Decentralized Software Services Protocol – DSSP/1.0

Authors

Henrik Frystyk Nielsen, Microsoft
George Chrysanthakopoulos, Microsoft

Abstract

DSSP is a simple SOAP-based application protocol that defines a lightweight service model with a common notion of service identity, state, and relationships between services. DSSP defines a set of state-oriented message operations that provide support for structured data retrieval, manipulation, and event notification. The intent of DSSP is to provide a flexible foundation for defining applications as compositions of services interacting in a decentralized environment. The functionality provided by DSSP is an extension of the application model provided by HTTP and is expected to be used as an addition to existing HTTP infrastructure.

Terms of Use

Status of this Document

This document is part of Microsoft Robotics Studio. Please visit Microsoft Robotics Studio Developer Center [8] for more details and the latest developments.

Table of Contents

1 Introduction

1.1 DSSP Overview

DSSP defines an application as a composition of services that can be harnessed to achieve a desired task through orchestration. Services are lightweight entities that can be created, manipulated, monitored, and destroyed repeatedly over the lifetime of an application by using operations defined by DSSP.

A service consists of:

- Identity – The globally unique reference of the service.
- Behavior – The definition of the service functionality.
- Service State – The current state of the service.
- Service Context – The relationships the service has to other services.

DSSP provides a uniform model for creating, deleting, manipulating, subscribing, and orchestrating services independent of the semantics of those services. DSSP achieves this by separating state from behavior, allowing services to expose their state and hide their behavior.

DSSP enables but does not require a shared data model across all services. As a result, some DSSP messages are concrete and others are polymorphic and must be tailored to the particular content model of a service.

1.1.1 DSSP and HTTP

The DSSP operations are designed to be a superset of the methods provided by HTTP/1.1. In particular, DSSP provides support for structured data manipulation and event notification as an integral part of the service model. Because HTTP and DSSP have inherently different protocol characteristics, DSSP is complementary to HTTP and not intended as a replacement. Rather, the close relationship between DSSP and HTTP allows DSSP services to be accessed either as regular HTTP resources or as DSSP services. This provides additional support for structured data manipulation and event notification.

1.2 Terminology

The terminology used in this specification is based on the terminology defined in SOAP 1.2 with the following additions:

Service
: A computational unit that has identity, state, behavior, and context (see section 1.3).

Application
: A composition of services that can be harnessed to achieve a desired task through orchestration. (See section 3.)

Service identifier
: The globally unique identifier of a service. (See section 2.1.)

Service state
: The data representing a service at a specific point in time. (See section 2.2.)

Service behavior/contract
: The combination of the content model describing the state and the messages exchanges that a service defines for communicating with other services. (See section 2.3.)

Contract identifier
: The globally unique identifier for the behavior of a service. (See section 2.3.)

Partner
: A labeled reference representing a behavioral relationship between services. (See section 2.3.)

Service context
: The service context contains information about a service instance including which contract it is using and which partners it communicates with (see section 2.4).

Message
: A one-way message, request, or response participating in a DSSP operation. (See section 4.1.)

Request
: A message participating in a request-response message exchange pattern. (See section 4.1.2.)

Response
: A message participating in a request-response message exchange pattern. (See section 4.1.2.)

1.3 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

As a notational convention this specification uses the namespace prefixes listed in Table 1. A namespace prefix is not semantically significant (see [4]).

Prefix	Namespace
S	http://www.w3.org/2003/05/soap-envelope
DSSP	http://schemas.microsoft.com/xw/2004/10/dssp.html

Table 1 Prefixes and namespaces used in this specification.

DSSP follows the SOAP 1.2 fault model of a "code" followed by a "subcode" where the "code" is dictated by the SOAP 1.2 specification. The notational convention for indicating a SOAP fault code and subcode is "Code / Subcode" where `Code` and `Subcode` are XML Qualified Names.

2 Service Model

2.1 DSSP Service Identity

A DSSP service is identified by a URI. It is *strongly* recommended that a service identifier be globally unique. The service identifier only provides identity; it does not convey any information about the service state, behavior, or context.

