Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.

- **Copyrights.** This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL’s, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.

- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.

- **Patents.** Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft Open Specification Promise or the Community Promise. If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.

- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.

- **Fictitious Names.** The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

**Reservation of Rights.** All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

**Tools.** The Open Specifications do not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them. Certain Open Specifications are intended for use in conjunction with publicly available standard specifications and network programming art, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.
## Revision Summary

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision History</th>
<th>Revision Class</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/06/2009</td>
<td>0.1</td>
<td>Major</td>
<td>Initial Availability</td>
</tr>
<tr>
<td>02/19/2010</td>
<td>1.0</td>
<td>Major</td>
<td>Updated and revised the technical content</td>
</tr>
<tr>
<td>03/31/2010</td>
<td>1.01</td>
<td>Editorial</td>
<td>Revised and edited the technical content</td>
</tr>
<tr>
<td>04/30/2010</td>
<td>1.02</td>
<td>Editorial</td>
<td>Revised and edited the technical content</td>
</tr>
<tr>
<td>06/07/2010</td>
<td>1.03</td>
<td>Editorial</td>
<td>Revised and edited the technical content</td>
</tr>
<tr>
<td>06/29/2010</td>
<td>1.04</td>
<td>Editorial</td>
<td>Changed language and formatting in the technical content.</td>
</tr>
<tr>
<td>07/23/2010</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>09/27/2010</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>11/15/2010</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>12/17/2010</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>03/18/2011</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>06/10/2011</td>
<td>1.04</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>01/20/2012</td>
<td>1.5</td>
<td>Minor</td>
<td>Clarified the meaning of the technical content.</td>
</tr>
<tr>
<td>04/11/2012</td>
<td>1.5</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
<tr>
<td>07/16/2012</td>
<td>1.5</td>
<td>No change</td>
<td>No changes to the meaning, language, or formatting of the technical content.</td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction .................................................................................................................. 7
   1.1 Glossary .................................................................................................................. 7
   1.2 References ............................................................................................................. 7
      1.2.1 Normative References .................................................................................... 7
      1.2.2 Informative References .................................................................................. 7
   1.3 Protocol Overview (Synopsis) ................................................................................. 8
      1.3.1 Remote Method Call Model ............................................................................ 8
      1.3.2 Localizing and Binding Servers ..................................................................... 9
      1.3.3 Fundamental Interfaces .................................................................................. 10
   1.4 Relationship to Other Protocols ............................................................................. 10
   1.5 Prerequisites/Preconditions .................................................................................... 11
   1.6 Applicability Statement .......................................................................................... 11
   1.7 Versioning and Capability Negotiation ................................................................. 11
   1.8 Vendor-Extensible Fields ...................................................................................... 11
   1.9 Standards Assignments .......................................................................................... 11

2 Messages .......................................................................................................................... 12
   2.1 Transport ................................................................................................................ 12
   2.2 Common Data Types ............................................................................................... 12
      2.2.1 Float ............................................................................................................... 13
      2.2.2 Void ............................................................................................................... 13
      2.2.3 LengthPrefixedByteSequence ..................................................................... 13
      2.2.4 String ............................................................................................................ 13
      2.2.5 LengthPrefixedInt32Sequence .................................................................... 14
      2.2.6 LengthPrefixedInt64Sequence .................................................................... 14
      2.2.7 LengthPrefixedStringSequence .................................................................. 14
      2.2.8 LengthPrefixedFloatSequence ................................................................... 15
      2.2.9 OutputValue ................................................................................................... 15
      2.2.10 CallResult .................................................................................................... 16
      2.2.11 CheetahValue .............................................................................................. 16
      2.2.12 SystemException ......................................................................................... 17
      2.2.13 UserException .............................................................................................. 17
      2.2.14 AbstractObjectReference .......................................................................... 17
      2.2.15 CallArguments .............................................................................................. 18
      2.2.16 ServerObjectURI .......................................................................................... 19
      2.2.17 ServerMethodURI ......................................................................................... 19
      2.2.18 cht::nameservermsg::aor .......................................................................... 20
      2.2.19 cht::nameservermsg::aor_list .................................................................... 20
      2.2.20 nameservice::nameserver::not_bound_exception ...................................... 21
      2.2.21 nameservice::nameserver::resolve_exception ......................................... 21
      2.2.22 cht::core::resource_report ........................................................................ 21
      2.2.23 cht::core::alloc ......................................................................................... 21
      2.2.24 cht::core::scope ......................................................................................... 22
      2.2.25 cht::core::named_value .............................................................................. 22
      2.2.26 cht::core::bool_value ............................................................................... 23
      2.2.27 cht::core::float_value ................................................................................ 23
      2.2.28 cht::core::string_value .............................................................................. 23
      2.2.29 cht::core::long_value ................................................................................. 23
      2.2.30 cht::core::longlong_value ......................................................................... 24
      2.2.31 core::lifecycle::state ................................................................................. 24
3 Protocol Details

