exception.ind	–	CONNECTING	CONNECTED	CLOSING	SUSPENDING	–	RESUMING
Disconnect.ind	–	NULL	NULL	NULL	NULL	NULL	NULL
MethodInvoke.cnf	–	–	CONNECTED	–	–	–	–
MethodResult.ind	–	–	CONNECTED	–	–	–	–
MethodAbort.ind	–	CONNECTING	CONNECTED	CLOSING	SUSPENDING	–	RESUMING
Push.ind	–	–	CONNECTED [4]	CLOSING [4]	SUSPENDING [4]	–	–
ConfirmedPush.ind	–	–	CONNECTED [5]	–	–	–	–
PushAbort.ind	–	–	CONNECTED [5]	–	SUSPENDING [5]	–	–
Suspend.ind	–	–	SUSPENDED [10]	–	SUSPENDED [10]	–	SUSPENDED [10]
Resume.cnf	–	–	–	–	–	–	CONNECTED [10]

Table 4: Permitted Server Session Layer Primitives

SERVER	Session States					
S-Primitive	NULL	CONNECTING	CONNECTED	CLOSING	SUSPENDED	RESUMING
Connect.res	N/A	CONNECTED	N/A	N/A	N/A	N/A
Disconnect.req	N/A	CLOSING	CLOSING	N/A	CLOSING	CLOSING
MethodInvoke.res	N/A	N/A	[2]	N/A	N/A	N/A
MethodResult.req	N/A	N/A	[2]	N/A	N/A	N/A
MethodAbort.req	N/A	N/A	[2]	N/A	N/A	N/A
Push.req	N/A	N/A	[6]	N/A	N/A	N/A
ConfirmedPush.req	N/A	N/A	[8]	N/A	N/A	N/A
Resume.res	N/A	N/A	N/A	N/A	N/A	CONNECTED [9]
Connect.ind	CONNECTING	–	–	–	–	–
Exception.ind	–	CONNECTING	CONNECTED	CLOSING	–	RESUMING
Disconnect.ind	–	NULL	NULL	NULL	NULL	NULL
MethodInvoke.ind	–	–	CONNECTED	–	–	–
MethodResult.cnf	–	–	CONNECTED	–	–	–
MethodAbort.ind	–	–	CONNECTED	CLOSING	–	–
ConfirmedPush.cnf	–	–	CONNECTED [5]	–	–	–
PushAbort.ind	–	–	CONNECTED [5]	CLOSING [5]	–	–
Suspend.ind	–	–	SUSPENDED [10]	–	–	SUSPENDED [10]
Resume.ind	–	–	RESUMING [10]	–	RESUMING [10]	–

The life cycles of transactions in the client and the server are defined by the following two tables. Once again, only the permitted primitives are listed on the rows.

Table 5: Permitted Client Transaction Primitives

CLIENT	Transaction States					
S-Primitive	NULL	REQUESTING	WAITING	WAITING2	COMPLETING	ABORTING
MethodInvoke.req	REQUESTING	N/A	N/A	N/A	N/A	N/A
MethodInvokeData.req	N/A	REQUESTING	N/A	N/A	N/A	N/A
MethodResult.res	N/A	N/A	N/A	WAITING2	NULL	N/A
MethodResultData.res	N/A	N/A	N/A	WAITING2	NULL	N/A
MethodAbort.req	N/A	ABORTING	ABORTING	ABORTING	ABORTING	N/A
MethodInvoke.cnf	-	REQUESTING (if method request is incomplete) OR WAITING (if method request is complete)	-		-	-
MethodInvokeData.cnf	-	REQUESTING (if method request is incomplete) OR WAITING (if method request is complete)	-	-	-	-
MethodResult.ind	-	-	WAITING2 (if MoreData == TRUE) OR COMPLETING (if MoreData == FALSE)	N/A	-	-
MethodResultData.ind	-	-	N/A	WAITING2 (if MoreData == TRUE) OR COMPLETING (if MoreData == FALSE)	-	-
MethodAbort.ind	-	NULL	NULL	NULL	NULL	NULL

Table 6: Permitted Server Transaction Primitives

SERVER	Transaction States				
S-Primitive	NULL	REQUESTING	PROCESSING	REPLYING	ABORTING
MethodInvoke.res	N/A	REQUESTING (if method request is incomplete) OR PROCESSING (if method is complete	N/A	N/A	N/A
MethodInvokeData.res	N/A	REQUESTING (if method request is incomplete) OR PROCESSING (if method request is complete)	N/A	N/A	N/A
MethodResult.req	N/A	N/A	PROCESSING (if MoreData == FALSE) OR REPLYING (if MoreData == TRUE)	N/A	N/A
MethodResultData.req	N/A	N/A	PROCESSING (if MoreData == FALSE) OR REPLYING (if MoreData == TRUE)	N/A	N/A
MethodAbort.req	N/A	ABORTING	ABORTING	ABORTING	N/A
MethodInvoke.ind	REQUESTING	-	-	-	-
MethodInvokeData.ind	-	REQUESTING	-	-	-
MethodResult.cnf	-	-	PROCESSING	NULL	-
MethodResultData.cnf	-	-	PROCESSING	NULL	-
MethodAbort.ind	-	NULL	NULL	NULL	NULL

The life cycles of confirmed push transactions in the server and the client are defined by the following two tables. Once again, only the permitted primitives are listed on the rows.

Table 7: Permitted Server Confirmed Push Primitives

SERVER	Confirmed Push States	
S-Primitive	NULL	PUSHING
ConfirmedPush.req	PUSHING	N/A
ConfirmedPush.cnf	-	NULL
PushAbort.ind	-	NULL

Table 8: Permitted Client Confirmed Push Primitives

CLIENT	Confirmed Push States		
S-Primitive	NULL	RECEIVING	ABORTING
ConfirmedPush.res	N/A	NULL	N/A
PushAbort.req	N/A	ABORTING	N/A
ConfirmedPush.ind	RECEIVING	-	-
PushAbort.ind	-	NULL	NULL

6.3.5 Error Handling

The connection-mode session service provider uses a four-tier strategy in handling errors and other exceptional conditions:

1. If an exceptional condition is not related to any particular transaction, it is reported through the *Exception Reporting* facility without disturbing the overall state of the session.
2. Errors related to a particular transaction cause a method or push abort indication with the appropriate reason code without disturbing the overall state of the session.
3. Conditions which prevent the session peers from communicating with each other will cause suspend indications, if the *Session Resume* facility is selected. Otherwise they will cause a disconnection to be indicated.
4. Other errors will cause a session disconnect to be indicated with the appropriate reason code.

Certain race conditions may cause the abort reason code of a method or push transaction to be reported as DISCONNECT, but this must not be interpreted as indicating that the session has been disconnected; session disconnection is indicated always only using the S-Disconnect primitive.

6.4 Connectionless Session Service

6.4.1 Overview

The connectionless session service provides non-confirmed facilities, which can be used to exchange content entities between layer users. The provided service is asymmetric in a manner similar to the connection-mode service.

Only the Method Invocation and Push facilities are available. The facilities are non-confirmed, so the communication between the peer entities MAY be unreliable.

6.4.2 Service Primitives

The service primitives are defined using types from the *Service Parameter Types* section.

6.4.2.1 S-Unit-MethodInvoke

This primitive is used to invoke a method in the server in a non-confirmed manner. It is part of the Method Invocation facility.

Primitive / Parameter	S-Unit-MethodInvoke	
	req	*ind*
Server Address	M	M(=)
Client Address	M	M(=)
Transaction Id	M	M(=)
Method	M	M(=)
Request URI	M	M(=)
Request Headers	O	C(=)
Request Body	C	C(=)

Server Address identifies the peer to which the request is to be sent.

Client Address identifies the originator of the request.

The service users MAY use *Transaction Id* to distinguish between transactions. It is communicated transparently from service user to service user.

Method identifies the requested operation, which must be one of the HTTP methods [RFC2616].

Request URI specifies the entity to which the operation applies.

Request Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Request Body is the data associated with the request, which is semantically equivalent to HTTP entity body. If the request *Method* is not defined to allow an entity-body, *Request Body* MUST NOT be provided [RFC2616].

6.4.2.2 S-Unit-MethodResult

This primitive is used to return the result of a method invocation from the server in a non-confirmed manner. It is part of the Method Invocation facility.

Primitive / Parameter	S-Unit-MethodResult	
	req	*ind*
Client Address	M	M(=)
Server Address	M	M(=)
Transaction Id	M	M(=)
Status	M	M(=)
Response Headers	O	C(=)
Response Body	C	C(=)

Client Address identifies the peer to which the result is to be sent.

Server Address identifies the originator of the result.

The service users MAY use *Transaction Id* to distinguish between transactions.

Status is semantically equivalent to an HTTP status code [RFC2616].

Response Headers are a list of attribute information semantically equivalent to HTTP headers [RFC2616].

Response Body is the data associated with the response, which is semantically equivalent to an HTTP entity body. If *Status* indicates an error, *Response Body* SHOULD provide additional information about the error in a form, which can be shown to the human user.

6.4.2.3 S-Unit-Push

This primitive is used to send unsolicited information from the server to the client in a non-confirmed manner. It is part of the Push facility.

Primitive / Parameter	S-Unit-Push	
	req	*ind*
Client Address	M	M(=)
Server Address	M	M(=)
Push Id	M	M(=)
Push Headers	O	C(=)
Push Body	O	C(=)

Client Address identifies the peer to which the push is to be sent.

Server Address identifies the originator of the push.

The service users MAY use *Push Id* to distinguish between pushes.

If the location of the pushed entity needs to be indicated, the Content-Location header [RFC2616] SHOULD be included in *Push Headers* to ensure interoperability.

6.4.3 Constraints on Using the Service Primitives

The service user MAY invoke the permitted request primitives at any time, once the underlying layers have been prepared for communication. This is expected to occur through the appropriate interactions with management entities, which are not part of this specification. A failure to do so is an error, and the appropriate action is a local implementation matter.

The service provider SHOULD deliver an indication primitive when it is notified that the corresponding request primitive has been invoked by a peer user entity.

The following table defines the primitives, which the client and server entities are permitted to invoke.

Table 9: Connectionless service primitives

Generic Name	Type				Description
	req	*ind*	*res*	*cnf*	
S-Unit-MethodInvoke	C	S	-	-	Invoke a method in the server with no confirmation
S-Unit-MethodResult	S	C	-	-	Return response from the server with no confirmation
S-Unit-Push	S	C	-	-	Push content with no confirmation
- – Primitive may not occur C – Primitive may occur on the client S – Primitive may occur on the server					

A failure to conform to these restrictions is an error. The appropriate action is a local implementation matter.

6.4.4 Error Handling

If a request cannot be communicated to the provider peer, the connectionless session service provider will not generate any indication primitive. Detection of exceptional conditions and appropriate actions are a local implementation matter.

7 WSP Protocol Operations

This section describes the protocols used between session service peers to realise functions described in the abstract service interface definition.

7.1 Connection-Mode WSP

This section describes the operations of WSP over the WTP transaction service [WTP].

7.1.1 Utilisation of WTP

The WTP transaction classes utilised by each WSP facility is summarised in Table 10.

Table 10. Utilisation of WTP

WSP Facility	WTP Transaction Classes
Session Management	Class 0 and Class 2
Method Invocation	Class 2
Session Resume	Class 0 and Class 2
Push	Class 0
Confirmed Push	Class 1

A connection-mode WSP client MUST support initiation of WTP Class 0 and Class 2 transactions. The client SHOULD accept Class 0 transaction invocations from the server, so that the server is able to disconnect the session explicitly. If the client is to support the push facilities, it MUST accept transactions in the class, which the table above defines to be used by each push facility.

7.1.2 Protocol Description

The following diagrams illustrate the use of a transaction service by the session facilities. The specific details of how the protocol works are expressed in the state tables in section 7.1.6, “State Tables”, below. Any discrepancy between the diagrams and the state tables shall be decided in favour of the state tables.

The dashed arrows represent the WTP protocol messages carrying acknowledgements and WSP PDUs as their data; the messages indicated by parallel arrows are likely to be concatenated into a single transport datagram.

7.1.2.1 Session Management Facility

Normal session creation proceeds without any error or redirection as shown in Figure 17.

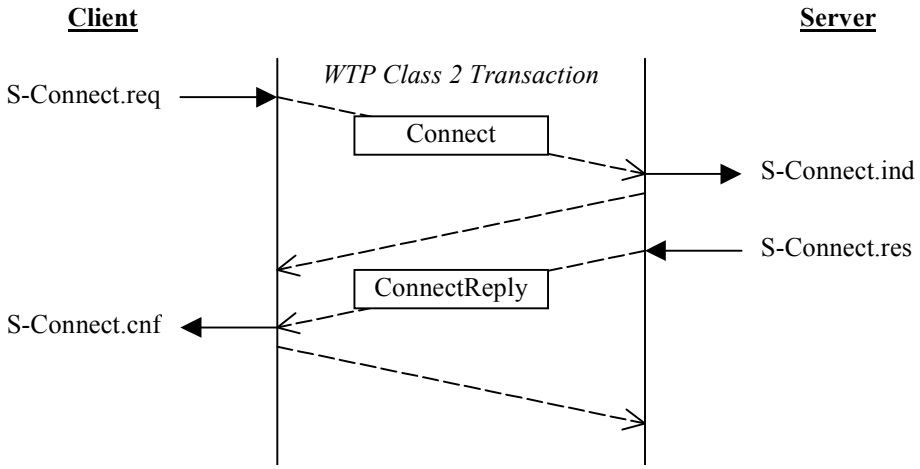

Figure 17. Normal Session Creation

Session creation wherein the client is redirected to another server is shown in Figure 18.

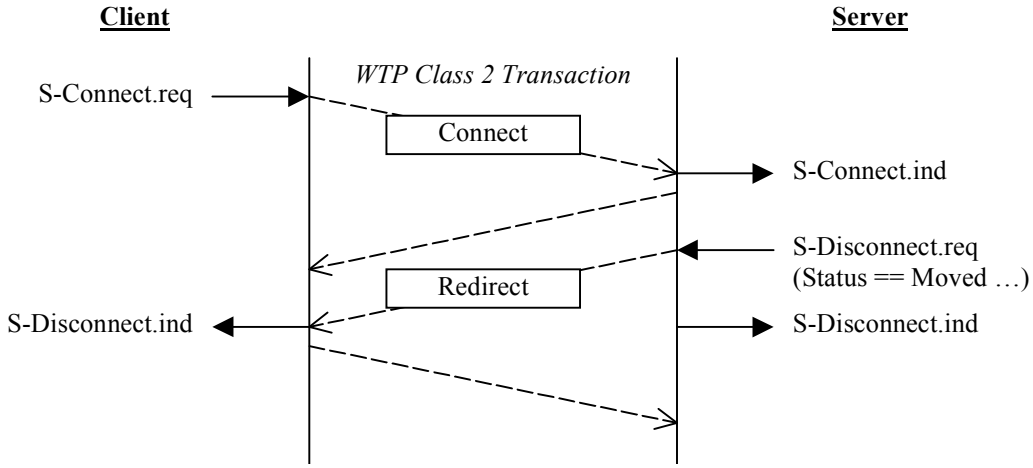

Figure 18. Session Creation with Redirect

Session creation wherein the server session user refuses to accept the session is shown in Figure 19.

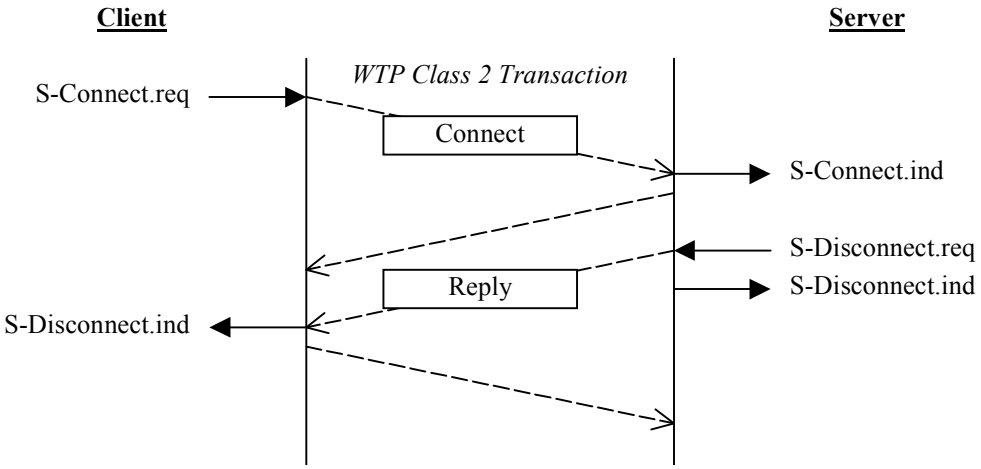

Figure 19. Session Creation with Server Error

Session termination is shown in Figure 20.

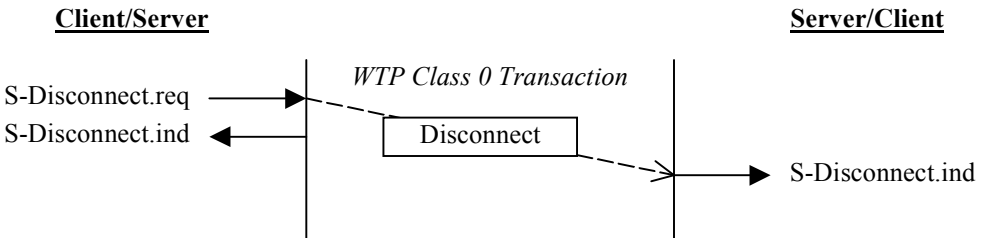

Figure 20. Session Termination

7.1.2.2 Session Resume Facility

Session suspend is shown in Figure 21.

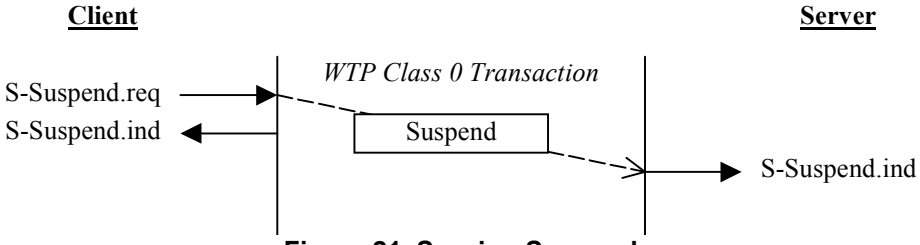

Figure 21. Session Suspend

When session resume succeeds, it proceeds as shown in Figure 22.

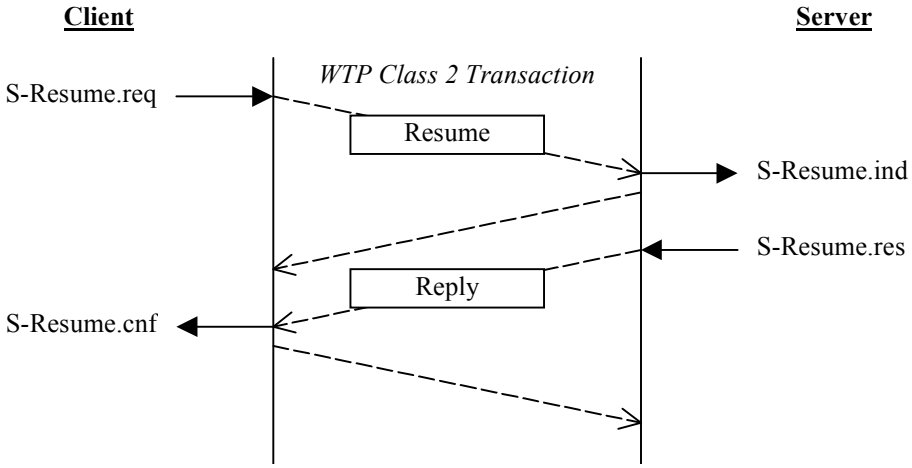

Figure 22. Normal Session Resume

A session resume wherein the server session user refuses to resume the session is shown in Figure 23.