2.2 DSSP Service State

The service state is a representation of a service at any given point in time. For example:

- The state of a service representing a motor may consist of rotations per minute, temperature, oil pressure, and fuel consumption.
- A service representing a work queue may contain a list of all queued work items and their current status. The work items themselves may be services allowing the work queue to simply refer to them using their identity.
- A service representing a keyboard may contain information about which keys have been pressed.

Any information that is to be retrieved modified, or monitored using DSSP must be expressed as part of the service state. (See section 5.11 for information on semantics that does not alter the service state.)

2.3 DSSP Service Behavior

The behavior of a service (the *contract*) is the combination of the content model describing the state and the message exchanges that a service defines for communicating with other services. The behavior of a service is identified by a globally unique URI known as the *contract identifier*.

The behavior of a service determines how it can compose with other services. Figure 1 illustrates an example of such composition. In this case, Service A needs service B to have the correct behavior in order to fulfill its task. Similarly, Service B needs Service C and E to have the correct behavior for it to fulfill its task and so forth.

Such composition is called *partnering*. A partner is a labeled reference representing a particular behavioral relationship that a service has to another service.

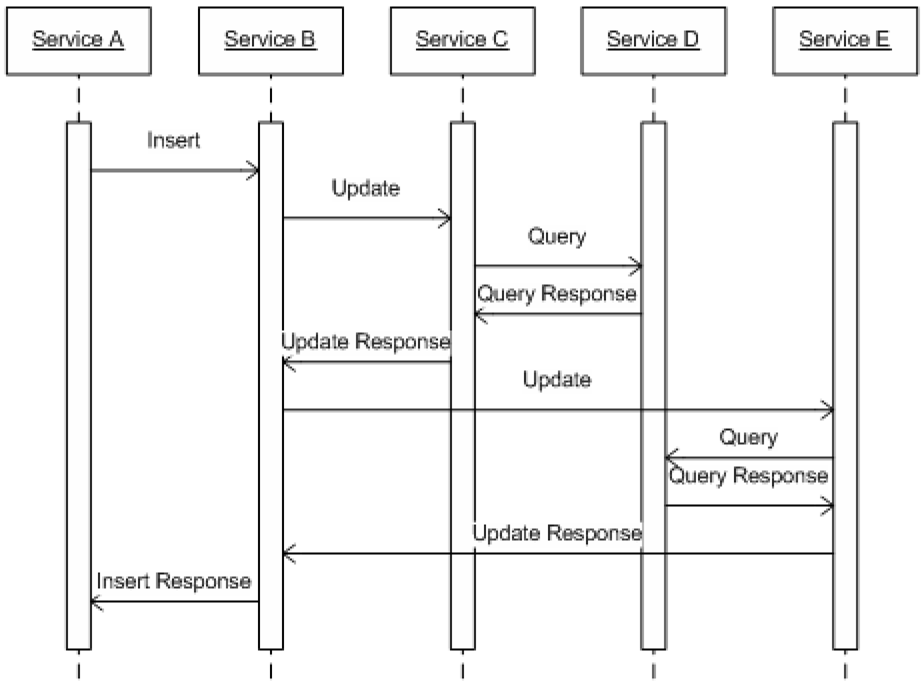

Figure 1 Example of how services can compose. Service A has partner B. Service B has partner C and E. Service C has partner D.

2.4 DSSP Service Context

While the contract of service A states the behavioral requirements on service B, a running Service A must have an actual Service B to communicate with. Partners are initialized at service creation (see section 3.1) and can be inspected using the `LOOKUP` operation (see section 3.2).

Aliasing is where two or more services share the exact same execution context. Like partners, aliasing is part of the service context and can be inspected using the `LOOKUP` operation (see section 3.2). While an implementation may choose to optimize aliasing by functionally exposing a single execution context with multiple identifiers, logically there is always a 1:1 relationship between a service and an identifier.