3.1 Common Middleware Details

3.1.1 Abstract Data Model

3.1.2 Timers

3.1.3 Initialization

3.1.4 Message Processing Events and Sequencing Rules

3.1.4.1 FSIDL Specifications

3.1.4.2 Mapping FSDL MethodDecl to Remote Method Specifications

3.1.4.3 Mapping Remote Method Request

3.1.4.4 Mapping Remote Method Reply

3.1.4.5 Mapping FSDL AtomicType

3.1.4.6 Mapping FSDL SequenceType

3.1.4.7 Mapping FSDL EnumName

3.1.4.8 Mapping FSDL CheetahEntityName

3.1.4.9 Mapping FSDL InterfaceName

3.1.4.10 Mapping FSDL ExceptionName

3.1.5 Timer Events

3.1.6 Other Local Events

3.2 Middleware Server Details

3.2.1 Abstract Data Model

3.2.2 Timers

3.2.3 Initialization

3.2.4 Message Processing Events and Sequencing Rules

3.2.4.1 Remote Method Invocation

3.2.4.2 __ping

3.2.5 Timer Events

3.2.6 Other Local Events

3.3 Middleware Client Details

3.3.1 Abstract Data Model

3.3.2 Timers

3.3.3 Initialization

3.3.4 Message Processing Events and Sequencing Rules

3.3.4.1 Remote Method Invocation

3.3.5 Timer Events

3.3.6 Other Local Events

3.4 Name Server Server Details

3.4.1 Abstract Data Model

3.4.2 Timers

3.4.3 Initialization

3.4.4 Message Processing Events and Sequencing Rules

3.4.4.1 resolve

3.4.4.2 bind

3.4.4.3 unbind

3.4.4.4 list_any

3.4.4.5 list_host

3.4.4.6 list_name

3.4.5 Timer Events

3.4.6 Other Local Events

3.5 Name Server Client Details

3.5.1 Abstract Data Model

3.5.2 Timers

3.5.3 Initialization

3.5.4 Message Processing Events and Sequencing Rules
1 Introduction

The Middleware Protocol provides a mechanism for an implementation to call methods that are located in a different address space over the network. A protocol client constructs parameters that it sends to the protocol server as part of the call message. The protocol server sends a return value to the protocol client in the response. In addition to the basic data types, applications exchange information encoded in the Cheetah data model. For more information, see [MS-FSCHT].

Sections 1.8, 2, and 3 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in RFC 2119. Sections 1.5 and 1.9 are also normative but cannot contain those terms. All other sections and examples in this specification are informative.