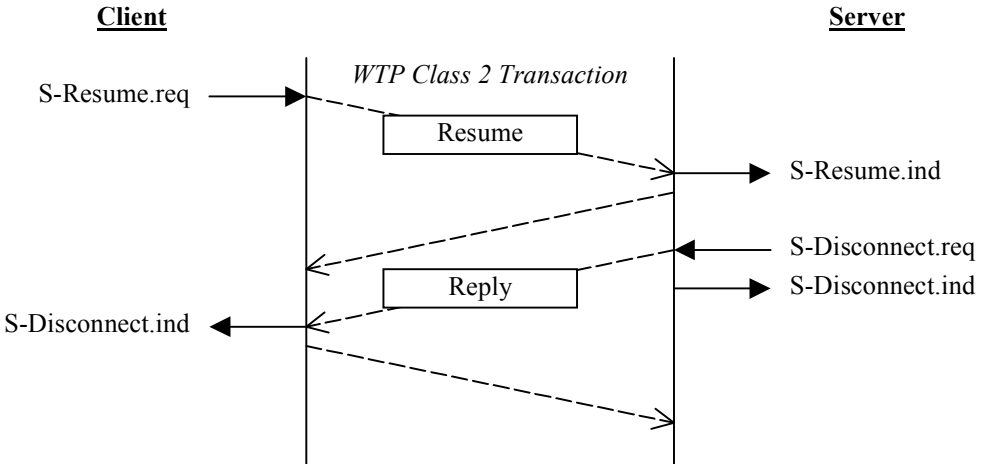

Figure 23. Session Resume with Server Error

7.1.2.3 Method Invocation Facility

A method invocation is shown in Figure 24.

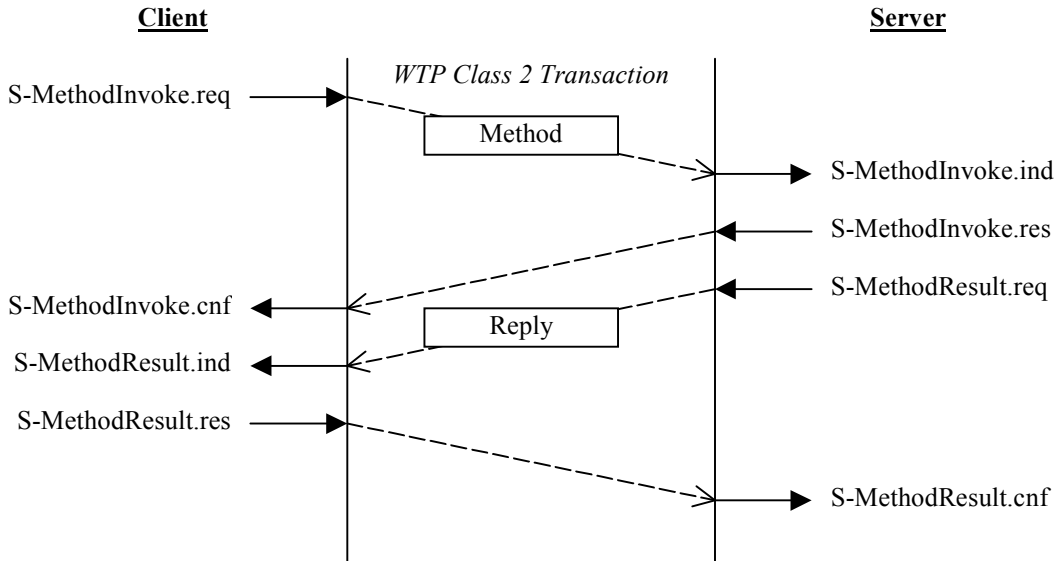

Figure 24. Normal Method Invocation

7.1.2.4 Push Facility

An unconfirmed push is shown in Figure 25.

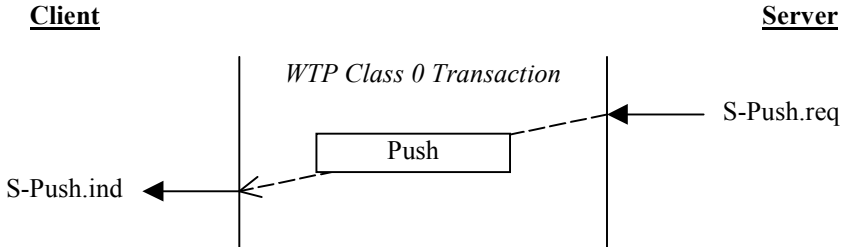

Figure 25. Push Invocation

7.1.2.5 Confirmed Push Facility

A confirmed push is shown in Figure 26.

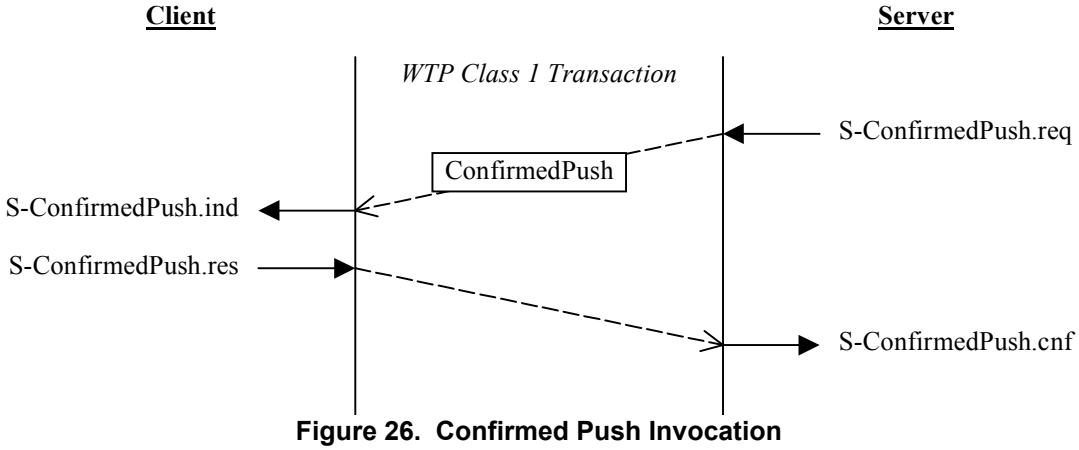

Figure 26. Confirmed Push Invocation

7.1.3 Protocol Parameters

The protocol state machine uses the following parameters.

7.1.3.1 Maximum Receive Unit (MRU)

The Maximum Receive Unit (MRU) is the size of the largest SDU the session layer can accept from the underlying service provider. The initial value is set to the default SDU sizes as specified in section 8.3.3, “Capability Defaults”, below. The value can be modified during capability negotiation.

7.1.3.2 Maximum Outstanding Method Requests (MOM)

The Maximum Outstanding Method Requests (MOM) is the number of method transactions that can be outstanding at a given time. The initial value is set to the default MOM as specified in section 8.3.3, “Capability Defaults”, below. The value can be modified during capability negotiation.

7.1.3.3 Maximum Outstanding Push Requests (MOP)

The Maximum Outstanding Push Requests (MOP) is the number of push transactions that can be outstanding at a given time. The initial value is set to the default MOP as specified in section 8.3.3, “Capability Defaults”, below. The value can be modified during capability negotiation.

7.1.4 Variables

The protocol state machine uses the following variables.

7.1.4.1 N_Methods

N_Methods keeps track of the number of method transactions in process in the server.

7.1.4.2 N_Pushes

N_Pushes keeps track of the number of push transactions in process in the client.

7.1.4.3 Session_ID

Session_ID saves the session identifier assigned by the server in both the client and the server. The method used to assign the identifiers must be chosen so that a session identifier value cannot be repeated during the lifetime of a message in the used transport network; otherwise the session management logic may be confused.

7.1.5 Event Processing

Sessions are associated with a peer address quadruplet, ie, the client address, client port, server address, and server port. Incoming transactions are assigned to a particular session based on the peer address quadruplet. As a consequence, the peer address quadruplet is the true unique protocol-level identifier of a session. There can be only one session bound to a peer address quadruplet at a time.

In order to create a new session for a particular peer address quadruplet when one already appears to exist, the server session provider must allow for the creation of a *proto-session*. This is a second, constrained instance of a session that is used to process the session creation transaction on the server, ie, the Connect and ConnectReply PDUs; this is detailed in the table below.

Indications and confirmations from the transaction layer are termed *events.* Each event is validated and then processed according to the protocol state tables. The protocol state tables also use *pseudo-events* to trigger state changes within the protocol implementation itself. Pseudo-events are generated by the actions in protocol state machines or by the

implementation itself, whenever this is considered appropriate. For instance, they may represent the effect of a management operation, which destroys a session that has been inactive for too long a period.

These pseudo-events are identified by names in *Italics*, and are defined as follows:

Pseudo-Event	Description
Abort	Abort a method or push transaction
Release	Allow a method transaction to proceed
Suspend	Suspend the session
Disconnect	Disconnect the session

Incoming transaction invocations are validated before being processed according to the state tables; the following tests are performed; and if no action is taken, the event is processed according to the state table.

Test	Action
TR-Invoke.ind with SDU size > MRU	TR-Abort.req(MRUEXCEEDED) the TR-Invoke
Class 2 TR-Invoke.ind, on server, Connect PDU	a) Create a new proto-session that is responsible for processing the remainder of the Connect transaction. b) The proto-session signals S-Connect.indication to the session user. c) If the session user accepts the new session by invoking S-Connect.response, the proto-session is turned into a new session for the peer address quadruplet. *Disconnect* is invoked on any old sessions bound to that quadruplet.
Class 2 TR-Invoke.ind, on server, Resume PDU	Pass to session identified by the SessionId in Resume PDU instead of the session identified by the peer address quadruplet. If the SessionId is not valid, ie, the session does not exist, TR-Abort.req(DISCONNECT) the TR-Invoke.
Class 1-2 TR-Invoke.ind, no session matching the peer address quadruplet	TR-Abort.req(DISCONNECT) the TR-Invoke
Class 1-2 TR-Invoke.ind PDU not handled by state tables	TR-Abort.req(PROTOERR) the TR-Invoke
Class 0 TR-Invoke.ind PDU not handled by state tables	Ignore
Any other event not handled by state tables	TR-Abort.req(PROTOERR) if it is some other transaction event than abort *Abort*(PROTOERR) all method and push transactions S-Disconnect.ind(PROTOERR)

The service provided by the underlying transaction layer is such that a protocol entity cannot reliably detect that the peer has discarded the session state information, unless a method or push transaction is in progress. This may eventually result in a large number of sessions, which no longer have any peer protocol entity. The implementation SHOULD be able to *Disconnect* sessions, which are considered to be in such a state.

7.1.6 State Tables

The following state tables define the actions of connection-mode WSP. Because multiple methods and pushes can occur at the same time, there are three state tables defined for client and server: one for the session states, one for the states of a method and one for the states of a push.

The state names used in the tables are logically completely separate from the states defined for the abstract service interface, although the names may be similar. Typically a particular state at the service interface maps into a protocol state with the same name, but a state also may map into multiple or no protocol states at all.

A single *Event* may have several entries in the *Condition* column. In such a case the conditions are expected to be evaluated row by row from top to bottom with the most specific condition being the first one. A single *Condition* entry may contain several conditions separated with a comma ",". In this case all of these have to be satisfied in order for the condition to be true.

7.1.6.1 Client Session State Tables

The following tables show the session states and event processing that occur on the client when using a transaction service.

Client Session NULL			
Event	**Conditions**	**Action**	**Next State**
S-Connect.req		*Disconnect* any other session for the peer address quadruplet. TR-Invoke.req(Class 2, Connect) N_PUSHES = 0	CONNECTING

Client Session CONNECTING			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		TR-Abort.req(DISCONNECT) the Connect *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(USERREQ)	NULL
Disconnect		TR-Abort.req(DISCONNECT) the Connect *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.req		Start a new method transaction with this event (see method state table)	
S-MethodAbort.req		See method state table	
Suspend		TR-Abort.req(DISCONNECT) the Connect *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(SUSPEND)	NULL
TR-Invoke.ind	Class 1, ConfirmedPush PDU	TR-Abort.req(PROTOERR) the TR-Invoke	
TR-Result.ind	Connect transaction, SDU size > MRU	TR-Abort.req(MRUEXCEEDED) the Connect *Abort*(CONNECTERR) all outstanding method transactions S-Disconnect.ind(MRUEXCEEDED)	NULL
	Connect transaction, ConnectReply PDU	TR-Result.res Session_ID = SessionId from PDU S-Connect.cnf	CONNECTED
	Connect transaction, Redirect PDU	TR-Result.res *Abort*(CONNECTERR) all method transactions S-Disconnect.ind(Redirect parameters)	NULL
	Connect transaction, Reply PDU	TR-Result.res *Abort*(CONNECTERR) all method transactions S-Disconnect.ind(Reply parameters)	NULL
	Method Transaction	*Abort* (PROTOERR) the method transaction	
	Other	TR-Abort.req(PROTOERR) *Abort*(CONNECTERR) all outstanding method transactions S-Disconnect.ind(PROTOERR)	NULL
TR-Invoke.cnf	Connect transaction	Ignore	
	Method transaction	*Abort*(PROTOERR) method transaction	
TR-Abort.ind	Connect transaction	*Abort*(CONNECTERR) all outstanding method transactions S-Disconnect.ind(abort reason)	NULL
	Method transaction	See method state table	

Client Session CONNECTED			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		*Abort*(DISCONNECT) all method and push transactions TR-Invoke.req(Class 0, Disconnect) S-Disconnect.ind(USERREQ)	NULL
Disconnect		*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.req		Start a new method transaction with this event	
S-MethodResult.res		See method state table	
S-MethodAbort.req		See method state table	
S-ConfirmedPush.res		See push state table	
S-PushAbort.req		See push state table	
S-Suspend.req		*Abort*(SUSPEND) all method and push transactions TR-Invoke.req(Class 0, Suspend) S-Suspend.ind(USERREQ)	SUSPENDED
Suspend	Session Resume facility disabled	*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(SUSPEND)	NULL
	Session Resume facility enabled	*Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
S-Resume.req		*Abort*(USERREQ) all method and push transactions Bind session to the new peer address quadruplet TR-Invoke(Class 2, Resume)	RESUMING
TR-Invoke.ind	Class 0, Disconnect PDU	*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
	Class 0, Push PDU, Push facility enabled	S-Push.ind	
	Class 1, ConfirmedPush PDU, Confirmed Push facility enabled	Start a new push transaction with this event	
TR-Result.ind	Method transaction	See method state table	
TR-Invoke.cnf	Method transaction	See method state table	
TR-Abort.ind	Method transaction	See method state table	
	Push transaction	See push state table	

Client Session SUSPENDED			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		S-Disconnect.ind(USERREQ)	NULL
Disconnect		S-Disconnect.ind(DISCONNECT)	NULL
S-Resume.req		TR-Invoke.req(Class 2, Resume)	RESUMING
TR-Invoke.ind	Class 0, Disconnect PDU	S-Disconnect.ind(DISCONNECT)	NULL
	Class 1, ConfirmedPush PDU, Confirmed Push facility enabled	TR-Abort.req(SUSPEND) the TR-Invoke	
TR-Invoke.cnf		Ignore	
TR-Abort.ind		Ignore	

Client Session RESUMING			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		TR-Abort.req(DISCONNECT) the Resume *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(USERREQ)	NULL
Disconnect		TR-Abort.req(DISCONNECT) the Resume *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.req		Start a new method transaction with this event (see method state table)	
S-MethodAbort.req		See method state table	
S-Suspend.req		TR-Abort.req(SUSPEND) the Resume *Abort*(SUSPEND) all outstanding method transactions TR-Invoke.req(Class 0, Suspend) S-Suspend.ind(USERREQ)	SUSPENDED
Suspend		TR-Abort.req(SUSPEND) the Resume *Abort*(SUSPEND) all outstanding method transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.ind	Class 0, Disconnect PDU	TR-Abort.req(DISCONNECT) the Resume *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(DISCONNECT)	NULL
	Class 1, ConfirmedPush PDU, Confirmed Push facility enabled	TR-Abort.req(PROTOERR) the TR-Invoke	
TR-Result.ind	Resume transaction, SDU size > MRU	TR-Abort.req(MRUEXCEEDED) the TR-Result *Abort*(SUSPEND) all outstanding method transactions S-Suspend.ind(MRUEXCEEDED)	SUSPENDED
	Resume transaction, Reply PDU (status == OK)	TR-Result.res S-Resume.cnf	CONNECTED
	Resume transaction, Reply PDU (status != OK)	TR-Result.res *Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(Reply parameters)	NULL
	Method Transaction	*Abort*(PROTOERR) the method transaction	
	Other	TR-Abort.req(PROTOERR) the TR-Result *Abort*(SUSPEND) all outstanding method transactions S-Suspend.ind(PROTOERR)	SUSPENDED
TR-Invoke.cnf	Resume transaction	Ignore	
	Method transaction	*Abort*(PROTOERR) method transaction	
TR-Abort.ind	Resume transaction, Reason == DISCONNECT	*Abort*(DISCONNECT) all outstanding method transactions S-Disconnect.ind(DISCONNECT)	NULL
	Resume transaction	*Abort*(SUSPEND) all outstanding method transactions S-Suspend.ind(abort reason)	SUSPENDED
	Method transaction	See method state table	

7.1.6.2 Client Method State Tables

The following tables show the method states and event processing that occur on the client when using a transaction service.

Client Method NULL			
Event	**Conditions**	**Action**	**Next State**
S-MethodInvoke.req		TR-Invoke.req(Class 2, Method, MoreData) *Note: "Method" means either the Get or Post PDU using the PDU type assigned to the particular method.*	REQUESTING

Client Method REQUESTING			
Event	**Conditions**	**Action**	**Next State**
S-MethodInvoke Data.req	More Data == TRUE	TR-InvokeData.req(request body, MoreData)	REQUESTING
	More Data == False , Request Headers not provided	TR-InvokeData.req(request body, MoreData)	REQUESTING
	More Data == FALSE, Request Headers provided	TR-InvokeData.req(Data Fragment PDU, MoreData, Frame Boundary)	REQUESTING
S-MethodAbort.req		TR-Abort.req(PEERREQ) the Method S-MethodAbort.ind(USERREQ)	NULL
Abort		TR-Abort.req(abort reason) the Method S-MethodAbort.ind(abort reason)	NULL
TR-Invoke.cnf	Marks completion of entire method	S-MethodInvoke.cnf	WAITING
	Other	S-MethodInvoke.cnf	REQUESTING
TR-InvokeData.cnf	Marks completion of entire method	S-MethodInvokeData.cnf	WAITING
	Other	S-MethodInvokeData.cnf	REQUESTING
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	S-MethodAbort.ind(abort reason)	NULL

Client Method WAITING			
Event	**Conditions**	**Action**	**Next State**
S-MethodAbort.req		TR-Abort.req(PEERREQ) the Method S-MethodAbort.ind(USERREQ)	NULL
Abort		TR-Abort.req(abort reason) the Method S-MethodAbort.ind(abort reason)	NULL
TR-Result.ind	SDU size > MRU	TR-Abort.req(MRUEXCEEDED) S-MethodAbort.ind(MRUEXCEEDED)	NULL
	Reply PDU, MoreData flag set	TR-Result.res S-MethodResult.ind(headers, body, MoreData)	WAITING2
	Reply PDU, MoreData flag cleared	S-MethodResult.ind(headers, body, MoreData)	COMPLETING
	Other	TR-Abort.req(PROTOERR) S-MethodAbort.ind(PROTOERR)	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	S-MethodAbort.ind(abort reason)	NULL

Client Method WAITING2			
Event	**Conditions**	**Action**	**Next State**
S-MethodAbort.req		TR-Abort.req(PEERREQ) the Method S-MethodAbort.ind(USERREQ)	NULL
S-MethodResult.res		Ignore	WAITING2
S-MethodResult Data.res		Ignore	WAITING2
Abort		TR-Abort.req(abort reason) the Method S-MethodAbort.ind(abort reason)	NULL
TR-ResultData.ind	SDU size > MRU	TR-Abort.req(MRUEXCEEDED) S-MethodAbort.ind(MRUEXCEEDED)	NULL
	Response Body, MoreData flag set	TR-ResultData.res S-MethodResultData.ind(body, MoreData)	WAITING2
	Frame Boundary, Data Fragment PDU, , MoreData flag cleared.	S-MethodResultData.ind(headers, body, MoreData)	COMPLETING
	No Frame Boundary, Response Body MoreData flag cleared.	S-MethodResultData.ind(body, MoreData)	COMPLETING
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	NULL
	Reason == SUSPEND	*Suspend* the session	
	Other	S-MethodAbort.ind(abort reason)	

Client Method COMPLETING			
Event	**Conditions**	**Action**	**Next State**
S-MethodResult.res		TR-Result.res(Exit Info = Acknowledgement Headers) *Note:support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature*	NULL
S-MethodResult Data.res		TR-ResultData.res(Exit Info = Acknowledgement Headers) *Note: support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature*	NULL
S-MethodAbort.req		TR-Abort.req(PEERREQ) the Method S-MethodAbort.ind(USERREQ)	NULL
Abort		TR-Abort.req(abort reason) the Method S-MethodAbort.ind(abort reason)	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	S-MethodAbort.ind(abort reason)	NULL

7.1.6.3 Client Push State Tables

The following tables show the push states and event processing that occur on the client when using a transaction service.