3 DSSP Application Model

An application is a composition of services that, through orchestration, can be harnessed to achieve a desired task. This section describes mechanisms provided for creating and destroying services, for retrieving and manipulating their state, and for subscribing for event notifications as a result of state changes.

3.1 DSSP Service Creation

Services can be created using the `CREATE` operation (see section 5.2). A service that supports the `CREATE` operation is called a *constructor* service. Constructor services are generic in the sense that they can support the creation of arbitrary services. A `CREATE` request does not carry any state specific to the newly created service. It *may* contain an optional service context indicating which specific partners the new service should use. Figure 2 illustrates the creation of a new service.

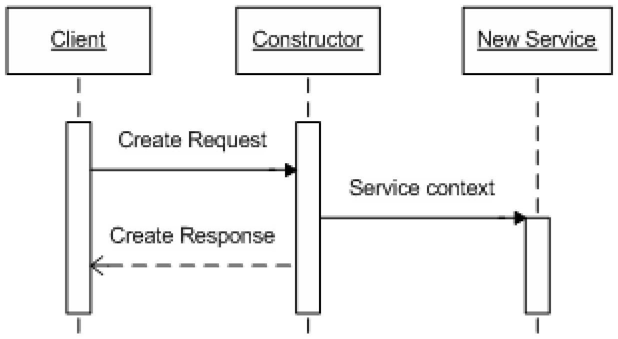

Figure 2 Sample sequence diagram illustrating the creation of a service using a constructor service. The service context can optionally be provided as part of the CREATE request.

A consequence of the DSSP constructor model is that initial state must either be passed by reference as part of an optional service context in the `CREATE` request, or by explicitly changing the state of the newly created service. The benefit is that setting the initial state becomes an explicit part of the behavior of a service. For example, a service may initialize itself simply by performing a `GET` request on another service and use that as its initial state (see Figure 3). By passing the appropriate service context as part of the `CREATE` request, the requesting party can control which service is used as the initial state partner.

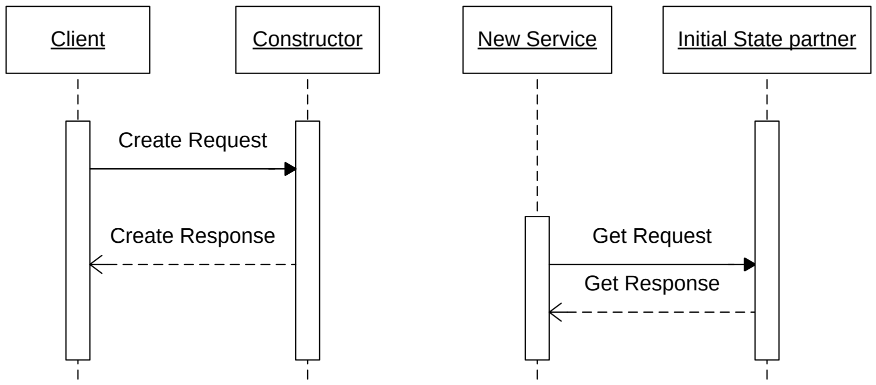

Figure 3 The newly created service gets its initial state from a partner service by doing a GET on that partner. The initial state partner may have been provided as part of the optional service context in the CREATE request or the newly created service may have a default initial state partner.

Alternatively, a service may be initialized using an explicit DSSP operation like `REPLACE` as illustrated in Figure 4.

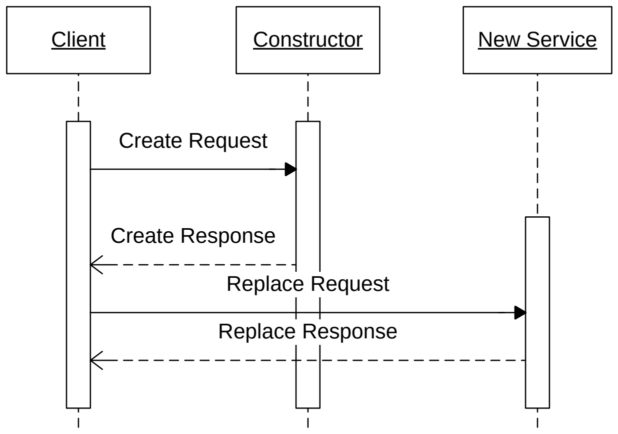

Figure 4 Example of service initialization using an explicit REPLACE operation after service has been constructed.