1.1 Glossary

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

- authentication
- certificate
- credential
- Hypertext Transfer Protocol (HTTP)
- Hypertext Transfer Protocol over Secure Sockets Layer (HTTPS)
- Kerberos
- NT LAN Manager (NTLM) Authentication Protocol
- UTF-8

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

- abstract object reference (AOR)
- channel URI
- Cheetah checksum
- Cheetah entity
- client proxy
- FAST Search Interface Definition Language (FSIDL)
- host name
- name server
- server interface

The following terms are specific to this document:

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

1.2 References

References to Microsoft Open Specifications documentation do not include a publishing year because links are to the latest version of the technical documents, which are updated frequently. References to other documents include a publishing year when one is available.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site,
1.2.2 Informative References


1.3 Protocol Overview (Synopsis)

This protocol specifies a mechanism for an application to call methods that are located in different address spaces over the network. Input values to the method are sent as part of the call message and return values are sent in the response. In addition to the basic data types, applications exchange information encoded as Cheetah entities. The procedure for serializing Cheetah entities is described in [MS-FSCHT].

This section presents a brief overview of the following:

- The remote method call model.
- Localizing and binding protocol servers.
- Fundamental interfaces.

1.3.1 Remote Method Call Model

This protocol specifies how to call a method when the calling application and the target method are located in different address spaces. The following figure is an example.
Figure 1: Middleware Protocol

This protocol specifies two roles: protocol client and protocol server. The protocol client initiates communication by calling a remote method with input values and a client proxy. The server object receives the request from the client proxy, processes the method as specified in the parameters in the request, and sends a return value to the client proxy, which then sends it to the protocol client.

The remote method sends user exceptions to the protocol client when errors occur during processing in the implementation-specific part of the protocol server. Generic protocol server errors such as connection errors, data validity errors, and availability errors are returned to the protocol client as system exceptions. Server objects and client proxies are represented as abstract object references (AORs) when associated with remote methods as parameters or return values. The following figure shows a protocol client that sends an abstract object reference to a protocol server, where the protocol server uses the abstract object reference to call back to the protocol client.

Figure 2: An abstract object reference sent by the protocol client and then used by the protocol server to call back the protocol client

1.3.2 Localizing and Binding Servers

A protocol client requires an abstract object reference to create a client proxy that communicates with a specified server object. When a server object is instantiated, a server object URI is constructed from the AOR, and includes information such as the network host name and port of the server object. Likewise, client proxies create a server method URI from the AOR to call a remote method.

A name server associates logical names with server objects so that protocol clients and protocol servers do not explicitly manage AORs. Protocol clients contact the name server to request a server object specified by its logical name. An example is shown in the following figure, where the protocol
client looks up the reference to the "Hello" server object in the name server, creates a client proxy based on the AOR and calls the `SayHello` method on the server object.

![Diagram](image)

**Figure 3: Binding a server object to the logical name "Hello" in the name server**

By using the name server, protocol clients and protocol servers explicitly manage only the AOR that represents the name server. They use the name server to localize all other server objects. The **server interface** implemented by the name server is described in section 3.4.

### 1.3.3 Fundamental Interfaces

In addition to the name server, this protocol specifies two fundamental server interfaces that are implemented by applications that use this protocol. The first interface specifies controls and queries for runtime states, including whether the protocol server is running, suspended, terminating, or stopped. This interface is described in section 3.8.

The second interface queries the protocol server for runtime statistics such as the average duration of method processing, or the values of implementation-specific parameters. The interface is described in section 3.6.

### 1.4 Relationship to Other Protocols

This protocol specifies how to convert a remote method into an exchange of encoded messages. User applications are layered on top of this protocol and use its services for application-specific purposes.

This protocol depends on other structures and protocols to encode and transport its messages. Cheetah entities specify additional data types to encode nested and tree-structured data structures. Transmission on the wire is performed using the Hypertext Transfer Protocol (HTTP) or the Hypertext Transfer Protocol over the Secure Sockets Layer (HTTPS), and protocol client
authentication (2) is optionally done through either the NT LAN Manager (NTLM) Authentication Protocol (NTLM) or Kerberos.

1.5 Prerequisites/Preconditions

The protocol server deploys a certificate (1) if the HTTPS transport is used. Typically, protocol clients and protocol servers are deployed with a name server protocol server to avoid managing AORs explicitly.