Client Push NULL			
Event	**Conditions**	**Action**	**Next State**
TR-Invoke.ind	Class 1, ConfirmedPush PDU, N_PUSHES == MOP	TR-Abort.req(MOREXCEEDED) the TR-Invoke	NULL
	Class 1, ConfirmedPush PDU, N_PUSHES < MOP	Increment N_PUSHES S-ConfirmedPush.ind	RECEIVING

Client Push RECEIVING			
Event	**Conditions**	**Action**	**Next State**
S-ConfirmedPush.res		TR-Invoke.res(Exit Info = Acknowledgement Headers) *Note:support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature* Decrement N_PUSHES	NULL
S-PushAbort.req		TR-Abort.req(abort reason) the TR-Invoke S-PushAbort.ind(USERREQ) Decrement N_PUSHES	NULL
Abort		TR-Abort.req(abort reason) the TR-Invoke S-PushAbort.ind(abort reason) Decrement N_PUSHES	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	S-PushAbort.ind(abort reason) Decrement N_PUSHES	NULL

7.1.6.4 Server Session State Tables

The following tables show the session states and event processing that occur on the server when using a transaction service.

Server Session NULL			
Event	**Conditions**	**Action**	**Next State**
TR-Invoke.ind	Class 2, Connect	TR-Invoke.res N_Methods = 0 S-Connect.ind	CONNECTING

Server Session CONNECTING			
Event	**Conditions**	**Action**	**Next State**
S-Connect.res		*Disconnect* any other session for this peer address quadruplet. Assign a Session_ID for this session. TR-Result.req(ConnectReply) *Release* all method transactions in HOLDING state	CONNECTING_2
S-Disconnect.req	Reason Code == Moved Permanently *or* Moved Temporarily	TR-Result.req(Redirect) *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(USERREQ)	TERMINATING
	Other	TR-Result.req(Reply(status = Reason Code)) *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(USERREQ)	TERMINATING
Disconnect		TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(DISCONNECT)	NULL
Suspend		TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(SUSPEND)	NULL
TR-Invoke.ind	Class 2, Method	Start new method transaction (see method state table)	
	Class 2, Resume	TR-Abort.req(DISCONNECT) the TR-Invoke	
TR-Abort.ind	Connect transaction	*Abort*(DISCONNECT) all method transactions S-Disconnect.ind(abort reason)	NULL
	Method transaction	See method state table	

Server Session TERMINATING			
Event	**Conditions**	**Action**	**Next State**
Disconnect		TR-Abort.req(DISCONNECT) remaining transport transaction	NULL
Suspend		TR-Abort.req(DISCONNECT) remaining transport transaction	NULL
TR-Result.cnf		Ignore	NULL
TR-Abort.ind		Ignore	NULL

Server Session CONNECTING_2			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method and push transactions TR-Invoke.req(Class 0, Disconnect) S-Disconnect.ind(USERREQ)	NULL
Disconnect		TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.res		See method state table	
S-MethodResult.req		See method state table	
S-MethodAbort.req		See method state table	
S-Push.req		TR-Invoke.req(Class 0, Push)	
S-ConfirmedPush.req		Start new push transaction (see push state table)	
Suspend	Session Resume facility disabled	TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(SUSPEND)	NULL
	Session Resume facility enabled	TR-Abort.req(SUSPEND) the Connect transaction *Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.ind	Class 2, Method	Start new method transaction (see method state table) *Release* the new method transaction	
	Class 2, Resume, Session Resume facility disabled	TR-Abort.req(DISCONNECT) the TR-Invoke	
	Class 2, Resume, Session Resume facility enabled	TR-Invoke.res TR-Abort.req(RESUME) the Connect transaction *Abort*(RESUME) all method and push transactions S-Suspend.ind(RESUME) S-Resume.ind	RESUMING
	Class 0, Disconnect	TR-Abort.req(DISCONNECT) the Connect transaction *Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
	Class 0, Suspend, Session Resume facility enabled	TR-Abort.req(SUSPEND) the Connect transaction *Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.cnf	Push transaction	See push state table	
TR-Result.cnf	Connect transaction		CONNECTED
	Method transaction	See method state table	
TR-Abort.ind	Connect transaction	*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(abort reason)	NULL
	Push transaction	See push state table	
	Method transaction	See method state table	

Server Session CONNECTED			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		*Abort*(DISCONNECT) all method and push transactions TR-Invoke.req(Class 0, Disconnect) S-Disconnect.ind(USERREQ)	NULL
Disconnect		*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.res		See method state table	
S-MethodResult.req		See method state table	
S-MethodAbort.req		See method state table	
S-Push.req		TR-Invoke.req(Class 0, Push)	
S-ConfirmedPush.req		Start new push transaction (see push state table)	
Suspend	Session Resume facility disabled	*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(SUSPEND)	NULL
	Session Resume facility enabled	*Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.ind	Class 2, Method	Start new method transaction (see method state table) *Release* the new method transaction	
	Class 2, Resume, Session Resume facility disabled	TR-Abort.req(DISCONNECT) the TR-Invoke	
	Class 2, Resume, Session Resume facility enabled	TR-Invoke.res *Abort*(RESUME) all method and push transactions S-Suspend.ind(RESUME) S-Resume.ind	RESUMING
	Class 0, Disconnect	*Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
	Class 0, Suspend, Session Resume facility enabled	*Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.cnf	Push transaction	See push state table	
TR-Result.cnf	Method transaction	See method state table	
TR-Abort.ind	Push transaction	See push state table	
	Method transaction	See method state table	

Server Session SUSPENDED			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		S-Disconnect.ind(USERREQ)	NULL
Disconnect		S-Disconnect.ind(DISCONNECT)	NULL
TR-Invoke.ind	Class 2, Method	TR-Abort.req(SUSPEND) the TR-Invoke	
	Class 2, Resume	TR-Invoke.res S-Resume.ind	RESUMING
	Class 0, Disconnect	S-Disconnect.ind(DISCONNECT)	NULL

Server Session RESUMING			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		TR-Abort.req(DISCONNECT) the Resume transaction *Abort*(DISCONNECT) all method transactions TR-Invoke.req(Class 0, Disconnect) S-Disconnect.ind(USERREQ)	NULL
Disconnect		TR-Abort.req(DISCONNECT) the Resume transaction *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(DISCONNECT)	NULL
S-Resume.res		*Disconnect* any other session for the peer address quadruplet. Bind session to new peer address quadruplet TR-Result.req(Reply) *Release* all method transactions in HOLDING state	RESUMING_2
Suspend		TR-Abort.req(SUSPEND) the Resume transaction *Abort*(SUSPEND) all method transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.ind	Class 2, Method	Start new method transaction (see method state table)	
	Class 2, Resume	TR-Invoke.res TR-Abort.req(RESUME) the old Resume transaction *Abort*(RESUME) all method transactions S-Suspend.ind(RESUME) S-Resume.ind	
	Class 0, Suspend	TR-Abort.req(SUSPEND) the Resume transaction *Abort*(SUSPEND) all method transactions S-Suspend.ind(SUSPEND)	SUSPENDED
	Class 0, Disconnect	TR-Abort.req(DISCONNECT) the Resume transaction *Abort*(DISCONNECT) all method transactions S-Disconnect.ind(DISCONNECT)	NULL
TR-Abort.ind	Resume transaction, Reason = DISCONNECT	*Abort* (DISCONNECT) all method transactions S-Disconnect.ind(DISCONNECT)	NULL
	Resume transaction	*Abort*(SUSPEND) all method transactions S-Suspend.ind(abort reason)	SUSPENDED

Server Session RESUMING_2			
Event	**Conditions**	**Action**	**Next State**
S-Disconnect.req		TR-Abort.req(DISCONNECT) the Resume transaction *Abort*(DISCONNECT) all method and push transactions TR-Invoke.req(Class 0, Disconnect) S-Disconnect.ind(USERREQ)	NULL
Disconnect		TR-Abort.req(DISCONNECT) the Resume *Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
S-MethodInvoke.res		See method state table	
S-MethodResult.req		See method state table	
S-MethodAbort.req		See method state table	
S-Push.req		TR-Invoke.req(Class 0, Push)	
S-ConfirmedPush.req		Start new push transaction (see push state table)	
Suspend		TR-Abort.req(SUSPEND) the Resume transaction *Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
TR-Invoke.ind	Class 2, Method	Start new method transaction (see method state table) *Release* the new method transaction	
	Class 2, Resume	TR-Invoke.res TR-Abort.req(RESUME) the old resume transaction *Abort*(RESUME) all method and push transactions S-Suspend.ind(RESUME) S-Resume.ind	RESUMING
	Class 0, Suspend	TR-Abort.req(SUSPEND) the Resume transaction *Abort*(SUSPEND) all method and push transactions S-Suspend.ind(SUSPEND)	SUSPENDED
	Class 0, Disconnect	TR-Abort.req(DISCONNECT) the Resume *Abort*(DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
TR-Invoke.cnf	Push transaction	See push state table	
TR-Result.cnf	Resume transaction		CONNECTED
	Method transaction	See method state table	
TR-Abort.ind	Resume transaction, Reason = DISCONNECT	*Abort* (DISCONNECT) all method and push transactions S-Disconnect.ind(DISCONNECT)	NULL
	Resume transaction	*Abort*(SUSPEND) all method and push transactions S-Suspend.ind(abort reason)	SUSPENDED
	Push transaction	See push state table	
	Method transaction	See method state table	

7.1.6.5 Server Method State Tables

The following tables show the method states and event processing that occur on the server when using a transaction service.

Server Method NULL			
Event	**Conditions**	**Action**	**Next State**
TR-Invoke.ind	Class 2, Method PDU, N_Methods == MOM	TR-Abort.req(MOREXCEEDED)	NULL
	Class 2, Method PDU, MoreData flag set	Store Method Increment N_Methods TR-Invoke.res	HOLDING
	Class 2, Method PDU, MoreData flag cleared	Store Method Increment N_Methods	HOLDING

Server Method HOLDING			
Event	**Conditions**	**Action**	**Next State**
TR-InvokeData.ind	Class 2, Request Body, MoreData flag set	Store Method TR-InvokeData.res	HOLDING
	Class 2, MoreData flag cleared	Store Method	HOLDING
Release		Generate S-MethodInvoke.ind (headers, body, MoreData) If stored, generate one or more	REQUESTING

Server Method HOLDING			
Event	**Conditions**	**Action**	**Next State**
		S-MethodInvokeData.ind(headers, body, MoreData) Note: headers can only be included in the S-MethodInvokeData.ind for which MoreData is cleared and where Frame Boundary was included in the TR-InvokeData.ind	
Abort		Decrement N_Methods TR-Abort.req(abort reason) the method	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	Decrement N_Methods	NULL

Server Method REQUESTING			
Event	**Conditions**	**Action**	**Next State**
S-MethodInvoke.res	Complete Invoke already received	TR-Invoke.res	PROCESSING
	Complete Invoke not received	Ignore	REQUESTING
S-MethodInvoke Data.res	Complete Invoke already received	TR-Invoke.res	PROCESSING
	Complete Invoke not received	Ignore	REQUESTING
S-MethodAbort.req		Decrement N_Methods TR-Abort.req(PEERREQ) the method S-MethodAbort.ind(USERREQ)	NULL
Abort		Decrement N_Methods TR-Abort.req(abort reason) the method S-MethodAbort.ind(abort reason)	NULL
TR-InvokeData.ind	Request Body, MoreData flag set	S-MethodInvokeData.ind(body, MoreData) TR-InvokeData.res	REQUESTING
	Frame Boundary, Data Fragment PDU, MoreData flag cleared	S-MethodInvokeData.ind(headers, body, MoreData)	PROCESSING
	No Frame Boundary, Response Body, MoreData flag cleared	S-MethodInvokeData.ind(body, MoreData)	
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	Decrement N_Methods S-MethodAbort.ind(abort reason)	NULL

Server Method PROCESSING			
Event	**Conditions**	**Action**	**Next State**
S-MethodResult.req	More Data flag set	TR-Result.req(Reply, MoreData)	PROCESSING2
	More Data flag cleared	TR-Result.req(Reply, MoreData)	REPLYING
S-MethodAbort.req		Decrement N_Methods TR-Abort.req(PEERREQ) the method S-MethodAbort.ind(USERREQ)	NULL
Abort		Decrement N_Methods TR-Abort.req(abort reason) the method S-MethodAbort.ind(abort reason)	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	Decrement N_Methods S-MethodAbort.ind(abort reason)	NULL

Server Method PROCESSING2			
Event	**Conditions**	**Action**	**Next State**

Server Method PROCESSING2			
Event	**Conditions**	**Action**	**Next State**
S-MethodResult Data.req	More Data flag set	TR-ResultData.req(Response Body, MoreData)	PROCESSING2
	More Data flag cleared, Response Headers provided	TR-ResultData.req(Data Fragment, , MoreData, Frame Boundary)	REPLYING
	More Data flag cleared, No Response Headers provided	TR-ResultData.req(Response Body, , MoreData)	REPLYING
S-MethodAbort.req		Decrement N_Methods TR-Abort.req(PEERREQ) the method S-MethodAbort.ind(USERREQ)	NULL
Abort		Decrement N_Methods TR-Abort.req(abort reason) the method S-MethodAbort.ind(abort reason)	NULL
TR-Result.cnf		S-MethodResult.cnf	PROCESSING2
TR-ResultData.cnf		S-MethodResultData.cnf	PROCESSING2
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	NULL
	Reason == SUSPEND	*Suspend* the session	
	Other	Decrement N_Methods S-MethodAbort.ind(abort reason)	

Server Method REPLYING			
Event	**Conditions**	**Action**	**Next State**
S-MethodAbort.req		Decrement N_Methods TR-Abort.req(PEERREQ) the method S-MethodAbort.ind(USERREQ)	NULL
Abort		Decrement N_Methods TR-Abort.req(abort reason) the method S-MethodAbort.ind(abort reason)	NULL
TR-Result.cnf	Marks completion of entire result	Decrement N_Methods S-MethodResult.cnf(Acknowledgement Headers = Exit Info) *Note:support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature*	NULL
TR-ResultData.cnf	Marks completion of entire result	Decrement N_Methods S-MethodResult.cnf(Acknowledgement Headers = Exit Info) *Note:support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature*	NULL
	Other	Ignore	REPLYING
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	Decrement N_Methods S-MethodAbort.ind(abort reason)	NULL

7.1.6.6 Server Push State Tables

The following tables show the push states and event processing that occur on the server when using a transaction service.

Server Push NULL			
Event	**Conditions**	**Action**	**Next State**
S-ConfirmedPush.req		TR-Invoke.req(Class 1, ConfirmedPush)	PUSHING

Server Push PUSHING			
Event	**Conditions**	**Action**	**Next State**
Abort		TR-Abort.req(abort reason) the push transaction S-PushAbort.ind(abort reason)	NULL
TR-Invoke.cnf		S-ConfirmedPush.cnf(Acknowledgement Headers = Exit Info) *Note:support for Acknowledgement Headers depends on successful negotiation of the Acknowledgement Headers protocol feature*	NULL
TR-Abort.ind	Reason == DISCONNECT	*Disconnect* the session	
	Reason == SUSPEND	*Suspend* the session	
	Other	S-PushAbort.ind(abort reason)	NULL

7.2 Connectionless WSP

This section is written as if the session service provider is using the Transport SAP directly. However, this section also applies to the use of the Security SAP. There is a one-to-one mapping of connectionless transport primitives [WDP] to Security primitives. For example, T-DUnitdata.request maps directly to SEC-UnitData.request. To allow for this ambiguity, the layer prefixes ("T-D" or "SEC-") have been omitted from the primitive names.

The connectionless WSP protocol does not require state machines. Each primitive of the connectionless WSP service interface maps directly to sending a WSP PDU with the underlying Unitdata primitive as shown in the following table.

Event	Condition	Action
S-Unit-MethodInvoke.req		Unitdata.req(Method) *Note: "Method" means either the Get or Post PDU using the PDU type assigned to the particular method.*
S-Unit-MethodResult.req		Unitdata.req(Reply)
S-Unit-Push.req		Unitdata.req(Push)
T-DError.ind		Ignore
Unitdata.ind	Method PDU *Note: "Method" means either the Get or Post PDU using the PDU type assigned to the particular method.*	S-Unit-MethodInvoke.ind
	Reply PDU	S-Unit-MethodResult.ind
	Push PDU	S-Unit-MethodPush.ind

Protocol parameters, such as the Maximum Receive Unit and the persistent session headers in effect, are defined by mutual agreement between the service users. No particular mechanism for this is required, but the well-known port of the server MAY be used to imply the parameter settings.

8 WSP Data Unit Structure and Encoding

This section describes the structure of the data units used to exchange WSP data units between client and server.

8.1 Data Formats

The following data types are used in the data format definitions.

8.1.1 Primitive Data Types

Table 11. Format Definition Data Types

Data Type	Definition
bit	1 bit of data
octet	8 bits of opaque data
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uintvar	variable length unsigned integer (see below)

Network octet order for multi-octet integer values is "big-endian". In other words, the most significant octet is transmitted on the network first followed subsequently by the less significant octets.

The leftmost bit (bit number 0) of an octet or a bit field is the most significant. Bit fields described first are placed in the most significant bits of an octet. The transmission order in the network is determined by the underlying transport mechanism.

8.1.2 Variable Length Unsigned Integers

Many fields in the data unit formats are of variable length. Typically, there will be an associated field that specifies the size of the variable length field. In order to keep the data unit formats as small as possible, a variable length unsigned integer encoding is used to specify lengths. The larger the unsigned integer, the larger the size of its encoding.

Each octet of the variable length unsigned integer is comprised of a single *Continue* bit and 7 bits of payload as shown in Figure 27.

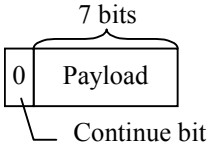

Figure 27. Variable Length Integer Octet

To encode a large unsigned integer, split it into 7-bit fragments and place them in the payloads of multiple octets. The most significant bits are placed in the first octets with the least significant bits ending up in the last octet. All octets MUST set the *Continue* bit to 1 except the last octet, which MUST set the *Continue* bit to 0.

For example, the number 0x87A5 (1000 0111 1010 0101) is encoded in three octets as shown in Figure 28.

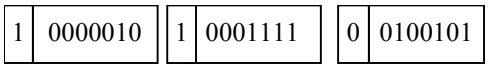

Figure 28. Long Field Length

The unsigned integer MUST be encoded in the smallest encoding possible. In other words, the encoded value MUST NOT start with an octet with the value 0x80.