3.2 DSSP Service Termination

Existing services can be terminated using the `DROP` operation (see section 5.4).

A successful shut down of a service is indicated by the generation of a `DROP` response which *must* be the final message sent by that service. After a DSSP service has been terminated it can no longer send or receive messages of any type.

As part of processing a `DROP` request, a service may initiate communications with other services as part of any cleanup processing required by the service such as removing itself from a directory service or shutting down dependent services.

3.3 Determining the Service Context

The *service context* of a service can be retrieved using the `LOOKUP` operation (see section 5.7). The `LOOKUP` operation is the only DSSP operation that provides a concrete definition for both the request and response and is required to be supported by all DSSP services.

A `LOOKUP` response contains the following information:

- The service identifier.
- The service contract identifier
- The service context.

The service context retrieved in this manner cannot be modified directly; it can only be inspected.

3.4 DSSP Service State Retrieval

DSSP defines two operations for service state retrieval: `GET` (see section 5.5) and `QUERY` (see section 5.8). While the two operations are similar, an important difference is that the `QUERY` request contains a structured query against the service state whereas a `GET` request is an application-independent request for the complete service state. This difference has two consequences worth noting:

- A `QUERY` request requires specific schema and query language knowledge about the service being queried whereas a `GET` request doesn't.

- A `QUERY` request contains parameters (the structured query) that are not part of the service identifier whereas a `GET` request provides all request parameters as part of the service identifier.

In some situations, it may be beneficial to map a structured `QUERY` operation to an unstructured `GET` operation. Creating a service which through service composition (partnering) maps a GET operation to a specific structured QUERY operation on another service can be useful in situations where the same query occurs in many different contexts and a single service identifier is easier to handle in terms of logging, determining equality, etc. Note that such mapping can be done by any service; it does not have to be related to the service against which the `QUERY` is ultimately targeted.

3.5 DSSP Service State Modification

DSSP enables the state of a service to be modified using the `INSERT`, `UPDATE`, `UPSERT`, `DELETE`, and `REPLACE` operations. These operations apply only to the state of a service. They do not create or terminate services.

Note that the concrete representation of an `INSERT`, `DELETE`, `UPDATE`, and `UPSERT` request (but not `REPLACE`) is service specific. If desired, these operations can be designed to include a query identifying the specific part of the service state that is to be modified, for example, using an XPath expression. DSSP does not mandate or require a particular query language.

3.5.1 Inserting Service State

When the state of a service is modified using the `INSERT` operation (see section 5.6), the state included in the `INSERT` request is added to the service state. Assuming a service is being observed in isolation, the result of an `INSERT` operation can be detected by looking at the difference between a `GET` operation issued before the `INSERT` operation and a `GET` operation issued after the `INSERT` operation.

3.5.2 Deleting Service State

When the state of a service is modified using the `DELETE` operation (see section 5.3), the part of the service state identified in the `DELETE` request is deleted. Assuming a service is being observed in isolation, the result of a `DELETE` operation can be detected by looking at the difference between a `GET` operation issued before the `DELETE` operation and a `GET` operation issued after the `DELETE` operation.

3.5.3 Changing Service State

DSSP defines three operations for changing service state:
- `UPDATE` (see section 5.11)
- `UPSERT` (see section 5.13)
- `REPLACE` (see section 5.9)

When the state of a service is modified using the `UPDATE` operation, the part of the current service state identified in the `UPDATE` request is replaced with the new state included in the `UPDATE` request. The `UPDATE` operation is similar to `DELETE` followed by `INSERT` performed in a single operation. That is, `UPDATE` succeeds only if the existing service state is successfully deleted and the new service state successfully inserted.