This protocol does not specify any means to activate a protocol server or protocol client. The protocol server is configured to listen on a channel URI as specified by the implementation. For more information about channel URIs, see section 3.2.1, Abstract Data Model.

Protocol clients and protocol servers need to agree on the remote method specifications and the Cheetah entity specifications.

1.6 Applicability Statement

This protocol calls remote methods in a distributed environment. It is designed for use on private networks, and is not appropriate for use on public networks. For more information, see section 5.1.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- **Supported transports**: This protocol can be implemented on top of HTTP/HTTPS.
- **Protocol versions**: There is only one version of this protocol.
- **Security and authentication methods**: The protocol relies on the security provided by HTTPS, NTLMv1, NTLMv2, and Kerberos. The protocol does not have any security provisions of its own.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

None.
2 Messages

2.1 Transport

This protocol uses HTTP transport as specified in [RFC2616] to transmit method requests and responses. The protocol client sends a message as part of an HTTP request, and the protocol server replies with an HTTP response. Port 80 is the standard port assignment for HTTP and port 443 is the standard port assignment for HTTPS. Use of other ports is implementation-dependent.

If the application that calls this protocol requires NTLM authentication [MS-NLMP] or Kerberos authentication [MS-KILE], the application MUST provide implementation-specific credentials as either a user name/password or a certificate. This protocol MUST NOT process the credentials or authentication information, because processing is implementation-dependent.

The calling application MUST specify the maximum number of octets that the HTTP request and response message body can contain.

2.2 Common Data Types

This section specifies the structures of the common types that are supported by this protocol. A protocol type is identified by a case-sensitive name, and specifies the structure of data. This protocol supports the BYTE, INT32 and INT64 types specified in [MS-DTYP] in addition to single-precision IEEE floating-point. The byte-ordering of the INT32 and INT64 data types MUST be big-endian. The signed data types use two's complement to represent the negative numbers.

The cht::core Cheetah entities specified by this protocol and the corresponding Cheetah type identifiers are specified in the following table.

<table>
<thead>
<tr>
<th>cht::core Cheetah entities</th>
<th>Cheetah type identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc</td>
<td>0</td>
</tr>
<tr>
<td>named_value</td>
<td>1</td>
</tr>
<tr>
<td>bool_value</td>
<td>2</td>
</tr>
<tr>
<td>scope</td>
<td>5</td>
</tr>
<tr>
<td>resource_report</td>
<td>6</td>
</tr>
<tr>
<td>float_value</td>
<td>8</td>
</tr>
<tr>
<td>long_value</td>
<td>10</td>
</tr>
<tr>
<td>string_value</td>
<td>11</td>
</tr>
<tr>
<td>longlong_value</td>
<td>12</td>
</tr>
</tbody>
</table>

The Cheetah checksum for cht::core entities MUST be -1479218033.

The cht::nameservermsg Cheetah entities specified by this protocol and the corresponding Cheetah type identifiers are specified in the following table.

<table>
<thead>
<tr>
<th>cht::nameservermsg Cheetah entities</th>
<th>Cheetah type identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>aor</td>
<td>0</td>
</tr>
</tbody>
</table>
cht::nameservermsg Cheetah entities | Cheetah type identifier
---|---
aor_list | 1

The Cheetah checksum for `cht::nameserver` entities MUST be 277807848.

### 2.2.1 Float

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
| Value |

**Value (4 bytes):** A 32-bit single-precision floating-point field, as specified in [IEEE754]. Float values range from negative 3.402823e38 to positive 3.402823e38.

### 2.2.2 Void

This represents an empty data value.

### 2.2.3 LengthPrefixedByteSequence

This represents a sequence of `BYTE` values.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
| Length |

**ByteSequence (variable):** A sequence of `BYTE` values. The number of `BYTE` values MUST be equal to the `Length` field.