In the data unit format descriptions, the data type *uintvar* will be used to indicate a variable length integer field. The maximum size of a *uintvar* is 32 bits. It will be encoded in no more than five octets. It MUST be present even if its value is zero.

8.2 Protocol Data Unit Structure

WSP generates WTP SDUs which contain a single WSP protocol data unit. Each PDU serves a particular function in the protocol and contains type-specific information.

8.2.1 PDU Common Fields

This section describes fields that are common across all or many PDUs.

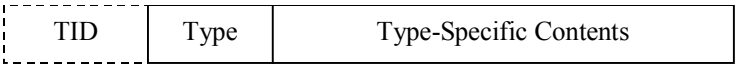

Figure 29. PDU Structure

Every PDU starts with a conditional transaction identifier and a type identifier.

Table 12. PDU Header Fields

Name	Type	Source
TID	uint8	S-Unit-MethodInvoke.req::Transaction Id *or* S-Unit-MethodResult.req::Transaction Id *or* S-Unit-Push.req::Push Id
Type	uint8	PDU type

The *TID* field is used to associate requests with replies in the connectionless session service. The presence of the *TID* is conditional. It MUST be included in the connectionless WSP PDUs, and MUST NOT be present in the connection-mode PDUs. In connectionless WSP, the TID is passed to and from the session user as the “Transaction Id” or “Push Id” parameters of the session primitives.

The *Type* field specifies the type and function of the PDU. The type numbers for the various PDUs are defined in in Assigned Numbers. The rest of the PDU is type-specific information, referred to as the contents.

The following sections describe the format of the contents for each PDU type. In the interest of brevity, the PDU header has been omitted from the description of each PDU in the sections that follow.

8.2.2 Session Management Facility

8.2.2.1 Connect

The *Connect* PDU is sent to initiate the creation of a session.

Table 13. Connect Fields

Name	Type	Source
Version	uint8	WSP protocol version
CapabilitiesLen	uintvar	Length of the *Capabilities* field
HeadersLen	uintvar	Length of the *Headers* field
Capabilities	*CapabilitiesLen* octets	S-Connect.req::Requested Capabilities
Headers	*HeadersLen* octets	S-Connect.req::Client Headers

The *Version* field identifies the version of the WSP protocol. This is used to determine the formats of this and all subsequent PDUs. The version number is encoded as follows: The major number of the version is stored in the high-order 4 bits, and the minor number is stored in the low-order 4 bits. This version number used for this specification is 1.0, ie, 0x10.

The *CapabilitiesLen* field specifies the length of the *Capabilities* field.

The *HeadersLen* field specifies the length of the *Headers* field.

The *Capabilities* field contains encoded capability settings requested by the sender. Each capability has capability-specific parameters associated with it. For more information on the encoding of this field, see section 8.3, “Capability Encoding”, below.

The *Headers* field contains headers sent from client to server that apply to the entire session.

8.2.2.2 ConnectReply

The *ConnectReply* PDU is sent in response to the *Connect* PDU.

Table 14. ConnectReply Fields

Name	Type	Source
ServerSessionId	Uintvar	Session_ID variable
CapabilitiesLen	Uintvar	Length of Capabilities field
HeadersLen	Uintvar	Length of the Headers field
Capabilities	*CapabilitiesLen* octets	S-Connect.res::Negotiated Capabilities
Headers	*HeadersLen* octets	S-Connect.res::Server Headers

The *ServerSessionId* contains the server session identifier. It is used to identify the session in subsequently sent PDUs used for session management. In particular, the client uses this session identifier, if it wants to resume the session after a change in the underlying transport.

The *CapabilitiesLen* field specifies the length of the *Capabilities* field.

The *HeadersLen* field specifies the length of the *Headers* field.

The *Capabilities* field contains zero or more capabilities accepted by the sender. For more information on capabilities, see section 8.3, “Capability Encoding”, below.

The *Headers* field contains headers that apply to the entire session.

8.2.2.3 Redirect

The *Redirect* PDU may be returned in response to a Connect PDU, when the session establishment attempt is refused. It can be used to migrate clients from servers whose addresses have changed or to perform a crude form of load balancing at session creation time.

Table 15. Redirect Fields

Name	Type	Source
Flags	uint8	S-Disconnect.req::Redirect Security *and* S-Disconnect.req::Reason Code
Redirect Addresses	multiple octets	S-Disconnect.req::Redirect Addresses

The *Flags* field indicates the nature of the redirect. Flags that are unassigned MUST be set to 0 by the server and MUST be ignored by the client. The flags are defined as follows:

Flag bit	Description
0x80	Permanent Redirect
0x40	Reuse Security Session

If the *Permanent Redirect* flag is set, the client SHOULD store the redirect addresses and use them to create and resume all future sessions with the server. If the *Permanent Redirect* flag is not set, the client SHOULD not reuse the redirect addresses to create and resume subsequent sessions beyond the current session being created. If the *Reuse Security Session* flag is set, the client MAY use the current security session when requesting a session from the server it is being redirected to.

The *Redirect Addresses* field contains one or more new addresses for the server. Subsequent Connect PDUs should be sent to these addresses instead of the server address, which caused the Redirect PDU to be sent. The length of the *Redirect Addresses* field is determined by the SDU size as reported from the underlying transport. Each redirect address is coded in the following format:

Table 16. AddressType

Name	Type	Purpose
Bearer Type Included	1 bit	Flag indicating inclusion of *Bearer Type* field
PortNumber Included	1 bit	Flag indicating inclusion of *PortNumber* field
Address Len	6 bits	Length of the *Address* field
BearerType	uint8	Type of bearer network to use
PortNumber	uint16	Port number to use
Address	*AddressLen* octets	Bearer address to use

The *BearerType Included* and *PortNumber Included* fields indicate the inclusion of the *BearerType* and *PortNumber* fields, respectively. The *BearerType* and *PortNumber* SHOULD be excluded, if the session establishment attempt is redirected to the same type of bearer network and same destination port number as used for the initial Connect PDU.

The *AddressLen* field contains the length of the *Address* field.

The *BearerType* field indicates the type of bearer network to be used. The bearer type codes are defined in [WDP].

The *PortNumber* field contains the destination port number. When redirecting to a different destination port number than the one used for the initial Connect PDU, the IANA dynamic and/or private ports (49152-65535) MUST be used. The client MUST NOT change protocol layer configuration as a result of a redirection (For example, a client shall not stop using the security layer as a result of a redirection). The *Permanent Redirect* flag MUST not be set when redirecting to a different destination port number than the one used for the initial Connect PDU.

The *Address* field contains the bearer address to use. The *BearerType* implies also the bearer-dependent address format used to encode this field. The encoding shall use the native address transmission format defined in the applicable bearer specifications. If this format uses a number of bits, which is not a multiple of eight, the address shall be encoded as a big-endian multi-octet integer; the necessary number of zero fill bits shall be included in the most significant octet so that the fill bits occupy the most significant bits. The used bearer address formats are defined in [WDP] together with the bearer type codes.

8.2.2.4 Disconnect

The *Disconnect* PDU is sent to terminate a session.

Table 17. Disconnect Fields

Name	Type	Source
ServerSessionId	uintvar	Session_ID variable

The *ServerSessionId* contains the session identifier of the session to be disconnected.

8.2.2.5 Reply

The Reply PDU is used by the session creation facility, and it is defined in section 8.2.3.3, “Reply”, below.

8.2.3 Method Invocation Facility

There are two PDUs used to invoke a method in the server, *Get* and *Post,* depending on the parameters required.

Methods defined in HTTP/1.1 [RFC2616] are assigned a specific PDU type number. PDU type numbers for methods not defined in HTTP/1.1 are established during capability negotiation. These methods use either the *Get* or *Post* PDU depending on whether the method includes request content or not. Methods using *Get* use PDU type numbers in the range 0x40-0x5F. Methods using *Post* use numbers in the range 0x60-0x7F.

8.2.3.1 Get

The *Get* PDU is used for the HTTP/1.1 GET, OPTIONS, HEAD, DELETE and TRACE methods, as well as extension methods that do not send request content to the server.

Table 18. Get Fields

Name	Type	Source
URILen	uintvar	Length of the *URI* field
URI	*URILen* octets	S-MethodInvoke.req::Request URI *or* S-Unit-MethodInvoke.req::Request URI
Headers	multiple octets	S-MethodInvoke.req::Request Headers *or* S-Unit-MethodInvoke.req::Request Headers

The *URILen* field specifies the length of the *URI* field.

The *URI* field contains the URI. If the URI is a normally stored as a null-terminated string, the implementation MUST NOT include the null in the field.

The *Headers* field contains the headers associated with the request. The length of the *Headers* field is determined by the SDU size as provided to and reported from the underlying transport. The *Headers* field starts immediately after the *URI* field and ends at the end of the SDU.

8.2.3.2 Post

The *Post* PDU is used for the HTTP/1.1 POST and PUT methods, as well as extended methods that send request content to the server.

Table 19. Post Fields

Name	Type	Source
UriLen	uintvar	Length of the *URI* field
HeadersLen	uintvar	Length of the *ContentType* and *Headers* fields combined
Uri	*UriLen* octets	S-MethodInvoke.req::Request URI *or* S-Unit-MethodInvoke.req::Request URI
ContentType	multiple octets	S-MethodInvoke.req::Request Headers *or* S-Unit-MethodInvoke.req::Request Headers
Headers	(*HeadersLen* – length of *ContentType*) octets	S-MethodInvoke.req::Request Headers *or* S-Unit-MethodInvoke.req::Request Headers
Data	multiple octets	S-MethodInvoke.req::Request Body *or* S-Unit-MethodInvoke.req::Request Body

The *UriLen* field specifies the length of the Uri field.

The *HeadersLen* field specifies the length of the *ContentType* and *Headers* fields combined.

The *Uri* field contains the Uri. If the URI is a normally stored as a null-terminated string, the implementation MUST NOT include the null in the field.

The *ContentType* field contains the content type of the data. It conforms to the Content-Type value encoding specified in section 8.4.2.24, “Content type field”, below.

The *Headers* field contains the headers associated with the request.

The *Data* field contains the data associated with the request. The length of the *Data* field is determined by the SDU size as provided to and reported from the underlying transport. The *Data* field starts immediately after the *Headers* field and ends at the end of the SDU.

8.2.3.3 Reply

Reply is the generic response PDU used to return information from the server in response to a request. Reply is used in the S-Connect primitive to indicate an error during session creation.

Table 20. Reply Fields

Name	Type	Source
Status	uint8	S-MethodResult.req::Status *or* S-Disconnect.req::Reason Code*or* S-Unit-MethodResult.req::Status
HeadersLen	uintvar	Length of the ContentType and Headers fields combined
ContentType	multiple octets	S-MethodResult.req::Response Headers *or* S-Disconnect.req::Error Headers *or* S-Unit-MethodResult.req::Response Headers
Headers	(*HeadersLen* – length of *ContentType*) octets	S-MethodResult.req::Response Headers *or* S-Disconnect.req::Error Headers *or* S-Unit-MethodResult.req::Response Headers S-Resume.res::Server Headers
Data	multiple octets	S-MethodResult.req::Response Body *or* S-Disconnect.req::Error Body *or* S-Unit-MethodResult.req::Response Body

The *Status* field contains a result code of the attempt to understand and satisfy the request. The status codes have been defined by HTTP/1.1 [RFC2616] and have been mapped into single-octet values listed in Table 36 in Assigned Numbers.

The *HeadersLen* field specifies the length of the *ContentType* and *Headers* fields combined.

The *ContentType* field contains the content type of the data. It conforms to the Content-Type value encoding specified in section 8.4.2.24, “Content type field”, below.

The *Headers* field contains the reply headers or the server headers in case of a Resume.

The *Data* field contains the data returned from the server. The length of the *Data* field is determined by the SDU size as provided to and reported from the underlying transport. The *Data* field starts immediately after the *Headers* field and ends at the end of the SDU.

8.2.3.4 Data Fragment PDU

Data Fragment PDU is used to continue when the last fragment of a request to or response from the server includes headers. This PDU MUST only be sent for an S-MethodInvokeData.req or S-MethodResultData.req where the MoreData flag is cleared and Headers are included. The PDU MUST NOT be sent if the MoreData Flag is set or Headers are not included in the S-MethodInvokeData.req or S-MethodResultData.req service primitives. When a Data Fragment PDU is sent, Frame Boundary MUST be set in the TR-InvokeData.req or TR-ResultData.req as appropriate.

The Data Fragment PDU is used when the service user needs to send headers with the last fragment of a request or response. Currently, the only use case for this is when Trailer Headers need to be sent at the end of a Chunked message [RFC2616].

Table 21. Data Fragment Fields

Name	Type	Source
HeadersLen	Uintvar	Length of the Headers field
Headers	(*HeadersLen e*) octets	S-MethodInvokeData.req::Request Headers, S-MethodResultData.req::Response Headers
Data	multiple octets	S-MethodInvokeData.req::Request Body, S-MethodResultData.req::Response Body

The *HeadersLen* field specifies the length of the *Headers* fields.

The *Headers* field contains request headers in case of a method invocation or the response headers in case of a method result.

The *Data* field contains the data sent as a continuation of the method invocation or method result. The length of the *Data* field is determined by the SDU size as provided to and reported from the underlying transport. The *Data* field starts immediately after the *Headers* field and ends at the end of the SDU.

8.2.3.5 Acknowledgement Headers

Acknowledgement Headers is not an actual PDU: it may be carried by the Exit Info parameter of the TR-Result primitive. The service provider uses it to carry the data needed by the optional Acknowledgement Headers feature.

Table 22. Acknowledgement Headers Fields

Name	Type	Source
Headers	multiple octets	S-MethodResult.res::Acknowledgement Headers *or* S-ConfirmedPush.res::Acknowledgement Headers

The *Headers* field contains information encoded in the manner defined in Section 8.4, “Header Encoding”, below. The size of the field is implied by the size of the transaction Exit Data.

8.2.4 Push and Confirmed Push Facilities

8.2.4.1 Push and ConfirmedPush

The Push and ConfirmedPush PDUs are used for sending unsolicited information from the server to the client. The formats of the two PDUs are the same, only the PDU type is different.

Table 23. Push and ConfirmedPush Fields

Name	Type	Source
HeadersLen	uintvar	Length of the *ContentType* and *Headers* fields combined
ContentType	multiple octets	S-Push.req::Push Headers *or* S-ConfirmedPush.req::Push Headers *or* S-Unit-Push.req::Push Headers
Headers	(*HeadersLen* – length of *ContentType*) octets	S-Push.req::Push Headers *or* S-ConfirmedPush.req::Push Headers *or* S-Unit-Push.req::Push Headers
Data	multiple octets	S-Push.req::Push Body *or* S-ConfirmedPush.req::Push Body *or* S-Unit-Push.req::Push Body

The *HeadersLen* field specifies the length of the *ContentType* and *Headers* fields combined.

The *ContentType* field contains the content type of the data. It conforms to the Content-Type value encoding specified in section 8.4.2.24, "Content type field", below.

The *Headers* field contains the push headers.

The *Data* field contains the data pushed from the server. The length of the *Data* field is determined by the SDU size as provided to and reported from the underlying transport. The *Data* field starts immediately after the *Headers* field and ends at the end of the SDU.

8.2.4.2 Acknowledgement Headers

If the service provider implements the optional Acknowledgement Headers feature with the Confirmed Push facility, *Acknowledgement Headers* are used to carry the associated data. It is defined in Section 8.2.3.5 above.

8.2.5 Session Resume Facility

8.2.5.1 Suspend

The *Suspend* PDU is sent to suspend a session.

Table 24. Suspend Fields

Name	Type	Source
SessionId	Uintvar	Session_ID variable

The *SessionId* field contains the session identifier of the session to be suspended.

8.2.5.2 Resume

The *Resume* PDU is sent to resume an existing session after a change in the underlying transport protocol.

Table 25. Resume Fields

Name	Type	Purpose
SessionId	uintvar	Session_ID variable
Capabilities Len	uintvar	Length of the *Capabilities* field
Capabilities	*CapabilitiesLen* octets	Reserved
Headers	multiple octets	S-Resume.req::Client Headers

The *SessionId* field contains the session identifier returned from the server when the session was originally created. The server looks up the session based on the session identifier. It then binds that session to the transaction service instance identified by the peer address quadruplet of the transaction that carried the PDU.

The Capabilities field is reserved for future use. In this version of the protocol it must not be used and the CapabilitiesLen field must be zero.

The *Headers* field contains headers sent from client to server that apply to the entire session.

8.2.5.3 Reply

The Reply PDU is used by the session resume facility, and it is defined in section 8.2.3.3, “Reply”, above

8.3 Capability Encoding

Capabilities allow the client and server to negotiate characteristics and extended behaviours of the protocol. A general capability format is defined so capabilities that are not understood can be ignored.

A set of capability values is encoded as a sequence of capability structures described below. If the sender wants to provide the receiver with a set of alternative values for a particular capability, one of which can be chosen, it sends multiple instances of the capability, each with different parameters and with the most preferred alternative first. A responder must not encode and send the value of a capability, unless the initiator is known to recognise it, as indicated by either the version number of the session protocol or by the initiator already having sent that capability during the session.

When the initiator of capability negotiation encodes a capability defined in Section 8.3.2 "Capability Definitions", below, and the value is equal to the capability setting (default or negotiated) currently in effect, the capability structure MAY be omitted. In this case the responder MUST interpret this in the same way, as if it had received the explicitly encoded value. When the responder encodes a capability defined in Section 8.3.2 "Capability Definitions", and the value is equal to the capability setting proposed by the initiator, the capability structure MAY be omitted; the initiator MUST interpret this in the same way, as if it had received the explicitly encoded value.

8.3.1 Capability Structure

The format of a capability is described using a table similar to the ones used in PDU definitions:

Table 26. Capability Fields

Name	Type	Purpose
Length	uintvar	Length of the *Identifier* and *Parameters* fields combined
Identifier	multiple octets	Capability identifier
Parameters	(*Length* – length of *Identifier*) octets	Capability-specific parameters

The *Length* field specifies the length of the *Identifier* and *Parameters* fields combined.

The *Identifier* field identifies the capability. The capability identifier values defined in this protocol version are listed in Table 37 in Assigned Numbers. It is encoded in the same way as the header field names, ie, using the *Field-name* BNF rule specified in Section 8.4.2.6, "Header", below.

The *Parameters* field (if not empty) contains capability-specific parameters.

If a capability with an unknown *Identifier* field is received during capability negotiation, its value must be ignored. The responder must also reply with the same capability with an empty *Parameters* field, which indicates that the capability was not recognised and did not have any effect. As a consequence, the encodings for any provider-specific additional capabilities MUST BE chosen so that an empty *Parameters* field either is illegal (as for capabilities with integer values) or indicates that no extended functionality is enabled.

8.3.2 Capability Definitions

8.3.2.1 Service Data Unit Size

There are two Service Data Unit (SDU) size capabilities, one for the client and one for the server:

- Client-SDU-Size
- Server-SDU-Size

These capabilities share the same parameter format.

Table 27. SDU Size Capability Fields

Name	*Type*	*Purpose*
MaxSize	uintvar	Maximum Size

The *MaxSize* field specifies the maximum SDU size that can be received or will be sent by the client or server, depending on the context of the capability, as described below. A *MaxSize* of 0 (zero) means there is no limit to the SDU size.

When the client sends the Client-SDU-Size capability, it is indicating the maximum size SDU it can receive (ie, the client MRU). When the server sends the Client-SDU-Size capability, it is indicating the maximum SDU size it will send.

When the client sends the Server-SDU-Size capability, it is indicating the maximum size SDU it will send. When the server sends the Server-SDU-Size capability, it is indicating the maximum SDU size it can receive (ie, the server MRU).

The default SDU sizes are specified in section 8.3.3, "Capability Defaults", below. The default Server SDU size SHOULD be treated as an implementation minimum. Otherwise a method request sent during session establishment would risk being aborted, since the server cannot indicate its true MRU until session has been established.