The `UPSERT` operation provides a commonly used variant: if the existing service state is already present then `UPSERT` operates as `UPDATE`, otherwise `UPSERT` operates as `INSERT`.

The `REPLACE` operation can be used to replace the entire service state regardless of the existing service state.

Assuming a service is being observed in isolation, you can see the impact of all three operations by looking at the difference between a `GET` operation issued before the `UPDATE`, `UPSERT`, or `REPLACE` operation and a `GET` operation issued after the operation.

3.6 Notifications of DSSP Service State Modifications

DSSP defines an event as a state change in a service. For example, in the case of an `UPDATE` operation on a service, the state of that service changes as a direct result of that `UPDATE` operation. Furthermore, the `UPDATE` operation itself directly represents the state change and so it is natural to think of the event notification as simply being the `UPDATE` operation (see Figure 5).

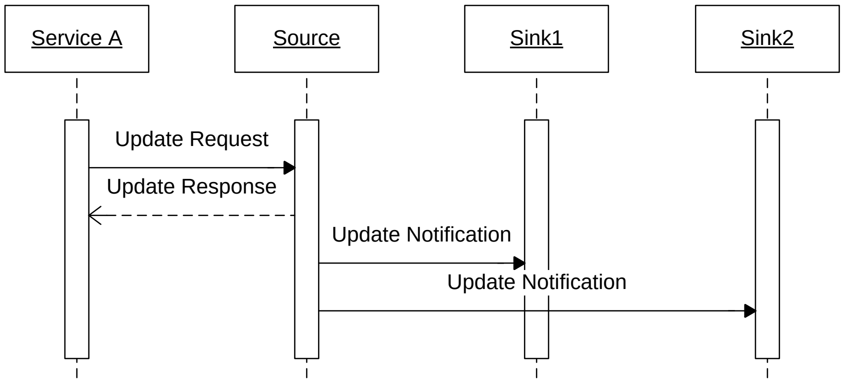

Figure 5 The Update request is directly representing the state change on a service.

A service can receive event notifications by subscribing to a service (the publisher) using the `SUBSCRIBE` operation (see section 5.10). While the `SUBSCRIBE` operation is abstract and can be tailored to a particular service, it follows the common subscription model described here.

When a service subscribes to a publisher, any existing matches to the subscription are sent to the subscription sink allowing the sink to "catch up" to the current state of the publisher. The use of service-specific subscriptions enables filtered subscriptions using queries over the service state for a particular publisher. As for queries, DSSP does not mandate or require a particular query language for subscriptions.

Event notifications are one-way messages and are not acknowledged by the receiver. In order to distinguish event notifications from the state changing operations themselves, the generated events are marked as event notifications using a special SOAP Action value (see section 6).

Although there are similarities between a subscription and the `QUERY` operation, an important difference is that a subscription results in an event notification being generated when the subscription can be fulfilled. A `QUERY` operation can succeed even if no match was found and the `QUERY` response is empty.

4 DSSP Protocol Model

DSSP communication happens between two services (see section 2) where the sending party is called the *initial sender* and the receiving party is called the *ultimate destination*.

DSSP messages are carried as part of the `S:Body` element information item of a SOAP envelope. All DSSP message types are identified by unique SOAP Action values enabling inspection at two levels:

- Intermediaries and other SOAP nodes that do not care about the specific type of a SOAP Body or that cannot inspect the contents of the SOAP Body if it has been encrypted can use the SOAP Action value.
- SOAP nodes that do have access to the SOAP Body can inspect the entire contents.

4.1 Message Exchange Patterns

DSSP operations are either one-way or single request/response interactions, meaning that no state is maintained in the communication channel between two services beyond what is necessary to correlate a request with a response. While some operations only support one message exchange pattern, others can support both; depending on whether a response is desired or not.

DSSP does not define any constraints on how long a DSSP operation can take to complete. For operations that take significant amounts of time, a work-queue-style service supporting the DSSP event notification model may be used to indicate the current state processing a work item (see section 3.6).