### 2.2.4 String

This represents a UTF-8 encoded string, and is prefixed by a value that specifies the number of `BYTE`es that represent the encoded string.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
| Length |

**ByteSequence (variable):**
Length (4 bytes): An **INT32** that represents the length of the sequence. The value MUST be 0 or a positive number.

**ByteSequence (variable):** A sequence of **BYTE** values. The number of **BYTE** values MUST be equal to the **Length** field.

### 2.2.5 LengthPrefixedInt32Sequence

This represents a sequence of **INT32** values.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
| Length |
| Int32Sequence (variable) |
| ... |

Length (4 bytes): An **INT32** that represents the length of the sequence. The value MUST be 0 or a positive number.

**Int32Sequence (variable):** A sequence of **INT32** values. The number of **INT32** values is specified in the **Length** field.

### 2.2.6 LengthPrefixedInt64Sequence

This represents a sequence of **INT64** values.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
| Length |
| Int64Sequence (variable) |
| ... |

Length (4 bytes): An **INT32** that represents the length of the sequence. The value MUST be 0 or a positive number.

**Int64Sequence (variable):** A sequence of **INT64** fields. The number of **INT64** fields is specified in the **Length** field.

### 2.2.7 LengthPrefixedStringSequence

This represents a sequence of **String** fields. The length MUST be of type **INT32**, and specifies the number of **String** elements in the sequence.
Length (4 bytes): An INT32 that represents the length of the sequence. The value MUST be 0 or a positive number.

StringSequence (variable): A sequence of String fields, as specified in section 2.2.4. The number of String fields is specified in the Length field.

2.2.8 LengthPrefixFixedFloatSequence

This represents a sequence of Float values.

Length (4 bytes): An INT32 that represents the length of the sequence. The value MUST be 0 or a positive number.

FloatSequence (variable): A sequence of Float values, as specified in section 2.2.1. The number of Float values is specified in the Length field.

2.2.9 OutputValue

This represents the protocol server response resulting from a remote method invocation at the server object.

ReturnType (1 byte): A BYTE value that MUST contain the value 48, 49, or 50. If the value is 48, the MessageContent field contains a CallResult message. If the value is 49, the MessageContent field contains a UserException. If the value is 50, the MessageContent field contains a SystemException.
**MessageContent (variable):** MUST contain a **CallResult**, **UserException**, or **SystemException** record.

### 2.2.10 CallResult

This represents the result value from invoking a remote method call.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |

**ResultContents (variable):**

```
...```

**ResultContents (variable):** MUST be a field of one of the following types:

- Void
- BYTE
- INT32
- INT64
- Float
- String
- LengthPrefixedByteSequence
- LengthPrefixedStringSequence
- LengthPrefixedInt32Sequence
- LengthPrefixedInt64Sequence
- LengthPrefixedFloatSequence
- AbstractObjectReference
- CheetahValue

### 2.2.11 CheetahValue

This represents the Cheetah entity in serialized form, as specified in [MS-FSCHT] section 2.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |

**CheetahValueContents (variable):**

```
...```

**CheetahValueContents (variable):** MUST be a single Cheetah entity.
2.2.12 SystemException

This represents a system exception thrown by the protocol server. The `SystemException` record is identified with a `Name` field and a `Description` field.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Name (variable) |
| ... |
| Description (variable) |
| ... |

**Name (variable):** A `String` field that MUST contain the value "system_exception".

**Description (variable):** A `String` field that contains the description of the system exception.

2.2.13 UserException

This represents an exception thrown by the implementation-specific part of a remote method in a server object. The `UserException` record is identified with a `Name` field and, optionally, a set of `Attributes` fields.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 0 | 1 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Name (variable) |
| ... |
| Attributes (variable) |
| ... |

**Name (variable):** A `String` field that represents the name of the `UserException`.

**Attributes (variable):** This contains 0 or more fields of type `BYTE`, `INT32`, `INT64`, `Float`, or `String`. If no fields are represented, the `Attributes` field MUST contain the value `Void`. Values are not padded to a byte boundary when more than 1 value is represented.