8.3.2.2 Message Size

There are two message size capabilities, one for the client and one for the server:

- Client-Message-Size
- Server-Message-Size

These capabilities share the same parameter format.

Table 28. Message Size Capability Fields

Name	*Type*	*Purpose*
MaxSize	uintvar	Maximum Size

The *MaxSize* field specifies the maximum message size that can be received or will be sent by the client or server, depending on the context of the capability, as described below. A *MaxSize* of 0 (zero) means there is no limit to the message size.

When the client sends the Client-Message-Size capability, it is indicating the maximum size message it can receive. When the server sends the Client-Message-Size capability, it is indicating the maximum message size it will send.

When the client sends the Server-Message-Size capability, it is indicating the maximum size message it will send. When the server sends the Server-Message-Size capability, it is indicating the maximum message size it can receive.

The default message sizes are specified in section 8.3.3, “Capability Defaults”. The default Server message size SHOULD be treated as an implementation minimum.

8.3.2.3 Protocol Options

The Protocol Options capability is used to enable extended, optional protocol functions.

Table 29. Protocol Options Capability Fields

Name	**Type**	**Purpose**
Flags	multiple octets	Option flags

When the client sends the Protocol Options capability to the server, the *Flags* field specifies the options the client will accept. When the server sends the Protocol Options capability back to the client, the *Flags* field specifies the options the server will perform. Although the *Flags* field may be multiple octets long, the currently defined flag bits fit into a single octet, and an implementation SHOULD send only one octet. All undefined bits must be set to zero, and the receiver MUST ignore them, including all additional trailing octets. As more flag bits are defined in the future, new octets can then be appended to the field.

A flag bit set to one (1) indicates that the associated optional function is enabled; a flag bit cleared to zero (0) indicates that it is disabled. The flags are defined as follows:

Flag bit	**Description**
0x80	Confirmed Push Facility
0x40	Push Facility
0x20	Session Resume Facility
0x10	Acknowledgement Headers
0x08	Large Data Transfer

When the client enables the Confirmed Push and/or Push facilities, it is advertising that it is able to and also wants to accept data pushes. If the client can receive data pushes, but the service provider in the server cannot send pushes, the appropriate push flags MUST be cleared when replying with the negotiated capabilities. If the service user in the server will not send any

data pushes of a certain type, the appropriate push flag SHOULD be cleared in the reply: this will allow the client to free up any resources that would otherwise be dedicated to receiving data pushes.

When the client enables the Session Resume facility, it is advertising that it would like to suspend and resume the session. If the server is not able or willing to support the Session Resume facility, it MUST clear the Session Resume facility flags when replying with the negotiated capabilities.

When the client sets the Acknowledgement Headers flag, it is advertising whether or not it would like to send Acknowledgement headers. The server indicates with the Acknowledgement Headers flag in the reply, whether or not it is able to process Acknowledgement Headers. If the server is not able to process the headers, the client SHOULD not send them; if the client still sends them, the headers shall be ignored.

When the client sets the Large Data Transfer flag, it is advertising whether or not it supports the Large Data Transfer feature; this includes support for multiple SDUs and support for the Data Fragment PDU. The server indicates with the Large Data Transfer flag in the reply, whether or not it supports the Large Data Transfer feature. Note that setting this flag indicates that sending and receiving large data is supported; a client that only wishes to receive large data (i.e. not send) SHOULD set the Client and Server Message size capability fields accordingly.

8.3.2.4 Maximum Outstanding Requests (MOR)

There are two MOR capabilities, one for methods and one for pushes:

- Method-MOR
- Push-MOR

The Method-MOR and Push-MOR capabilities respectively indicate the number of outstanding method or push transactions that may occur simultaneously.

Table 30. Maximum Outstanding Requests Capability Fields

Name	Type	Purpose
MOR	uint8	Maximum Outstanding Requests

When the client is able to submit multiple outstanding method requests, it indicates the maximum number of simultaneous requests it will ever send in the Method-MOR capability. The server replies with the lesser of the client's Method-MOR and the number of method transactions the server can simultaneously process.

Similarly, when the client is able to process multiple outstanding push requests, it indicates the maximum number of simultaneous requests it can process in the Push-MOR capability. The server replies with the lesser of the client's Push-MOR and the maximum number of simultaneous push transactions the server will ever send.

8.3.2.5 Extended Methods

The Extended Methods capability declares the set of extended methods to be used during the session and assigns PDU types to them.

Table 31. Extended Methods Capability Field Entries

Name	Type	Purpose
PDU Type	uint8	PDU Type for method
Method Name	multiple octets	Null terminated method name

When sent from client to server in the Connect PDU, the capability-specific parameters for the Extended Methods capability contain zero or more *PDU Type* to *Method Name* assignments. The end of the list of assignments is determined from the end of the capability as specified in the capability length. Each capability assignment contains a *PDU Type* and a *Method Name*.

The PDU types are assigned by the client from the range 0x50-0x5F for methods that use the Get PDU format and the range 0x70-0x7F for methods that use the Post PDU format. The method name is a null terminated string.

When sent from server to client in the ConnectReply PDU, the capability-specific parameters for the Extended Methods capability contain the zero or more PDU type codes (without the method names) that the server accepts and can receive.

8.3.2.6 Header Code Pages

The Header Code Pages capability declares the set of header code pages to be used during the session and assigns page codes to them.

Table 32. Header Code Pages Capability Field Entries

Name	Type	Purpose
Page Code	uint8	Code for header page
Page Name	multiple octets	Name of header page

When sent from client to server in the Connect PDU, the capability-specific parameters for the Header Code Pages capability contain zero or more header page name to code assignments. The end of the list of assignments is determined from the end of the capability as specified in the capability length. Each capability assignment contains a *Page Code* and a *Page Name*. The *Page Name* is a null terminated string.

When sent from server to client in the ConnectReply PDU, the capability-specific parameters for the Header Code Pages capability contain the zero or more *Page Codes* (without the *Page Names*), that the server can and will use.

When the client sends this capability, it is indicating its desire to use the named header code pages. The response from the server indicates, which of these pages actually shall be used during the remainder of the session. Once the use of an extension header code page has been negotiated, the headers belonging to it MUST be sent encoded using the binary syntax defined by the code page. If the server declines to use a particular header code page, the (application-specific) headers MUST be sent in textual format, unless some other code page defines an encoding syntax for them.

If the server agrees to use a header code page, the *Page Code* selected by the client shall be used during the remainder of the session, when the header code page needs to be identified in a code page shift sequence.

8.3.2.7 Aliases

The Aliases capability declares a list of alternate addresses for the sender.

Table 33. Aliases Capability Fields

Name	Type	Purpose
Addresses	multiple octets	Alternate addresses

The *Addresses* field is encoded in the same format as the *Redirect Addresses* field in the Redirect PDU, described in Section 8.2.2.3. The addresses sent by a server may be used to facilitate a switch to an alternate bearer network, when a session is resumed. The addresses sent by a client may be used to facilitate the use of the connectionless session service.

8.3.3 Capability Defaults

Unless otherwise specified for a specific bearer or well-known application port, the capability defaults are as follows:

Name	Setting
Aliases	*None*
Client SDU Size	1400 octets
Extended Methods	*None*
Header Code Pages	*None*
Protocol Options	0x00
Maximum Outstanding Method Requests	1
Maximum Outstanding Push Requests	1
Server SDU Size	1400 octets
Client Message Size	1400 octets
Server Message Size	1400 octets

8.4 Header Encoding

8.4.1 General

In this section, both sender and recipient refer to the peer entities (client or server), terminating the WSP protocol. Each peer entity is always associated with an encoding version. The encoding version indicates which encodings that are recognized by a peer. However even though an encoding is recognized, it does not imply that the peer supports the functionality associated with the encoding. For example, a server can recognize a Profile header and thereby indicate support for encoding version 1.2. However, the server might not have support for User Agent Profile. The supported encoding version is exposed to the other peer entity during the first transaction or by means of Client Provisioning [PROVCONT]. In addition all new headers defined by WAP Forum MUST have well defined rules for encoding headers in text format as well in binary format. For end-to-end headers the encoding defined for HTTP MUST be applied as the textual encoding.

WSP header fields are included in WSP PDUs or in multi-part data objects. The header fields contain general information, request information, response information, or entity information. Each header field consists of a field name followed by a field value.

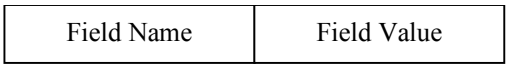

Figure 30. Header field comprised of field name and field value

WSP defines a compact format for encoding header fields that is compatible with HTTP/1.1 header fields.

The following procedures are used to reduce the size of the headers:

- Well-known tokens are mapped to binary values.
- Date values, integer values, quality factors and delta second values are coded in binary format.
- Redundant information is removed.

The encoding utilises the fact that the first octet of the text strings in HTTP headers is typically in the range 32-126, except for some rare cases when a text string is initiated with an 8-bit character value (eg, national characters). Range 0-31 and 127-255 can be used for binary values, quote characters or length indicators of binary data. This makes it possible to mix binary data and text strings efficiently, which is an advantage when the generic parts of HTTP/1.1 headers shall be encoded.

8.4.1.1 Field name

Field names with assigned integer encoding values MUST be encoded using the integer value if encoding version associated with the field name is less or equal to 1.2 or if the recipient supports the encoding version associated with field name. If the encoding version is higher than the encoding version supported by the recipient or if the client encoding version is unknown by the server, the field name MUST be sent in text format. If the encoding version of the server is unknown by the client, the field name MAY be sent in text format or in binary format. Unrecognized encodings MUST be handled as defined in section 8.4.2.70. The representation of the integer encodings is made more compact by dividing them into *header code pages*. Each header code page encodes up to 128 identities of well-known field names, so that the integer encoding value is represented using a single octet. The most common well-known header names are defined in the default header code page, but additional encoding values can be made available by shifting between code pages.

The header code pages used during a session are identified with numeric codes. Header code page 1 is the default page and is always active at the beginning of a set of headers. A shift to a new code page is accomplished by sending a *shift sequence* between two header fields. The new header code page remains active until the end of the set of headers being decoded. This procedure applies to the header fields in each WSP PDUs, as well as to the header fields of each entity embedded in a multipart entity.

The default header code pages defines all HTTP/1.1 field names and header fields specified by the WAP Forum. The numbers for header code pages are assigned in the following way:

- 1, default header code page, including HTTP/1.1 headers and headers specified by the WAP Forum
- 2-15, reserved for header code pages specified by the WAP Forum
- 16-127, reserved for application specific code pages
- 128-255, reserved for future use

An application-specific header code page is identified by a textual name (string). However, when capability negotiation is used to agree on the set of extension header code pages (see Section 8.3.2.6), which shall be used during the session, each application-specific code page is also assigned a numeric identity from the range reserved for them. This identity remains in effect to the end of the session and MUST be used to identify the page in a shift sequence.

If capability negotiation leads to an agreement on the use of a header code page, then the application-specific field names MUST be sent using the well-known single-octet values defined by the page. If there is no agreement on the use of a header code page, the application-specific field names MUST be encoded using the *Token-text* rule from Section 8.4.2.1 below.

For example, a sequence of well-known headers and application specific header can be structured as follows:

```
<WSP header 1>
.
.
<WSP header n>
<Shift to application specific code page>
<Application specific header 1>
.
.
<Application specific header m>
```

8.4.1.2 Field values

The syntax of encoded field values is defined by the field name. Well-known field values MUST be encoded using the compact binary formats defined by the header syntax below. If the field name is encoded in text format, textual values MUST be used. The WSP field values are encoded so that the length of the field value can always be determined, even if the detailed format of a specific field value is not known. This makes it possible to skip over individual header fields without interpreting their content. The header syntax in Section 8.4.2 below is defined so, that the first octet in all the field values can be interpreted as follows:

Value	Interpretation of First Octet
0 - 30	This octet is followed by the indicated number (0 –30) of data octets

31	This octet is followed by a **uintvar**, which indicates the number of data octets after it
32 - 127	The value is a text string, terminated by a zero octet (NUL character)
128 - 255	It is an encoded 7-bit value; this header has no more data

It is up to the application to define how application-specific field values shall be encoded, but the encodings MUST adhere to the general format described in the table above.

If there is a mutual agreement between server and client on the used extension header code pages, then there is also a mutual agreement on, how application-specific field values defined by these code pages shall be encoded. In this case the applicable field values MUST be encoded according to the syntax rules defined by these code pages.

If the client and server cannot agree on the use of a header code page during capability negotiation, application-specific field values MUST be encoded using the *Application-specific-value* rule from Section 8.4.2.6.

8.4.1.3 Encoding of list values

If the syntax defined by RFC2616 for a header field with a well-known field name permits a comma-separated list using 1#*rule*, the header MUST be converted into a sequence of headers. Each shall have the original field name and contain one of the values in the original list. The order of the headers shall be the same as the order of their values in the original list value. The encoding rule for the well-known header shall be applied only after this transformation.

8.4.2 Header syntax

This section defines the syntax and semantics of all HTTP/1.1 header fields in WSP. The mechanisms specified in this document are described in augmented BNF similar to that used by [RFC2616].

The notation <Octet N> is used to represent a single octet with the value *N* in the decimal system. The notation <Any octet M-N> is used for a single octet with the value in the range from *M* to *N*, inclusive.

8.4.2.1 Basic rules

The following rules are used through this specification to describe the basic parsing constructs. The rules for Token, TEXT and OCTET have the same definition as per [RFC2616].

```
Text-string = [Quote] *TEXT End-of-string
; If the first character in the TEXT is in the range of 128-255,  a Quote character must precede it.
; Otherwise the Quote character must be omitted. The Quote is not part of the contents.

Token-text = Token End-of-string

Quoted-string = <Octet 34> *TEXT End-of-string
;The TEXT encodes an RFC2616 Quoted-string with the enclosing quotation-marks <"> removed

Extension-media = *TEXT End-of-string
; This encoding is used for media values, which have no well-known binary encoding

Short-integer = OCTET
; Integers in range 0-127 shall be encoded as a one octet value with the most  significant bit set
; to one (1xxx xxxx) and with the value in the remaining least significant bits.

Long-integer = Short-length Multi-octet-integer
; The Short-length indicates the length of the Multi-octet-integer

Multi-octet-integer = 1*30 OCTET
; The content octets shall be an unsigned integer value
; with the most significant octet encoded first (big-endian representation).
; The minimum number of octets must be used to encode the value.
```

Uintvar-integer = 1*5 OCTET
; The encoding is the same as the one defined for **uintvar** in Section 8.1.2.

Constrained-encoding = Extension-Media | Short-integer
; This encoding is used for token values, which have no well-known binary encoding, or when
; the assigned number of the well-known encoding is small enough to fit into Short-integer.

Quote = <Octet 127>
End-of-string = <Octet 0>

8.4.2.2 Length

The following rules are used to encode length indicators.

Value-length = Short-length | (Length-quote Length)
; Value length is used to indicate the length of the value to follow

Short-length = <Any octet 0-30>

Length-quote = <Octet 31>

Length = Uintvar-integer

8.4.2.3 Parameter Values

The following rules are used in encoding parameter values.

No-value = <Octet 0>
; Used to indicate that the parameter actually has no value,
; eg, as the parameter "bar" in ";foo=xxx; bar; baz=xyzzy".

Text-value = No-value | Token-text | Quoted-string

Integer-Value = Short-integer | Long-integer

Date-value = Long-integer
; The encoding of dates shall be done in number of seconds from
; 1970-01-01, 00:00:00 GMT.

Delta-seconds-value = Integer-value

Q-value = 1*2 OCTET
; The encoding is the same as in Uintvar-integer, but with restricted size. When quality factor 0
; and quality factors with one or two decimal digits are encoded, they shall be multiplied by 100
; and incremented by one, so that they encode as a one-octet value in range 1-100,
; ie, 0.1 is encoded as 11 (0x0B) and 0.99 encoded as 100 (0x64). Three decimal quality
; factors shall be multiplied with 1000 and incremented by 100, and the result shall be encoded
; as a one-octet or two-octet uintvar, eg, 0.333 shall be encoded as 0x83 0x31.
; Quality factor 1 is the default value and shall never be sent.

Version-value = Short-integer | Text-string
; The three most significant bits of the Short-integer value are interpreted to encode a major
; version number in the range 1-7, and the four least significant bits contain a minor version
; number in the range 0-14. If there is only a major version number, this is encoded by
; placing the value 15 in the four least significant bits. If the version to be encoded fits these
; constraints, a Short-integer must be used, otherwise a Text-string shall be used.

Uri-value = Text-string
; URI value should be encoded per [RFC2616], but service user may use a different format.

8.4.2.4 Parameter

The following rules are used to encode parameters.

```
Parameter = Typed-parameter | Untyped-parameter

Typed-parameter =  Well-known-parameter-token Typed-value
; the actual expected type of the value is implied by the well-known parameter
Well-known-parameter-token = Integer-value
; the code values used for parameters are specified in the Assigned Numbers appendix
Typed-value = Compact-value | Text-value
; In addition to the expected type, there may be no value.
; If the value cannot be encoded using the expected type, it shall be encoded as text.

Compact-value = Integer-value |
                Date-value | Delta-seconds-value | Q-value | Version-value |
                Uri-value
Untyped-parameter = Token-text Untyped-value
; the type of the value is unknown, but it shall be encoded as an integer, if that is possible.

Untyped-value = Integer-value | Text-value
```

8.4.2.5 Authorization

The following common rules are used for authentication and authorisation.

```
Credentials = ( Basic Basic-cookie ) | ( Authentication-scheme *Auth-param )
Basic = <Octet 128>
Basic-cookie = User-id Password
User-id = Text-string
Password = Text-string
; Note user identity and password shall not be base 64 encoded.
Authentication-scheme = Token-text
Auth-param = Parameter
Challenge = ( Basic Realm-value ) | ( Authentication-scheme Realm-value *Auth-param )
Realm-value = Text-string
; shall be encoded without the quote characters <"> in the corresponding RFC2616 Quoted-string
```

8.4.2.6 Header

The following rules are used to encode headers.

```
Header = Message-header | Shift-sequence
Shift-sequence = ( Shift-delimiter Page-identity ) | Short-cut-shift-delimiter
Shift-delimiter = <Octet 127>
Page-identity = <Any octet 1-255>
Short-cut-shift-delimiter = <Any octet 1-31>

Message-header =  Well-known-header | Application-header

Well-known-header = Well-known-field-name Wap-value

Application-header = Token-text Application-specific-value

Field-name = Token-text | Well-known-field-name
Well-known-field-name = Short-integer
Application-specific-value = Text-string
```

Wap-value =
Accept-value |
Accept-charset-value |
Accept-encoding-value |
Accept-language-value |
Accept-ranges-value |
Age-value |
Allow-value |
Authorization-value |
Cache-control-value |
Connection-value |
Content-encoding-value |
Content-language-value |
Content-length-value |
Content-location-value |
Content-MD5-value |
Content-range-value |
Content-type-value |
Date |
Etag-value |
Expires-value |
From-value |
Host-value |
If-modified-since-value |
If-match-value |
If-none-match-value |
If-range-value |
If-unmodified-since-value |
Location-value |
Last-modified |
Max-forwards-value |
Pragma-value |
Proxy-authenticate-value |
Proxy-authorization-value |
Public-value |
Range-value |
Referer-value |
Retry-after-value |
Server-value |
Transfer-encoding-value |
Upgrade-value |
User-agent-value |
Vary-value |
Via-value |
Warning |
WWW-authenticate-value |
Content-disposition-value |
Application-id-value |
Content-uri-value |
Initiator-uri-value |
Accept-application-value |
Bearer-indication-value |
Push-flag-value |
Profile-value |
Profile-diff-value |
Profile-warning-value |
Expect-value |
TE-value |
Trailer-value |
X-Wap-Tod-value |
Content-ID-value |
Set-Cookie-value |
Cookie-value |
Encoding-Version-value |

X-WAP-Security |
X-Wap-Loc-Invocation-value |
X-Wap-Loc-Delivery-value;

8.4.2.7 Accept field

The following rules are used to encode accept values.