4.1.1 One-way

The one-way message exchange pattern consists of a single message sent from an initial sender to an ultimate receiver (see Figure 6):

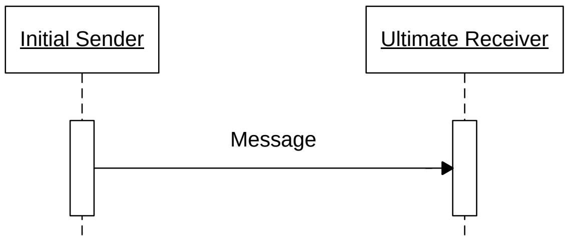

Figure 6 One-way message exchange pattern.

If a one-way message results in the generation of a SOAP fault, no SOAP Fault message is returned to the initial sender (see section 4.2).

4.1.2 Request-Response

The request-response, MEP (Message Exchange Pattern), consists of a single request message sent from an initial sender to an ultimate receiver, followed by a single response sent from the ultimate receiver to the initial sender of the request (see Figure 7).

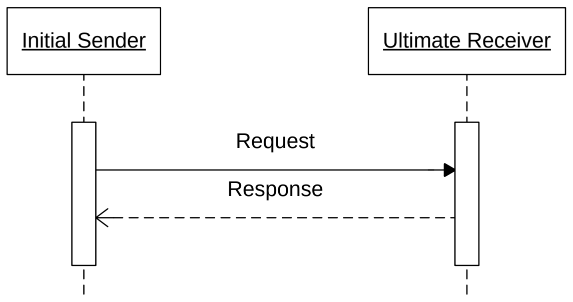

Figure 7 Request-response message exchange pattern.

No time constraints are imposed on this message exchange pattern and no expectation is made whether the operations are implemented used synchronous or asynchronous programming models. A robust implementation needs to take timeouts into account to avoid exhausting resources and opening your system to DOS (Denial-Of-Service) attacks.

If the request message results in the generation of a SOAP fault, a SOAP Fault message is returned to the initial sender (see section 4.2).

4.2 Fault Handling

Following the SOAP 1.2 processing rules, failure to successfully process a SOAP message results in the generation of a SOAP fault as described in [7], section 5.4. The rules for handling SOAP faults in DSSP are as follows:

1. If processing a DSSP request results in the generation of a SOAP fault then it MUST be sent in a DSSP response if a response is requested by the sender.
2. If processing a DSSP response or a one-way message results in the generation of a SOAP fault then no claims are made as to the handling of such faults.

Regardless of how a SOAP fault is generated, it MUST NOT be sent in response to another SOAP fault since doing so can result in fault storms.

4.2.1 Generic DSSP Faults

DSSP follows the SOAP 1.2 fault model of a "code" followed by one or more "subcodes"; where the "code" is dictated by the SOAP 1.2 specification and DSSP provides a set of "subcode" values (see section 1.3 for notational conventions). In addition to specific SOAP faults defined in section 5, DSSP operations can result in the following SOAP faults:

- **S:Receiver/DSSP:OperationFailed**
  The operation could not be successfully completed due to a receiver side problem handling the DSSP request.
- **S:Sender/DSSP:OperationFailed**
  The operation could not be successfully completed due to a sender side problem in the DSSP request.
- **S:Receiver/DSSP:InsufficientResources**
  The operation could not be successfully completed due to insufficient resources on the receiver side.
- **S:Sender/DSSP:ActionNotSupported**
  The DSSP operation is not supported by the receiver.

- **S:Sender/DSSP:MessageNotSupported**
  The DSSP operation is supported but the DSSP message type is not.

5 DSSP Message Operations

A DSSP operation involves exchanging messages of specific types using one of the message exchange patterns defined in section 4.1.

With the exception of the `LOOKUP` operation, all operations are optional. DSSP services MUST support the `LOOKUP` operation (see section 5.7).