2.2.14 AbstractObjectReference

This represents a server object that is sent between a protocol client and a protocol server.
2.2.15 CallArguments

This represents the input values specified by the protocol client. The input values are specified by one or more Arguments fields.

Arguments (variable): Contains 0 or more values of one of the following types:

- BYTE
- INT32
- INT64
• Float
• String
• CheetahValue
• AbstractObjectReference
• LengthPrefixedByteSequence
• LengthPrefixedInt32Sequence
• LengthPrefixedInt64Sequence
• LengthPrefixedFloatSequence
• LengthPrefixedStringSequence

If no values are represented, the Arguments field MUST be the Void value. Values are not padded when more than 1 value is represented.

2.2.16 ServerObjectURI

This represents the server object URI. The server object URI is the path segments of the HTTP Request-URI that identifies a server object, and is thus a prefix of the ServerMethodURI. The ServerObjectURI is a URI path that consists of the following three path segments, delimited by a slash (/):

InterfaceType: Represents the protocol server interface.
InterfaceVersion: Represents the server interface version.
ServerObjectId: A 64-bit number in decimal digit form that represents the server object identifier.

The following string pattern using Augmented Backus-Naur Form (ABNF) syntax specified in [RFC5234] specifies the ServerObjectURI:

```
InterfaceType     = 1*(ALPHA / DIGIT / ".") "::" 1*(ALPHA / DIGIT / ".")
InterfaceVersion  = 1*DIGIT "." 1*DIGIT
ServerObjectId    = 1*DIGIT
ServerObjectURI   = InterfaceType "/" InterfaceVersion "/" ServerObjectId
```

For example, in the ServerMethodURI string "core::fds_component/1.2/42", the InterfaceType field is "core::fds_component", the InterfaceVersion field contains "1.2", and the ServerObjectId field is 42.

2.2.17 ServerMethodURI

This represents a specific remote method. It MUST contain a ServerObjectURI that specifies which server object contains the remote method.

MethodName: Represents the name of a remote method.

ServerMethodURI: URI specified in the following string pattern, using Augmented Backus-Naur Form (ABNF) syntax specified in [RFC5234].
MethodName        =  1*(ALPHA / DIGIT / "_")
ServerMethodURI   =  ServerObjectURI "/" MethodName

For example, in the ServerMethodURI string "core::fds_component/1.2/42/get_resource_report", the MethodName is "get_resource_report" and the ServerObjectURI is "core::fds_component/1.2/42".

2.2.18  cht::nameservermsg::aor
This represents an abstract object reference, and is specified by the following Cheetah entity:

root entity aor
{
    attribute string host;
    attribute int port;
    attribute string interface_type;
    attribute string interface_version;
    attribute longint object_id;
    attribute string bound_name;
};

host: A string that represents the host name of the server object.
port: A field of type int that represents the port number of the server object.
interface_type: A string that represents the interface of the server object.
interface_version: A string that represents the version of the interface.
object_id: A field of type longint that represents the identifier of the server object.
bound_name: A string that represents the name field in the logical name associated with this abstract object reference, as specified in section 3.4.1.

The cht::nameservermsg::aor attributes are mapped to AbstractObjectReference fields except the bound_name attribute, as specified in section 2.2.14, and in the following table.

<table>
<thead>
<tr>
<th>cht::nameservermsg::aor attributes</th>
<th>AbstractObjectReference fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Host</td>
</tr>
<tr>
<td>Port</td>
<td>Port</td>
</tr>
<tr>
<td>interface_type</td>
<td>InterfaceType</td>
</tr>
<tr>
<td>interface_version</td>
<td>InterfaceVersion</td>
</tr>
<tr>
<td>object_id</td>
<td>ServerObjectId</td>
</tr>
</tbody>
</table>

The Cheetah entity attributes are the same as the corresponding AbstractObjectReference fields.