Accept-value = Constrained-media | Accept-general-form

Accept-general-form = Value-length Media-range [Accept-parameters]

Media-range = (Well-known-media | Extension-Media) *(Parameter)

Accept-parameters = Q-token Q-value *(Accept-extension)

Accept-extension = Parameter

Constrained-media = Constrained-encoding
Well-known-media = Integer-value
; Both are encoded using values from Content Type Assignments table in Assigned Numbers

Q-token = <Octet 128>

8.4.2.8 Accept charset field

The following rules are used to encode accept character set values.

Accept-charset-value = Constrained-charset | Accept-charset-general-form

Accept-charset-general-form = Value-length (Well-known-charset | Token-text) [Q-value]

Constrained-charset = Any-charset | Constrained-encoding
Well-known-charset = Any-charset | Integer-value
; Both are encoded using values from Character Set Assignments table in Assigned Numbers

Any-charset = <Octet 128>
; Equivalent to the special RFC2616 charset value “*”

8.4.2.9 Accept encoding field

The following rules are used to encode accept encoding values.

Accept-encoding-value = Content-encoding-value | Accept-encoding-general-form

Accept-encoding-general-form = Value-length (Content-encoding-value | Any-encoding) [Q-value]

Any-encoding = <Octet 131>
; Equivalent to the special RFC2616 encoding value “*”

8.4.2.10 Accept language field

The following rules are used to encode accept language values.

Accept-language-value = Constrained-language | Accept-language-general-form

Accept-language-general-form = Value-length (Well-known-language | Text-string) [Q-value]

Constrained-language = Any-language | Constrained-encoding
Well-known-language = Any-language | Integer-value
; Both are encoded using values from ISO 639 Language Assignments table in Assigned Numbers

Any-language = <Octet 128>
; Equivalent to the special RFC2616 language range "*"

8.4.2.11 Accept ranges field

The following rules are used to encode accept range values.

Accept-ranges-value = (None | Bytes | Token-text)

None = <Octet 128>
Bytes = <Octet 129>

8.4.2.12 Age field

The following rule is used to encode age values.

Age-value = Delta-seconds-value

8.4.2.13 Allow field

The following rules are used to encode allow values.

Allow-value = Well-known-method

Well-known-method = Short-integer
; Any well-known method or extended method in the range of 0x40-0x7F

8.4.2.14 Authorization field

The following rule is used to encode authorisation values.

Authorization-value = Value-length Credentials

8.4.2.15 Cache-control field

The following rules are used to encode cache control values.

```
Cache-control-value =       No-cache |
                            No-store |
                            Max-stale |
                            Only-if-cached |
                            Private |
                            Public |
                            No-transform |
                            Must-revalidate |
                            Proxy-revalidate |
                            Cache-extension |
                            Value-length Cache-directive
Cache-directive =   No-cache 1*(Field-name) |
                    Max-age Delta-second-value |
                    Max-stale Delta-second-value |
                    Min-fresh Delta-second-value |
                    Private 1*(Field-name) |
                    S-maxage Delta-second-value |
                    Cache-extension Untyped-value
No-cache = <Octet 128>
No-store = <Octet 129>
Max-age = <Octet 130>
Max-stale = <Octet 131>
Min-fresh = <Octet 132>
Only-if-cached = <Octet 133>
Public = <Octet 134>
Private = <Octet 135>
No-transform = <Octet 136>
Must-revalidate = <Octet 137>
Proxy-revalidate = <Octet 138>
```

S-maxage = <Octet 139>
Cache-extension = Token-text

8.4.2.16 Connection field

The following rules are used to encode connection values.

Connection-value = (Close | Token-text)

Close = <Octet 128>

8.4.2.17 Content-base field

This field is deprecated.

8.4.2.18 Content encoding field

The following rules are used to encode content encoding values.

Content-encoding-value = (Gzip | Compress | Deflate | Token-text)

Gzip = <Octet 128>
Compress = <Octet 129>
Deflate = <Octet 130>

8.4.2.19 Content language field

The following rule is used to encode content language values.

Content-language-value = (Well-known-language | Token-text)

8.4.2.20 Content length field

The following rule is used to encode content length values. Normally the information in the content length header is redundant and MAY not be sent -- the content length is available in the PDU or can be calculated when the transport layer provides the PDU size.

If the PDU contains no entity body at all (response to HEAD), then the Content-Length SHOULD be encoded in the header fields, so that the client can learn the size of the entity.

Content-length-value = Integer-value

8.4.2.21 Content location field

The following rule is used to encode content location values.

Content-location-value = Uri-value

8.4.2.22 Content MD5 field

The following rules are used to encode content MD5 values.

Content-MD5-value = Value-length Digest
; 128-bit MD5 digest as per [RFC1864]. Note the digest shall not be base-64 encoded.
Digest = 16*16 OCTET

8.4.2.23 Content range field

The following rules are used to encode content range values. Last-byte-pos available in the HTTP/1.1 header is redundant. The content range length is available in the PDU or can be calculated when the transport layer provides the PDU size. Last-byte-pos can be calculated by adding together First-byte-pos with size of content range.

```
Content-range = Value-length First-byte-pos Entity-length
First-byte-pos = Uintvar-integer
Entity-length = Unknown-length | Uintvar-integer
Unknown-length = <Octet 128>
; Equivalent to the special RFC 2616 Content-Range entity-length "*"
```

A server sending a response 416 (Requested range not satisfiable), should encode the First-byte-pos with a value of zero (0). The client should ignore the value of the First-byte-pos, if status code is 416.

8.4.2.24 Content type field

The following rules are used to encode content type values. The short form of the Content-type-value MUST only be used when the well-known media is in the range of 0-127 or a text string. In all other cases the general form MUST be used.

```
Content-type-value = Constrained-media | Content-general-form
Content-general-form = Value-length Media-type
Media-type = (Well-known-media | Extension-Media) *(Parameter)
```

Note that the value of the content type field must be placed in the PDU Content-Type field and therefore the header itself shall not be transmitted. Similarly on reception, the value in the PDU Content Type field must be passed to the application as a Content-Type header.

8.4.2.25 Date field

The following rule is used to encode date values.

```
Date = Date-value
```

8.4.2.26 Etag field

The following rule is used to encode entity tag values.

```
Etag-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.27 Expires field

The following rule is used to encode expires values.

```
Expires-value = Date-value
```

8.4.2.28 From field

The following rule is used to encode from values.

```
From-value = Text-string
; The value shall be encoded as an e-mail address as per [RFC822]
```

8.4.2.29 Host field

The following rule is used to encode host values.

```
Host-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.30 If modified since field

The following rule is used to encode if modified since values.

```
If-modified-since-value = Date-value
```

8.4.2.31 If match field

The following rule is used to encode if match values.

```
If-match-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.32 If none match field

The following rule is used to encode if none match values.

```
If-none-match-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.33 If range field

The following rule is used to encode if range values.

```
If-range = Text-string | Date-value
; The value shall be encoded as per [RFC2616]
```

8.4.2.34 If unmodified since field

The following rule is used to encode if unmodified since values.

```
If-unmodified-since-value = Date-value
```

8.4.2.35 Last modified field

The following rule is used to encode last modified values.

```
Last-modified-value = Date-value
```

8.4.2.36 Location field

The following rule is used to encode location values.

```
Location-value = Uri-value
```

8.4.2.37 Max forwards field

The following rule is used to encode max forwards values.

```
Max-forwards-value = Integer-value
```

8.4.2.38 Pragma field

The following rule is used to encode pragma values.

Pragma-value = No-cache | (Value-length Parameter)
; The quoted text string shall be encoded as per [RFC2616]

8.4.2.39 Proxy-authenticate

The following rules are used to encode proxy authenticate values.

Proxy-authenticate-value = Value-length Challenge

8.4.2.40 Proxy authorization field

The following rules are used to encode proxy authorization values.

Proxy-authorization-value = Value-length Credentials

8.4.2.41 Public field

The following rule is used to encode public values.

Public-value = (Well-known-method | Token-text)

8.4.2.42 Range field

The following rules are used to encode range values.

Range-value = Value-Length (Byte-range-spec | Suffix-byte-range-spec)

Byte-range-spec = Byte-range First-byte-pos [Last-byte-Pos]

Suffix-byte-range-spec = Suffix-byte-range Suffix-length

;First-byte-pos as defined in section 8.4.2.23

Last-byte-pos = Uintvar-integer

Suffix-length = Uintvar-integer

Byte-range = <Octet 128>
Suffix-byte-range = <Octet 129>

8.4.2.43 Referer field

The following rule is used to encode referrer values.

Referer-value = Uri-value

8.4.2.44 Retry after field

The following rules are used to encode retry after values.

Retry-after-value = Value-length (Retry-date-value | Retry-delta-seconds)

Retry-date-value = Absolute-time Date-value

Retry-delta-seconds = Relative-time Delta-seconds-value

Absolute-time = <Octet 128>
Relative-time = <Octet 129>

8.4.2.45 Server field

The following rule is used to encode server values.

Server-value = Text-string
; The value shall be encoded as per [RFC2616]

8.4.2.46 Transfer encoding field

The following rules are used to encode transfer-encoding values.

```
Transfer-encoding-values = Chunked | Token-text
Chunked = <Octet 128>
```

8.4.2.47 Upgrade field

The following rule is used to encode upgrade values.

```
Upgrade-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.48 User agent field

The following rule is used to encode user agent values.

```
User-agent-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.49 Vary field

The following rule is used to encode vary values.

```
Vary-value = Field-name
```

8.4.2.50 Via field

The following rule is used to encode via values.

```
Via-value = Text-string
; The value shall be encoded as per [RFC2616]
```

8.4.2.51 Warning field

The following rules are used to encode warning values. The warning code values are defined in [RFC2616].

```
Warning = Warn-code | Warning-value
Warning-value = Value-length Warn-code Warn-agent Warn-text
Warn-code = Short-integer
; The code values used for warning codes are specified in the Assigned Numbers appendix
Warn-agent = Text-string
; The value shall be encoded as per [RFC2616]
Warn-text = Text-string
```

8.4.2.52 WWW-authenticate field

The following rule is used to encode WWW authenticate values.

```
WWW-authenticate-value = Value-length Challenge
```

8.4.2.53 Content-disposition field

The following rule is used to encode the Content-disposition fields used when submitting form data.

```
Content-disposition-value = Value-length Disposition *(Parameter)
Disposition = Form-data | Attachment  | Inline | Token-text

Form-data = <Octet 128>
Attachment = <Octet 129>
Inline = <Octet 130>
```

8.4.2.54 X-Wap-Application-Id field

The following rule is used to encode the X-Wap-Application-Id field.

```
Application-id-value = Uri-value | App-assigned-code
App-assigned-code = Integer-value
```

8.4.2.55 X-Wap-Content-URI field

The following rule is used to encode the X-Wap-Content-URI field.

```
Content-uri-value = Uri-value
```

8.4.2.56 X-Wap-Initiator-URI field

The following rule is used to encode the X-Wap-Initiator-URI field.

```
Initiator-uri-value = Uri-value
```

8.4.2.57 Accept-application field

The following rule is used to encode the Accept-application field.

```
Accept-application-value = Any-application | Application-id-value
; lists of Application Id values encoded using multiple Accept-application headers
Any-application = <Octet 128>          ; encoding for "*"
```

8.4.2.58 Bearer-indication field

The following rule is used to encode the Bearer-indication field.

```
Bearer-indication-value = Integer-value
```

8.4.2.59 Push-flag field

The following rule is used to encode the Push-Flag field.

```
Push-flag-value = Short-integer
```

8.4.2.60 Profile field

The following rule is used to encode Profile values (user profile information as defined in [UAPROF]):

Profile-value = Uri-value

8.4.2.61 Profile-Diff field

The following rule is used to encode Profile-Diff values (profile difference information as defined in [UAPROF]):

Profile-diff-value = value-length CCPP-profile
CCPP-profile = *OCTET ; encoded in WBXML form – see [WBXML]

8.4.2.62 Profile-Warning field

The following rule is used to encode Profile-Warning values (responses from gateways or origin servers as defined in [UAPROF]):

Profile-warning-value = Warn-code | (Value-length Warn-code Warn-target *Warn-date)
;Warn-code as defined in 8.4.2.51
; assigned value Warning code (see [CCPPEx])
;0x10 100
;0x11 101
;0x12 102
;0x20 200
;0x21 201
;0x22 202
;0x23 203

Warn-target = Uri-value | host [":" port]
Warn-date = Date-value

8.4.2.63 Expect field

The following rule is used to encode Expect values:

Expect-value= 100-continue | Expect-expression
Expect-expression = Value-length Expression-var Expression-Value
Expression-var = Token-text
Expression-value = (Token-text | Quoted-string) *Expect-params
Expect-params = Token-text Expect-param-value
Expect-param-value = Text-value
100-continue = <Octet 128>

8.4.2.64 TE field

The following rule is used to encode TE values:

TE-value = Trailers | TE-General-Form
TE-General-Form = Value-length (Well-known-TE | Token-text) [Q-Parameter]
Q-Parameter = Q-token Q-value
Well-known-TE = (Chunked | Identity | Gzip | Compress | Deflate)
;Q-token as defined in section 8.4.2.7
Trailers = <Octet 129>
Chunked = <Octet 130>
Identity = <Octet 131>
Gzip = <Octet 132>
Compress = <Octet 133>
Deflate = <Octet 134>

8.4.2.65 Trailer field

The following rule is used to encode Trailer values:

```
Trailer-value = (Well-known-header-field | Token-text)
Well-known-header-field = Integer-value
; Encoded using values from Header Field Name Assignments table in Assigned Numbers
```

8.4.2.66 X-Wap-Tod field

The following rule is used to encode the X-Wap-Tod field:

```
X-Wap-Tod-value = Date-value
```

8.4.2.67 Content-ID field

The following rule is used to encode the Content-ID field. This header field is specified in RFC 2045.

```
Content-ID-value = Quoted-string
```

8.4.2.68 Set-Cookie field

The following rules are used to encode set cookie values:

```
Set-Cookie-value = Value-length Cookie-version Cookie-name Cookie-val *Parameter
Cookie-version = Version-value
Cookie-name = Text-string
Cookie-val = Text-string
```

8.4.2.69 Cookie field

The following rules are used to encode cookie values:

```
Cookie-value = Value-length Cookie-version *Cookie
Cookie = Cookie-length Cookie-name Cookie-val [Cookie-parameters]
Cookie-length = Uintvar-integer
Cookie-parameters = Path [Domain]
Path = Text-string
; if path is an empty string, it indicates that there is no path value present
Domain = Text-string
```

8.4.2.70 Encoding-Version field

The following rule should be used to encode the maximum supported binary encoding version value. The binary encoding version applies for any content types, parameter types and headers listed in its assigned number table or any headers defined for a dedicated extended header code page.

```
Encoding-version-value = Version-value | Value-length Code-page [Version-value]
;encoded using values from tables in the Assigned Numbers Appendix or from an
;extended header code page.

Code-page = Short-integer
;Identity of the extended header code page which the encoding applies for. If the
;Code-page is omitted, the version value refers to the header code pages reserved for
;headers specified by WAP Forum
```

The Encoding-version header is a hop-by-hop header that MUST be included in all request and reply. In connection oriented case it can be sent as static header so it doesn't have to be sent over the air all the time.

If a client does not include the Encoding-version header to its request then the server MUST assume that client only supports encoding defined in version 1.2 or lower. Similarly, if a server does not include the Encoding-version header to its response it indicates for the client that the server is only capable of handling binary encodings with version 1.2 or lower.

If extended header code pages are used, there SHOULD be a dedicated encoding version header for each extended header code page. In the absence of the encoding version header, the lowest possible version MUST be assumed for the extended header code page.

The usage of Encoding-version header is similar to HTTP version as defined in [RFC2145]. A WSP client MUST send the highest encoding version for which it is compliant, and whose version is no higher than the highest version supported by the server, if this is known. A WSP client MUST NOT send a version for which it is not compliant.

A WSP server MUST send a response version equal to the highest versions for which the server is compliant, and whose version is less than or equal to the one received in the request. A WSP server MUST NOT send a version for which it is not compliant. If extended header code pages are used, the WSP server MUST respond with an encoding version header for each extended header code page that is request by the client and supported by the server.

The client MAY save the supported encoding version received from the server response and use it to optimize header encoding for subsequent requests or session establishments. If the encoding version is provided by means of Client Provisioning [PROVCONT], the client SHOULD use this information to optimize subsequent requests or session establishments.

If a binary encoding, not supported by the negotiated encoding version or extended header code page, is received by the server. Status code 400 (Bad Request) MUST be returned to the client including the supported encoding versions supported by the server. If the binary encoding is defined in an unsupported header code page, the version number shall be omitted in the encoding version header in the response, to indicate which requested header code page is not supported by the server. The client can repeat the request (or re-establish the session with a new connect PDU) using textual encoding for unsupported headers.

Note: the encoding version is not linked to the WAP release version number (1.1, 1.2, etc) although it might be decided to increase the encoding version number at each new WAP release.

Note: Older implementations are encouraged to send it to help the differentiation between WSP 1.1 and WSP 1.2 implementations.

8.4.2.71 X-WAP-Security field

The following rule is used to encode X-WAP-Security value:

```
X-WAP-Security-value = Close-subordinate
Close-subordinate = <Octet 128>
```

X-WAP-Security usage is defined in [TLE2E].

8.4.2.72 X-Wap-Loc-Invocation field

The following rule is used to encode X-Wap-Loc-Invocation-value:

```
X-Wap-Loc-Invocation-value = value-length Loc-Invocation-document
Loc-Invocation-document = 1*OCTET; encoded in WBXML form - see [WAPWBXML]
```

8.4.2.73 X-Wap-Loc-Delivery field

The following rule is used to encode X-Wap-Loc-Delivery-value:

```
X-Wap-Loc-Delivery-value = value-length Loc-Delivery-document
Loc-Delivery-document = 1*OCTET; encoded in WBXML form - see [WAPWBXML]
```

8.4.3 Textual Header Syntax

The header definition rules in this sub-section follow the rules defined in [RFC2616]. When the header is sent in text format between the WSP peers, the Application-header rule in section 8.4.2.6 shall be used to encode the field name and field value.

8.4.3.1 Encoding-Version field

The following rule is used to encode Encoding-Version value in text format:

```
Encoding-version = "Encoding-version" ":" [Code-page SP] Version
;Encoding version supported by the peer as defined in section 8.4.2.70

Code-page = 1*2Hex
; Hexadecimal digits identifying the Code-page. If the Code-page is omitted,
; the version value refers to the header code pages reserved for headers specified
; by WAP Forum

Version = Short-integer | Text-version
; Version number as defined in appendix A. The version must be encoded as short-integer if ; ; ;possible.
; The encoding is defined in section 8.4.2.3 by the Version-value rule. If the version to be encoded
; does not fit constraints for a Short-integer, the Text-version rule shall be applied.

Text-version = 1*Digit"."1*Digit
; The first digit contains the major number and the second digit
; contains the minor version number
```

8.4.3.2 X-WAP-Security field

The following rule is used to encode X-WAP-Security value in text format:

```
X-WAP-Security = "X-WAP-Security" ":" "Close-Subordinate"
```

8.4.3.3 X-WAP-Tod field

The following rule is used to encode X-WAP-Tod value in text format:

```
X-WAP-Tod = "X-WAP-Tod" ":" [HTTP-date]
;HTTP-date is defined in [RFC2616]
```

The HTTP-date value MUST be omitted when the header is used in a request. This definition replaces the old definition defined in [UACACHE] that has been deprecated. The old definition should only be supported for backward compatibility purpose.