5.1 Safe and Idempotent Operations

In order to distinguish operations that strictly retrieve information from operations that manipulate information, this specification uses the terms "safe" and "idempotent" as qualifiers on DSSP operations:

- **Idempotent**
  An operation is idempotent when the result of a successful execution is independent of the number of times the operation is executed.
- **Safe**
  An operation is safe if its semantics are limited to read-only access to the resource to which it is applied.

A safe operation is idempotent by definition; the reverse is not true. Read-only operations are both safe and idempotent.

A particular implementation of a safe or idempotent operation might have second-order side effects associated with executing the operation. Typical examples of such side effects are logging, processing associated with generating the read-only data in the case of dynamic contents, and so on. Such second-order side effects do not affect the notion of safe and idempotent operations.

5.2 CREATE

Services can be created using the `CREATE` operation as described in section 3.1.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:createrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:createresponse
Safe	No
Idempotent	No
Generates Event	No

Table 2 Property sheet for CREATE

In addition to the generic faults (see section 4.2.1), this operation can result in the following SOAP faults:

- **S:Sender/DSSP:UnknownContract**
  If the constructor service cannot create a service of the given type then it should generate a `S:Sender/DSSP:UnknownContract` SOAP fault.

5.3 DELETE

Service state can be deleted using the `DELETE` operation as described in section 3.5.2.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:deleterequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:deleteresponse
Safe	No
Idempotent	No
Generates Event	Yes

Table 3 Property sheet for DELETE

In addition to the generic faults (see section 4.2.1), this operation can result in the following SOAP faults:

- **S:Sender/DSSP:UnknownEntry**
  If the service does not have the state which the `DELETE` operation is requesting be deleted then the service should generate a `S:Sender/DSSP:UnknownEntry` SOAP fault.

5.4 DROP

The `DROP` operation can be used to terminate a service as described in section 3.2.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:droprequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:dropresponse
Safe	No
Idempotent	No
Generates Event	No

Table 4 Property sheet for DROP

There are no SOAP faults specific to the `DROP` operation.

5.5 GET

The `GET` operation can be used to retrieve the complete service state as described in section 3.4.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:getrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:getresponse
Safe	Yes
Idempotent	Yes
Generates Event	No

Table 5 Property sheet for GET

There are no SOAP faults specific to the `GET` operation.

5.6 INSERT

The `INSERT` operation can be used to add to the service state as described in section 3.5.1.

Property	Value
MEP	One-way, Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:insertrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:insertresponse
Safe	No
Idempotent	No
Generates Event	Yes

Table 6 Property sheet for INSERT

In addition to the generic faults (see section 4.2.1), this operation can result in the following SOAP faults:

- **S:Sender/DSSP:DuplicateEntry**
  If a service has uniqueness constraints on its service state and the state carried in an `INSERT` request conflicts with those constraints then it SHOULD generate a `S:Sender/DSSP:DuplicateEntry` SOAP fault.

5.7 LOOKUP

The `LOOKUP` operation can be used to retrieve the service type and context as described in section 3.3. All DSSP applications MUST support this operation.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:lookuprequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:lookupresponse
Safe	Yes
Idempotent	Yes
Generates Event	No

Table 7 Property sheet for LOOKUP

There are no SOAP faults specific to the `LOOKUP` operation.

5.8 QUERY

The `QUERY` operation can be used to query the service state as described in section 3.4.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:queryrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:queryresponse
Safe	Yes
Idempotent	Yes
Generates Event	No

Table 8 Property sheet for QUERY

There are no SOAP faults specific to the `QUERY` operation.

5.9 REPLACE

The `REPLACE` operation can be used to replace all existing service state of a service as described in 3.5.3.

Property	Value
MEP	One-way, Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:replacerequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:replaceresponse
Safe	No
Idempotent	Yes
Generates Event	Yes

Table 9 Property sheet for REPLACE

The `REPLACE` operation may result in one of the generic SOAP faults (see section 4.2.1).