8.4.3.4 X-Wap-Loc-Invocation field

The following rule specifies the syntax of the X-Wap-Loc-Invocation header in augmented BNF defined by [RFC2616]:

```
X-Wap-Loc-Invocation = "X-Wap-Loc-Invocation" ":" *TEXT
;The value of the header is an XML document that contains location related information
```

8.4.3.5 X-Wap-Loc-Delivery field

The following rule specifies the syntax of the X-Wap-Loc-Delivery header in augmented BNF defined by [RFC2616]:

```
X-Wap-Loc-Delivery = "X-Wap-Loc-Delivery" ":" *TEXT
;The value of the header is an XML document that contains location related information
```

8.4.4 End-to-end and Hop-by-hop Headers

The classification whether an HTTP header is a Hop-by-hop or an End-to-end header is defined in [RFC2616]. The following header(s) defined by WAP Forum are Hop-by-hop headers:

- Encoding-version
- Profile
- Profile-Diff
- X-Wap-Tod

All Hop-by-hop headers MUST be listed in the Connection header except for those defined in [RFC2616]. Unsupported hop-by-hop headers SHOULD be ignored.

8.5 Multipart Data

HTTP/1.1 has adopted the MIME multipart format to transport composite data objects (eg, “multipart/mixed”). WSP defines a compact binary form of the MIME multipart entity. There is a straightforward translation of both the multipart entity and the content type. After translation, a “multipart/mixed” entity becomes an “application/vnd.wap.multipart.mixed” entity. Thus, all MIME “multipart/*” content types can be converted into “ application/vnd.wap.multipart.*” content types. No information is lost in the translation.

This translation may equally be applied to situations where the MIME multipart itself appears in a nested part of a parent MIME multipart. In this case the substitution of the content type is applied to the ContentType field in the Multipart Entry block. (see section 8.5.3) and the content format follows the rules for “application/vnd.wap.multipart” as specified in section 8.5.1.

8.5.1 Application/vnd.wap.multipart Format

Figure 31. Application/vnd.wap.multipart Format

The application/vnd.wap.multipart content type consists of a header followed by 0 or more entries.

8.5.2 Multipart Header

The multipart header format is as follows:

Table 34. Multipart Header Fields

Name	Type	Purpose
nEntries	Uintvar	The number of entries in the multipart entity

The *nEntries* field was used to specify the number of entries in the multipart entity in WAP implementations using encoding version 1.3 or earlier. This field is being deprecated as the receiver can determine the number of entries by stepping through each entry till the end of the PDU.

WSP implementations supporting encoding versions later than 1.3 MUST NOT use this field to determine the number of entries in any received multipart content.

WSP implementations supporting encoding versions later than 1.3 MUST set this field correctly if they cannot determine that the peer WSP supports an encoding version later than 1.3; if the peer implementation supports an encoding version later than 1.3, the sender SHOULD set the nEntries field to 0.

8.5.3 Multipart Entry

The multipart entry format is as follows:

Table 35. Multipart Entry Fields

Name	Type	Purpose
HeadersLen	Uintvar	Length of the *ContentType* and *Headers* fields combined
DataLen	Uintvar	Length of the *Data* field
ContentType	Multiple octets	The content type of the data
Headers	(*HeadersLen* – length of *ContentType)* octets	The headers
Data	*DataLen* octets	The data

The *HeadersLen* field specifies the length of the *ContentType* and *Headers* fields combined.

The *DataLen* field specifies the length of the *Data* field in the multipart entry.

The *ContentType* field contains the content type of the data. It conforms to the Content-Type value encoding specified in section 8.4.2.24, “Content type field”, above.

The *Headers* field contains the headers of the entry.

The *Data* field contains the data of the entry.

Appendix A. Assigned Numbers

This section contains tables of the WSP assigned numbers. The WAP Architecture Group is responsible for administering the values. Values in tables covered by the version header (Well-Known Parameters, Header Field and Content Type) MUST be given in a sequential fashion to simplify client version mapping implementation. New entity MUST be added at the end of any of those tables. When removing an entity in one of those tables, the assigned number MUST be deprecated and it MUST not be re-used for another entity. If the encoding rules of an entity need to be changed, a new entity MUST be created.

Table 34. PDU Type Assignments

Name	Assigned Number
Reserved	0x00
Connect	0x01
ConnectReply	0x02
Redirect	0x03
Reply	0x04
Disconnect	0x05
Push	0x06
ConfirmedPush	0x07
Suspend	0x08
Resume	0x09
Unassigned	0x10–0x3F
Get	0x40
Options (Get PDU)	0x41
Head (Get PDU)	0x42
Delete (Get PDU)	0x43
Trace (Get PDU)	0x44
Unassigned (Get PDU)	0x45-0x4F
Extended Method (Get PDU)	0x50-0x5F
Post	0x60
Put (Post PDU)	0x61
Unassigned (Post PDU)	0x62–0x6F
Extended Method (Post PDU)	0x70-0x7F
Data Fragment PDU	0x80
Reserved	0x81-0xFF

Table 35. Abort Reason Code Assignments

Name	Description	Assigned Number
PROTOERR	Protocol error, illegal PDU received	0xE0
DISCONNECT	Session has been disconnected	0xE1
SUSPEND	Session has been suspended	0xE2
RESUME	Session has been resumed	0xE3
CONGESTION	The peer is congested and can not process the SDU	0xE4
CONNECTERR	The session connect failed	0xE5
MRUEXCEEDED	The Maximum Receive Unit size was exceeded	0xE6
MOREXCEEDED	The Maximum Outstanding Requests was exceeded	0xE7
PEERREQ	Peer request	0xE8
NETERR	Network error	0xE9
USERREQ	User request	0xEA
USERRFS	User refused Push message. No specific cause, no retries	0xEB
USERPND	Push message cannot be delivered to intended destination	0xEC
USERDCR	Push message discarded due to resource shortage	0xED
USERDCU	Content type of Push message cannot be processed	0xEE

Table 36. Status Code Assignments

HTTP Status Code	Description	Assigned Number
none	reserved	0x00 to 0x0F
100	Continue	0x10
101	Switching Protocols	0x11
200	OK, Success	0x20
201	Created	0x21
202	Accepted	0x22
203	Non-Authoritative Information	0x23
204	No Content	0x24
205	Reset Content	0x25
206	Partial Content	0x26
300	Multiple Choices	0x30
301	Moved Permanently	0x31
302	Moved temporarily	0x32
303	See Other	0x33
304	Not modified	0x34
305	Use Proxy	0x35
306	(reserved)	0x36
307	Temporary Redirect	0x37
400	Bad Request - server could not understand request	0x40
401	Unauthorized	0x41
402	Payment required	0x42
403	Forbidden – operation is understood but refused	0x43
404	Not Found	0x44
405	Method not allowed	0x45
406	Not Acceptable	0x46
407	Proxy Authentication required	0x47
408	Request Timeout	0x48
409	Conflict	0x49
410	Gone	0x4A
411	Length Required	0x4B
412	Precondition failed	0x4C
413	Request entity too large	0x4D
414	Request-URI too large	0x4E
415	Unsupported media type	0x4F
416	Requested Range Not Satisfiable	0x50
417	Expectation Failed	0x51
500	Internal Server Error	0x60
501	Not Implemented	0x61
502	Bad Gateway	0x62
503	Service Unavailable	0x63
504	Gateway Timeout	0x64
505	HTTP version not supported	0x65

Table 37. Capability Assignments

Capability	Assigned Number
Client-SDU-Size	0x00
Server-SDU-Size	0x01
Protocol Options	0x02
Method-MOR	0x03
Push-MOR	0x04
Extended Methods	0x05
Header Code Pages	0x06
Aliases	0x07
Client-Message-Size	0x08
Server-Message-Size	0x09
Unassigned	0x0A to 0x7F

Table 38. Well-Known Parameter Assignments

Token	Encoding Version	Assigned Number	Expected BNF Rule for Value
Q	1.1	0x00	Q-value
Charset	1.1	0x01	Well-known-charset
Level	1.1	0x02	Version-value
Type	1.1	0x03	Integer-value
Name[1]	1.1	0x05	Text-string
Filename[1]	1.1	0x06	Text-string
Differences	1.1	0x07	Field-name
Padding	1.1	0x08	Short-integer
Type (when used as parameter of Content-Type: multipart/related)	1.2	0x09	Constrained-encoding
Start (with multipart/related)[1]	1.2	0x0A	Text-string
Start-info (with multipart/related)[1]	1.2	0x0B	Text-string
Comment[1]	1.3	0x0C	Text-string
Domain[1]	1.3	0x0D	Text-string
Max-Age	1.3	0x0E	Delta-seconds-value
Path[1]	1.3	0x0F	Text-string
Secure	1.3	0x10	No-value
SEC (when used as parameter of Content-Type: Application/vnd.wap.connectivity-wbxml)	1.4	0x11	Short-integer
MAC (when used as parameter of Content-Type: Application/vnd.wap.connectivity-wbxml)	1.4	0x12	Text-value
Creation-date	1.4	0x13	Date-value
Modification-date	1.4	0x14	Date-value
Read-date	1.4	0x15	Date-value
Size	1.4	0x16	Integer-value
Name	1.4	0x17	Text-value
Filename	1.4	0x18	Text-value
Start (with multipart/related)	1.4	0x19	Text-value

Token	Encoding Version	Assigned Number	Expected BNF Rule for Value
Start-info (with multipart/related)	1.4	0x1A	Text-value
Comment	1.4	0x1B	Text-value
Domain	1.4	0x1C	Text-value
Path	1.4	0x1D	Text-value

(1): These numbers have been deprecated and should not be used.

Table 39. Header Field Name Assignments

Name	Encoding Version	Assigned Number
Accept	1.1	0x00
Accept-Charset[1]	1.1	0x01
Accept-Encoding[1]	1.1	0x02
Accept-Language	1.1	0x03
Accept-Ranges	1.1	0x04
Age	1.1	0x05
Allow	1.1	0x06
Authorization	1.1	0x07
Cache-Control[1]	1.1	0x08
Connection	1.1	0x09
Content-Base[1]	1.1	0x0A
Content-Encoding	1.1	0x0B
Content-Language	1.1	0x0C
Content-Length	1.1	0x0D
Content-Location	1.1	0x0E
Content-MD5	1.1	0x0F
Content-Range[1]	1.1	0x10
Content-Type	1.1	0x11
Date	1.1	0x12
Etag	1.1	0x13
Expires	1.1	0x14
From	1.1	0x15
Host	1.1	0x16
If-Modified-Since	1.1	0x17
If-Match	1.1	0x18
If-None-Match	1.1	0x19
If-Range	1.1	0x1A
If-Unmodified-Since	1.1	0x1B
Location	1.1	0x1C
Last-Modified	1.1	0x1D
Max-Forwards	1.1	0x1E
Pragma	1.1	0x1F
Proxy-Authenticate	1.1	0x20
Proxy-Authorization	1.1	0x21
Public	1.1	0x22
Range	1.1	0x23
Referer	1.1	0x24
Retry-After	1.1	0x25
Server	1.1	0x26
Transfer-Encoding	1.1	0x27
Upgrade	1.1	0x28
User-Agent	1.1	0x29
Vary	1.1	0x2A
Via	1.1	0x2B
Warning	1.1	0x2C
WWW-Authenticate	1.1	0x2D
Content-Disposition[1]	1.1	0x2E
X-Wap-Application-Id	1.2	0x2F
X-Wap-Content-URI	1.2	0x30

Name	Encoding Version	Assigned Number
X-Wap-Initiator-URI	1.2	0x31
Accept-Application	1.2	0x32
Bearer-Indication	1.2	0x33
Push-Flag	1.2	0x34
Profile	1.2	0x35
Profile-Diff	1.2	0x36
Profile-Warning[1]	1.2	0x37
Expect[1]	1.3	0x38
TE	1.3	0x39
Trailer	1.3	0x3A
Accept-Charset	1.3	0x3B
Accept-Encoding	1.3	0x3C
Cache-Control[1]	1.3	0x3D
Content-Range	1.3	0x3E
X-Wap-Tod	1.3	0x3F
Content-ID	1.3	0x40
Set-Cookie	1.3	0x41
Cookie	1.3	0x42
Encoding-Version	1.3	0x43
Profile-Warning	1.4	0x44
Content-Disposition	1.4	0x45
X-WAP-Security	1.4	0x46
Cache-Control	1.4	0x47
Expect	1.5	0x48
X-Wap-Loc-Invocation	1.5	0x49
X-Wap-Loc-Delivery	1.5	0x4A

(1): These numbers have been deprecated and should only be supported for backward compatibility purpose

Table 40. Content Type Assignments

Content-Type	Encoding Version	Assigned Number

Note: this table is managed by the WAP Forum Naming Authority (WINA). Please refer to http://www.wapforum.org/wina for more details.

Table 41. ISO 639 Language Assignments

Language	Short	Assigned Number	Language	Short	Assigned Number
Afar	aa	0x01	Maori	mi	0x47
Abkhazian	ab	0x02	Macedonian	mk	0x48
Afrikaans	af	0x03	Malayalam	ml	0x49
Amharic	am	0x04	Mongolian	mn	0x4A
Arabic	ar	0x05	Moldavian	mo	0x4B
Assamese	as	0x06	Marathi	mr	0x4C
Aymara	ay	0x07	Malay	ms	0x4D
Azerbaijani	az	0x08	Maltese	mt	0x4E
Bashkir	ba	0x09	Burmese	my	0x4F
Byelorussian	be	0x0A	Nauru	na	0x81
Bulgarian	bg	0x0B	Nepali	ne	0x51
Bihari	bh	0x0C	Dutch	nl	0x52
Bislama	bi	0x0D	Norwegian	no	0x53
Bengali; Bangla	bn	0x0E	Occitan	oc	0x54
Tibetan	bo	0x0F	(Afan) Oromo	om	0x55
Breton	br	0x10	Oriya	or	0x56
Catalan	ca	0x11	Punjabi	pa	0x57
Corsican	co	0x12	Polish	po	0x58
Czech	cs	0x13	Pashto, Pushto	ps	0x59
Welsh	cy	0x14	Portuguese	pt	0x5A
Danish	da	0x15	Quechua	qu	0x5B
German	de	0x16	Rhaeto-Romance	rm	0x8C
Bhutani	dz	0x17	Kirundi	rn	0x5D
Greek	el	0x18	Romanian	ro	0x5E
English	en	0x19	Russian	ru	0x5F
Esperanto	eo	0x1A	Kinyarwanda	rw	0x60
Spanish	es	0x1B	Sanskrit	sa	0x61
Estonian	et	0x1C	Sindhi	sd	0x62
Basque	eu	0x1D	Sangho	sg	0x63
Persian	fa	0x1E	Serbo-Croatian	sh	0x64
Finnish	fi	0x1F	Sinhalese	si	0x65
Fiji	fj	0x20	Slovak	sk	0x66
Faeroese	fo	0x82	Slovenian	sl	0x67
French	fr	0x22	Samoan	sm	0x68
Frisian	fy	0x83	Shona	sn	0x69
Irish	ga	0x24	Somali	so	0x6A
Scots Gaelic	gd	0x25	Albanian	sq	0x6B
Galician	gl	0x26	Serbian	sr	0x6C
Guarani	gn	0x27	Siswati	ss	0x6D
Gujarati	gu	0x28	Sesotho	st	0x6E
Hausa	ha	0x29	Sundanese	su	0x6F
Hebrew (formerly iw)	he	0x2A	Swedish	sv	0x70
Hindi	hi	0x2B	Swahili	sw	0x71
Croatian	hr	0x2C	Tamil	ta	0x72
Hungarian	hu	0x2D	Telugu	te	0x73
Armenian	hy	0x2E	Tajik	tg	0x74
Interlingua	ia	0x84	Thai	th	0x75
Indonesian (formerly in)	id	0x30	Tigrinya	ti	0x76
Interlingue	ie	0x86	Turkmen	tk	0x77

Language	Short	Assigned Number	Language	Short	Assigned Number
Inupiak	ik	0x87	Tagalog	tl	0x78
Icelandic	is	0x33	Setswana	tn	0x79
Italian	it	0x34	Tonga	to	0x7A
Inuktitut	iu	0x89	Turkish	tr	0x7B
Japanese	ja	0x36	Tsonga	ts	0x7C
Javanese	jw	0x37	Tatar	tt	0x7D
Georgian	ka	0x38	Twi	tw	0x7E
Kazakh	kk	0x39	Uighur	ug	0x7F
Greenlandic	kl	0x8A	Ukrainian	uk	0x50
Cambodian	km	0x3B	Urdu	ur	0x21
Kannada	kn	0x3C	Uzbek	uz	0x23
Korean	ko	0x3D	Vietnamese	vi	0x2F
Kashmiri	ks	0x3E	Volapuk	vo	0x85
Kurdish	ku	0x3F	Wolof	wo	0x31
Kirghiz	ky	0x40	Xhosa	xh	0x32
Latin	la	0x8B	Yiddish (formerly ji)	yi	0x88
Lingala	ln	0x42	Yoruba	yo	0x35
Laothian	lo	0x43	Zhuang	za	0x3A
Lithuanian	lt	0x44	Chinese	zh	0x41
Latvian, Lettish	lv	0x45	Zulu	zu	0x5C
Malagasy	mg	0x46			

The character set encodings are done using the MIBEnum values assigned by the IANA in the registry available in <URL:ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets>. The following table provides just a quick reference:

Table 42. Character Set Assignment Examples

Character set	Assigned Number	IANA MIBEnum value
big5	0x07EA	2026
iso-10646-ucs-2	0x03E8	1000
iso-8859-1	0x04	4
iso-8859-2	0x05	5
iso-8859-3	0x06	6
iso-8859-4	0x07	7
iso-8859-5	0x08	8
iso-8859-6	0x09	9
iso-8859-7	0x0A	10
iso-8859-8	0x0B	11
iso-8859-9	0x0C	12
shift_JIS	0x11	17
us-ascii	0x03	3
utf-8	0x6A	106
gsm-default-alphabet	Not yet assigned	Not yet assigned

Table 43. Warning Code Assignments

The warning code encodings are chosen to be compatible with older specifications of the HTTP protocol. If a gateway receives two-digit warning codes from a server that follows an older specification, it MAY use them directly without referring to this table.

Code	Description	Assigned Number
110	Response is stale	10
111	Revalidation failed	11
112	Disconnected operation	12
113	Heuristic expiration	13
199	Miscellaneous warning	99[1]
214	Transformation applied	14
299	Miscellaneous persistent warning	99[1]

(1): Codes 199 and 299 have the same assigned number in order to maintain compatibility with previous specifications.

Appendix B. Header encoding examples

This section contains some illustrative examples for how header encoding shall be applied.

B.1 Header values

The header values are given in HTTP/1.1 syntax together with the corresponding WSP header encoded octet stream.

B.1.1 Encoding of primitive value

HTTP/1.1 header: Accept: application/vnd.wap.wmlc

Encoded header:

0x80	-- Well-known field name ”Accept” coded as a short integer
0x94	-- Well-known media ”application/vnd.wap.wmlc” coded as a short integer

B.1.2 Encoding of structured value

HTTP/1.1 header: Accept-Language: en;q=0.7

Encoded header:

0x83	-- Well-known field name ”Accept-Language”
0x02	-- Value length, general encoding must be applied.
0x99	-- Well-known language ”English”
0x47	-- Quality factor 0.7 (0.7 * 100 + 1 = 0x47)

B.1.3 Encoding of well-known list value

HTTP/1.1 header: Accept-Language: en, sv

Encoded header:

0x83	-- Well-known field name ”Accept-Language”
0x99	-- Well-known language ”English”
0x83	-- Well-known field name ”Accept-Language”
0xF0	-- Well-known language ”Swedish”

B.1.4 Encoding of date value

HTTP/1.1 header: Date: Thu, 23 Apr 1998 13:41:37 GMT

Encoded header:

0x92	-- Well-known field name ”Date”
0x04	-- Length of multi-octet integer
0x35	-- 4 date octets encoded as number of seconds from 1970-01-01,
0x3f	-- 00:00:00 GMT. The most significant octet shall be first.
0x45	--
0x11	--

B.1.5 Encoding of Content range

HTTP/1.1 header: Content-range: bytes 0-499/1025

Encoded header:

0x90	-- Well-known field name ”Content-range”
0x03	-- Value length
0x00	-- First octet position
0x88	-- Entity length
0x01	-- Entity length

B.1.6 Encoding of a new unassigned token

HTTP/1.1 header: Accept-ranges: new-range-unit

Encoded header:

0x84	-- Well-known field name ”Accept-ranges”
’n’’e’’w’’-’’r’’a’’n’’g’’e’’-’’u’’n’’i’’’t’ 0x00	-- Token coded as a null terminated text string

B.1.7 Encoding of a new unassigned header field name

HTTP/1.1 header: X-New-header: foo

Encoded header:

'X' '-' ‘N’’e’’w’’-’’h’’e’’a’’d’’e’’r’ 0x00	-- Field name coded as a null terminated text string
’f’’o’’o’ 0x00	-- Field value coded as null terminated text string

B.1.8 Encoding of a new unassigned list-valued header

HTTP/1.1 header: X-New-header: foo, bar

Encoded header:

'X' '-' ‘N’’e’’w’’-’’h’’e’’a’’d’’e’’r’ 0x00	-- Field name coded as a null terminated text string
’f’’o’’o’’,’’ ’’b’’a’’r’ 0x00	-- Field value coded as null terminated text string

B.2 Shift header code pages

This section illustrates how header code pages can be shifted.

B.2.1 Shift sequence

Shift to header code page 64

Encoded shift sequence:

```
0x7F        -- Shift delimiter
0x40        -- Page identity
```

B.2.2 Short cut

Shift to header code page 16

Encoded shift sequence:

```
0x10        -- Short cut shift delimiter
```

Appendix C. Implementation Notes

The following implementation notes are provided to identify areas where implementation choices may impact the performance and effectiveness of the WSP protocols. These notes provide guidance to implementers of the protocols.

C.1 Confirmed Push and Delayed Acknowledgements

One of the features of the Wireless Transaction Protocol is delayed acknowledgement of transactions, which may significantly reduce the number of messages sent over the bearer network. However, this feature may also result in poor throughput for push traffic, especially if the server waits for a confirmed push to be acknowledged before starting the next confirmed push transaction. Use of delayed acknowledgements will make the push cycle to take at least one round-trip time plus the duration of the delayed acknowledgement timer. This effect will be even more pronounced when the bearer network has a long round-trip delay, since then WTP will typically use a larger delayed acknowledgement timer value.

The session layer protocol does not address this issue, because the WTP service interface does not include a means to effect the delayed acknowledgement timer. Rather, the control of that timer is a matter local to the implementation. If the performance implications are considered significant, an implementation should provide the service user with means to specify the largest acceptable acknowledgement delay for each push transaction. Forcing the delayed acknowledgement timer always to have a value that is small enough to provide good push throughput is not a good solution. This will prevent the remaining WTP message traffic associated with method requests from being optimised, and the number of messages sent over the air-interface will be doubled.

C.2 Handling of Race Conditions

Connection-mode WSP is layered on top of the service provided by the Wireless Transaction Protocol, which does not guarantee that transaction invocations and results arrive to the peer in the same order as in which the service user has submitted them. This results in certain race conditions, if method or push transactions are initiated while the session creation procedure has not yet been fully completed. In order to reduce protocol complexity WSP does not attempt to handle all of these gracefully, but in many cases simply chooses to abort the transaction caught in the race condition. In such a case the service user should simply retry the transaction request.

This policy was chosen, since these race conditions were not considered frequent enough to make the cost of the additional protocol complexity worthwhile. However, if the problem is considered significant, it can still be alleviated using certain implementation strategies. First of all, if session management, method and push transactions are initiated so close together that the race conditions are possible, then WTP concatenation procedures should be capable of combining the resulting PDUs into the same transport datagram. WTP should also handle the concatenation and separation in such a manner that the order of operations is preserved, if the resulting PDUs are carried by the same datagram. This will ensure that the state machine of WSP will not need to react to primitives related to method and push transactions before it has had a chance to complete creation of the session.

If an implementation wants to prevent completely these kinds of race conditions, it can postpone the initiation of method and push transactions until the session creation process is fully complete – this is quite legal as far as the protocol peer is concerned. However, the resulting user experience may be considered unacceptably poor, if the used bearer has a very long round-trip time.

C.3 Optimising Session Disconnection and Suspension

The protocol requires all pending method and push transactions to be aborted, when a peer starts disconnecting or suspending a session. This may result in a burst of very short messages containing transaction abort PDUs being sent in addition to the actual Disconnect or Suspend PDU. However, all these PDUs are so short, that typically it will be possible to concatenate them into a single transport datagram. An implementation should ensure that it is able to concatenate the PDUs at the WTP level at least in this special case, so that the impact on the network will be minimised.

C.4 Decoding the Header Encodings

WSP defines compact binary encodings for HTTP/1.1 headers. One method used to achieve this is the use of context information to define, how a particular encoding is supposed to be interpreted, instead of encoding it explicitly. For instance, the header field name implies the format of the header field value. In a structured value, the position of each item implies its type, even if the binary encodings used to represent the values of different types may in fact be identical. The most obvious method, which an implementation can use to support this, is using a top-down strategy when parsing the header encoding.

C.5 Adding Well-known Parameters and Tokens

The header encoding defined by WSP imposes a strict syntax on the header field values. Within it only such values that have been assigned well-known binary identities in advance can be encoded very compactly. If an application turns out to use extensively token values and especially parameters, which have not been foreseen, the overhead of the required textual encoding may eventually be considered prohibitive. If updating the WSP specification so that a new protocol version is produced is not a viable approach, then more efficient encodings can still be implemented within the WSP framework. The application may introduce an extension header code page, which redefines the syntax for the appropriate standard HTTP/1.1 header so that the needed new well-known values are recognised. The application peers can then use WSP capability negotiation to agree on using this new code page. Once this has been done, the application can modify its header processing so, that the header defined on the new code page will be used instead of the standard header with the same name. The cost of shifting to the new code page should be only one extra octet, which should be more than offset by the more compact value encoding.

C.6 Use of Custom Header Fields

Client or server implementations may make use of custom header fields, either as part of the header data supplied in a WSP request or response message, or as data supplied in "Acknowledgement Headers" (section 8.2.3.5). In these cases, it is important to choose field names that will be unambiguous and will not "collide" with other implementations. To avoid such problems, custom header field names should be created according to the guidelines specified by the WAP Wireless Interim Naming Authority (WINA), and registered with the authority as a means of public record. These measures will avoid name collision with standard field names, as well as custom header fields that may be defined in other implementations.

When using custom header fields, implementers should provide an extension header code page to allow compact encoding of these new field names and values (see section C.5). Use of extension code pages is strongly recommended for Acknowledgement Headers, since no standard header field names have been defined for them. Because Acknowledgement headers are only transferred between WSP peers, the negotiation of the extension code page can be performed at the same stage as the negotiation of the use of Acknowledgement Headers. Extension header code pages should also be named in accordance with WINA guidelines, and registered with the authority for public reference.

Client or server implementations should ignore custom header fields that they do not recognize. Gateway implementations that bridge between WSP and HTTP should pass any unrecognized HTTP header fields which are not specifically defined to be “hop-by-hop” fields.

Similarly a value in an accept header that is not recognized by the receiver should simply be ignored. If this leaves the server without any accepted values the server can chose to either send a report as “Not Acceptable” (406) or return some generic content it expects the client to be able to handle.

In case the custom header clearly breaks the protocol, such as if using a non-negotiated code page, then the server should return a “Bad Request” (400) error to the client.

Appendix D. Static Conformance Requirement

D.1 Client Mode

Item	Function	Reference	Status	Requirement
WSP-C-001	Device Mode	Section 6,7&8	M	WSP-CL-C-001 OR WSP-CO-C-001
WSP-CL-C-001	Connectionless	Section 6,7&8	O	WDP:MCF AND WSP-CL-C-003 AND WSP-CL-C-004 AND WSP-CL-C-005 AND WSP-CL-C-006 AND WSP-CL-C-007 AND WSP-CL-C-020
WSP-CO-C-001	Connection-Oriented	Section 6,7&8	O	WTP: MCF AND WSP-CO-C-002 AND WSP-CO-C-003 AND WSP-CO-C-004 AND WSP-CO-C-005 AND WSP-CO-C-006 AND WSP-CO-C-007 AND WSP-CO-C-014 AND WSP-CO-C-017 AND WSP-CO-C-018 AND WSP-CO-C-020 AND WSP-CO-C-021 AND WSP-CO-C-037 AND WSP-CO-C-039

D.2 Server Mode

Item	Function	Reference	Status	Requirement
WSP-S-001	Server Mode	Section 6,7&8	M	WSP-CL-S-001 AND WSP-CO-S-001
WSP-CL-S-001	Connectionless	Section 6,7&8	O	WDP:MSF AND WSP-CL-S-003 AND WSP-CL-S-004 AND WSP-CL-S-005 AND WSP-CL-S-006 AND WSP-CL-S-007 AND WSP-CL-S-020
WSP-CO-S-001	Connection-Oriented	Section 6,7&8	O	WTP: MSF AND WSP-CO-S-002 AND WSP-CO-S-003 AND WSP-CO-S-005 AND WSP-CO-S-006 AND WSP-CO-S-007 AND WSP-CO-S-014 AND WSP-CO-S-017 AND WSP-CO-S-018 AND WSP-CO-S-020 AND WSP-CO-S-021 AND WSP-CO-S-037

D.3 Connection-Oriented Client

Item	Function	Reference	Status	Requirement
WSP-CO-C-002	Connect PDU	6.3.3.1 6.3.4 7.1.2.1 7.1.5 7.1.6.1 8.2.2.1	O	WTP:MCF
WSP-CO-C-003	ConnectReply PDU	7.1.2.1 7.1.5 7.1.6.1 8.2.2.2	O	WTP:MCF
WSP-CO-C-004	Redirect PDU	7.1.2.1 7.1.5 7.1.6.1 8.2.2.3	O	WTP:MCF
WSP-CO-C-005	Capability Negotiation - Connect PDU	6.3.2 6.3.3.1 7.1.5 7.1.6.1 8.2.2.1 8.3	O	WTP:MCF
WSP-CO-C-006	Capability Negotiation - ConnectReply PDU	7.1.5 7.1.6.1 8.2.2.2 8.3	O	WTP:MCF
WSP-CO-C-007	Disconnect PDU	6.3.3.2 7.1.2.1 7.1.5 7.1.6.1 8.2.2.4	O	WTP:MCF
WSP-CO-C-008	Suspend PDU	6.3.3.3 7.1.2.2 7.1.5 7.1.6.1 8.2.5.1	O	WTP:MCF
WSP-CO-C-009	Resume PDU	6.3.3.4 7.1.2.2 7.1.5 7.1.6.1 8.2.5.2	O	WTP:MCF
WSP-CO-C-010	Push PDU	6.3.3.9 7.1.2.4 8.2.4.1	O	WTP:MCF
WSP-CO-C-011	ConfirmedPush PDU	6.3.3.10 6.3.3.11 6.3.4 7.1.2.5 7.1.5 7.1.6.3 8.2.4.1	O	WSP-CO-C-038 AND WTP:MCF
WSP-CO-C-012	Ack. Headers	6.3.3.7 6.3.3.10	O	WTP:MCF AND WTP-C-013

		7.1.6.2 7.1.6.3 8.2.4.2		
WSP-CO-C-013	Extended Methods	6.3.2.2 8.3.2.4	O	WTP:MCF
WSP-CO-C-014	Default Header Code Page Encoding	8.4 Table 39	O	
WSP-CO-C-015	Extended Header Code Page Encoding	8.3.2.5	O	WTP:MCF
WSP-CO-C-016	Aliases	6.3.2.2 8.3.2.6	O	WTP:MCF
WSP-CO-C-017	Method GET - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.1	O	WTP:MCF
WSP-CO-C-018	Method GET - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-019	Method GET - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4	O	WTP:MCF
WSP-CO-C-020	Method POST - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.2	O	WTP:MCF
WSP-CO-C-021	Method POST - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-022	Method POST - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4	O	WTP:MCF

		7.1.2.3 7.1.5 7.1.6.2 8.2.3.4		
WSP-CO-C-023	Method DELETE - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.1	O	WTP:MCF
WSP-CO-C-024	Method DELETE - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-025	Method HEAD - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.1	O	WTP:MCF
WSP-CO-C-026	Method HEAD - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-027	Method OPTIONS - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.1	O	WTP:MCF
WSP-CO-C-028	Method OPTIONS - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-029	Method TRACE - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8	O	WTP:MCF

		6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.1		
WSP-CO-C-030	Method TRACE - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-031	Method PUT - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.2	O	WTP:MCF
WSP-CO-C-032	Method PUT - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-033	Method PUT - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4	O	WTP:MCF
WSP-CO-C-034	Multipart Data - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.2	O	WTP:MCF
WSP-CO-C-035	Multipart Data - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.3	O	WTP:MCF
WSP-CO-C-036	Multipart Data - Fragment PDU	6.3.3.6 6.3.3.7	O	WTP:MCF

		6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4		
WSP-CO-C-037	Method Abort	6.3.3.2 6.3.3.3 6.3.3.8	O	WTP:MCF
WSP-CO-C-038	Push Abort	6.3.3.10 6.3.3.11	O	WTP:MCF
WSP-CO-C-039	Encoding Version Framework	8.4.1 8.4.2.70	O	WTP:MCF

D.4 Connectionless Client

Item	Function	Reference	Status	Requirement
WSP-CL-C-002	Push PDU	6.4.2.3 6.3.4 7.2 8.2.4.1	O	WDP:MCF
WSP-CL-C-003	Header Encoding Default page	8.4 Table 39	O	
WSP-CL-C-004	Method GET - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MCF
WSP-CL-C-005	Method GET - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-006	Method POST - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MCF
WSP-CL-C-007	Method POST - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-008	Method DELETE - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MCF
WSP-CL-C-009	Method DELETE - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-010	Method HEAD - Get PDU	6.4.2.1 6.4.2.2	O	WDP:MCF

		6.4.3 7.2 8.2.3.1		

WSP-CL-C-011	Method HEAD - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-012	Method OPTIONS - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MCF
WSP-CL-C-013	Method OPTIONS - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-014	Method TRACE - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MCF
WSP-CL-C-015	Method TRACE - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-016	Method PUT - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MCF
WSP-CL-C-017	Method PUT - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-018	Multipart Data - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MCF
WSP-CL-C-019	Multipart Data - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MCF
WSP-CL-C-020	Encoding Version Framework	8.4.1 8.4.2.70	O	WDP:MCF

D.5 Connection-Oriented Server

Item	Function	Reference	Status	Requirement
WSP-CO-S-002	Connect PDU	6.3.3.1 6.3.4 7.1.2.1 7.1.5 7.1.6.4 8.2.2.1	O	WTP:MSF
WSP-CO-S-003	ConnectReply PDU	7.1.2.1 7.1.5 7.1.6.4 8.2.2.2	O	WTP:MSF
WSP-CO-S-004	Redirect PDU	7.1.2.1 7.1.5 7.1.6.4 8.2.2.3	O	WTP:MSF
WSP-CO-S-005	Capability Negotiation - Connect PDU	6.3.2 6.3.3.1 6.3.3.4 7.1.5 7.1.6.4 8.2.2.1 8.3	O	WTP:MSF
WSP-CO-S-006	Capability Negotiation - ConnectReply PDU	7.1.5 7.1.6.4 8.2.2.2 8.3	O	WTP:MSF
WSP-CO-S-007	Disconnect PDU	6.3.3.2 7.1.2.1 7.1.5 7.1.6.4 8.2.2.4	O	WTP:MSF
WSP-CO-S-008	Suspend PDU	6.3.3.3 7.1.2.2 7.1.5 7.1.6.4 8.2.5.1	O	WTP:MSF
WSP-CO-S-009	Resume PDU	6.3.3.4 7.1.2.2 7.1.5 7.1.6.4 8.2.5.2	O	WTP:MSF
WSP-CO-S-010	Push PDU	6.3.3.9 7.1.2.4 8.2.4.1	O	WTP:MSF
WSP-CO-S-011	ConfirmedPush PDU	6.3.3.10 6.3.3.11 6.3.4 7.1.2.5 7.1.5 7.1.6.6 8.2.4.1	O	WTP:MSF
WSP-CO-S-012	Ack. Headers	6.3.3.7	O	WTP:MSF AND

		6.3.3.10 7.1.6.5 7.1.6.6 8.2.4.2		WTP-S-013
WSP-CO-S-013	Extended Methods	6.3.2.2 8.3.2.4	O	WTP:MSF
WSP-CO-S-014	Default Header Code Page Encoding	6.3.2.2 8.4 Table 39	O	
WSP-CO-S-015	Extended Header Code Page Encoding	6.3.2.2 8.3.2.5	O	WTP:MSF
WSP-CO-S-016	Aliases	6.3.2.2 8.3.2.6	O	WTP:MSF
WSP-CO-S-017	Method GET - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.1	O	WTP:MSF
WSP-CO-S-018	Method GET - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-019	Method GET - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4	O	WTP:MSF
WSP-CO-S-020	Method POST - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.2	O	WTP:MSF
WSP-CO-S-021	Method POST - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-022	Method POST - Data Fragment PDU	6.3.3.6 6.3.3.7	O	WTP:MSF

		6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4		
WSP-CO-S-023	Method DELETE - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.1	O	WTP:MSF
WSP-CO-S-024	Method DELETE - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-025	Method HEAD - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.1	O	WTP:MSF
WSP-CO-S-026	Method HEAD - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-027	Method OPTIONS - Get PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.1	O	WTP:MSF
WSP-CO-S-028	Method OPTIONS - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-029	Method TRACE -	6.3.3.6	O	WTP:MSF

	Get PDU	6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.1		
WSP-CO-S-030	Method TRACE - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-031	Method PUT - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.2	O	WTP:MSF
WSP-CO-S-032	Method PUT - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF
WSP-CO-S-033	Method PUT - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4	O	WTP:MSF
WSP-CO-S-034	Multipart Data - Post PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.2	O	WTP:MSF
WSP-CO-S-035	Multipart Data - Reply PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.5 8.2.3.3	O	WTP:MSF

WSP-CO-S-036	Multipart Data - Data Fragment PDU	6.3.3.6 6.3.3.7 6.3.3.8 6.3.4 7.1.2.3 7.1.5 7.1.6.2 8.2.3.4	O	WTP:MSF
WSP-CO-S-037	Encoding Version Framework	8.4.1 8.4.2.70	O	WTP:MSF

D.6 Connectionless Server

Item	Function	Reference	Status	Requirement
WSP-CL-S-002	Push PDU	6.4.2.3 6.3.4 7.2 8.2.4.1	O	WDP:MSF
WSP-CL-S-003	Header Encoding Default page	8.4 Table 39	O	
WSP-CL-S-004	Method GET - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MSF
WSP-CL-S-005	Method GET - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-006	Method POST - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MSF
WSP-CL-S-007	Method POST - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-008	Method DELETE - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MSF
WSP-CL-S-009	Method DELETE - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-010	Method HEAD - Get PDU	6.4.2.1 6.4.2.2 6.4.3	O	WDP:MSF

		7.2 8.2.3.1		
WSP-CL-S-011	Method HEAD - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-012	Method OPTIONS - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MSF
WSP-CL-S-013	Method OPTIONS - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-014	Method TRACE - Get PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.1	O	WDP:MSF
WSP-CL-S-015	Method TRACE - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-016	Method PUT - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MSF
WSP-CL-S-017	Method PUT - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-S-018	Multipart Data - Post PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.2	O	WDP:MSF
WSP-CL-S-019	Multipart Data - Reply PDU	6.4.2.1 6.4.2.2 6.4.3 7.2 8.2.3.3	O	WDP:MSF
WSP-CL-020	Encoding Version Framework	8.4.1 8.4.2.70	O	WDP:MSF

Appendix E. Change History (Informative)

Type of Change	Date	Section	Description
OMA-WAP-WSP-1_0-20020920-c	20-Sept-2002		Document Approved as Candidate
OMA-WAP-WSP-1_0-20020920-C	10-Jan-2005	2.1	Reference documents for WCMP, WTP and WDP corrected