5.10 SUBSCRIBE

The `SUBSCRIBE` operation can be used to subscribe to changes in the state of a service as described in section 3.6.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:subscriberequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:subscriberesponse
Safe	No
Idempotent	No
Generates Event	No

Table 10 Property sheet for SUBSCRIBE

In addition to the generic faults (see section 4.2.1), a `SUBSCRIBE` operation may generate the following SOAP fault:

- **S:Sender/DSSP:DuplicateSubscriber**
  If a service is already registered as a sink by the subscription manager the `SUBSCRIBE` operation should generate a `S:Sender/DSSP:DuplicateSubscriber` SOAP fault.

5.11 SUBMIT

The `SUBMIT` operation is intended for capturing semantics that is not possible or practical to represent as state manipulations. The `SUBMIT` operation is explicitly defined to allow computations that do not change the state of a service.

Property	Value
MEP	Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:submitrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:submitresponse
Safe	No
Idempotent	No
Generates Event	No

Table 11 Property sheet for SUBMIT

There are no SOAP faults specific to the `SUBMIT` operation.

5.12 UPDATE

The `UPDATE` operation can be used to update existing service state as described in 3.5.3.

Property	Value
MEP	One-way, Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:updaterequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:updateresponse
Safe	No
Idempotent	No
Generates Event	Yes

Table 12 Property sheet for UPDATE

In addition to the generic faults (see section 4.2.1), this operation can result in the following SOAP faults:

- **S:Sender/DSSP:DuplicateEntry**
  If a service has uniqueness constraints on its service state and the state carried in an `UPDATE` request conflicts with those constraints, it should generate a `S:Sender/DSSP:DuplicateEntry` SOAP fault.
- **S:Sender/DSSP:UnknownEntry**
  If the service does not have the state for which the operation is requesting an update, then the service should generate a `S:Sender/DSSP:UnknownEntry` SOAP fault.

5.13 UPSERT

The `UPSERT` operation can be used to update existing state or add new state as described in 3.5.3.

Property	Value
MEP	One-way, Request-Response
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:upsertrequest
Response Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:upsertresponse
Safe	No
Idempotent	No
Generates Event	Yes

Table 13 Property sheet for UPSERT

In addition to the generic faults (see section 4.2.1), this operation can result in the following SOAP faults:

- **S:Sender/DSSP:DuplicateEntry**
  If a service has uniqueness constraints on its service state and the state carried in an `UPSERT` request conflicts with those constraints then it should generate a `S:Sender/DSSP:DuplicateEntry` SOAP fault.

6 DSSP Event notifications

As described in section 3.6, an event notification is a one-way message containing a DSSP request classified as **`Generates Event`** with the properties specified in Table 14.

Property	Value
MEP	One-way
Request Action	http://schemas.microsoft.com/xw/2004/10/dssp.html:notify
Safe	Yes
Idempotent	Yes
Generates Event	No

Table 14 Property sheet for event notifications

There are no SOAP faults specific to event notifications.

7 References

[1] IETF RFC 2119: "Keywords for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. (See http://www.ietf.org/rfc/rfc2119.txt.)
[2] IETF RFC 2396: "Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, R. Fielding, L. Masinter, August 1998. (See http://www.ietf.org/rfc/rfc2396.txt.)
[3] IETF RFC 2616: "Hypertext Transfer Protocol -- HTTP/1.1", R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, January 1997. (See http://www.ietf.org/rfc/rfc2616.txt.)
[4] W3C Recommendation "XML Information Set", John Cowan, Richard Tobin, 24 October 2001. (See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/.)
[5] W3C Recommendation "XML Schema Part 1: Structures", Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, 2 May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.)
[6] W3C Recommendation "XML Schema Part 2: Datatypes", Paul V. Biron, Ashok Malhotra, 2 May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.)
[7] W3C Recommendation "SOAP Version 1.2 Part 1: Messaging Framework", M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. F. Nielsen, May 2003. (See http://www.w3.org/TR/soap12-part1/.)
[8] Microsoft Robotics Studio Development Center (See http://www.microsoft.com/robotics)