2.2.19  cht::nameservermsg::aor_list
This represents a collection of cht::nameservermsg::aor, and is specified by the following Cheetah entity:
root entity aor_list
{
    collection aor aors;
};

aors: A collection of cht::nameservermsg::aor Cheetah entities, as specified in section 2.2.18.

2.2.20 nameservice::nameserver::not_bound_exception
This states that there is no association between a given logical name, as specified in section 3.4.1, and an AOR in the name server. The FAST Search Interface Definition Language (FSIDL) specification for the exception is as follows:

    exception not_bound_exception {};

2.2.21 nameservice::nameserver::resolve_exception
This states that a specified Logical Name, as specified in section 3.4.1, does not exist in the name server. The FSIDL specification for the exception is as follows:

    exception resolve_exception {};

2.2.22 cht::core::resource_report
This represents a resource allocation report based on the resource allocation table, resource scope table, and resource value table of the protocol server, as specified in section 3.6.1. The cht::core::resource_report is specified by the following Cheetah entity:

    root entity resource_report {
        attribute longint when;
        collection alloc allocs;
        collection scope scopes;
        collection named_value values;
    };

    when: A field of type longint that represents the time in number of seconds since January 1, 1970.
allocs: A collection of cht::core::alloc Cheetah entities, as specified in section 2.2.23.
scopes: A collection of cht::core::scope Cheetah entities, as specified in section 2.2.24.
values: A collection of cht::core::named_value Cheetah entities, as specified in section 2.2.25.

2.2.23 cht::core::alloc
This represents an implementation-specific counter that counts named resources such as files or memory units in the server object. The content is based on an entry of the resource allocation table, as specified in section 3.6.1. The cht::core::alloc is specified by the following Cheetah entity:

    entity alloc {
        attribute string name;
        attribute int current;
    };
attribute int total;
};

name: A field of type string that represents the name of the resource allocation.
current: A field of type int that represents the current number of resource allocations.
total: A field of type int that represents the total number of resource allocations.

2.2.24  cht::core::scope

This represents an implementation-specific name specified by a well-formed set of programming language statements, typically in a function or method within the server object. The content of a resource scope is based on an entry in the resource scope table, as specified in section 3.6.1. The cht::core::scope is specified by the following Cheetah entity:

entity scope {
    attribute string name;
    attribute int current;
    attribute int total;
    attribute int min_time;
    attribute int max_time;
    attribute int avg_time;
};

name: A string that represents the name of the resource scope.
current: A field of type int that represents the current number of calls for this resource scope.
total: A field of type int that represents the total number of calls for this resource scope.
min_time: A field of type int that represents the minimum time in milliseconds used for a call of this resource scope.
max_time: A field of type int that represents the maximum time in milliseconds used for a call of this resource scope.
avg_time: A field of type int that represents the average time in milliseconds used for a call of this resource scope.

2.2.25  cht::core::named_value

This represents a resource value, which is an implementation-specific field that is associated with a unique name within the server object. The content is based on an entry in the resource value table, as specified in section 3.6.1.

The cht::core::named_value Cheetah entity is subtyped by the following Cheetah entities:

- cht::core::bool_value
- cht::core::float_value
- cht::core::string_value
- cht::core::long_value
• `cht::core::longlong_value`

The `cht::core::named_value` is specified by the following Cheetah entity:

```cpp
entity named_value {
    attribute string name;
};
```

**name:** A string that represents the name of the resource value.

### 2.2.26 `cht::core::bool_value`

This is a subtype of the `cht::core::named_value` that represents a resource value of type `BOOL` and is specified by the following Cheetah entity:

```cpp
entity bool_value : named_value {
    attribute bool value;
};
```

**value:** A field of type `BOOL` that represents the value.

### 2.2.27 `cht::core::float_value`

This is a subtype of the `cht::core::named_value` that represents a resource value of type `float` and is specified by the following Cheetah entity:

```cpp
entity float_value : named_value {
    attribute float value;
};
```

**value:** A field of type `float` that represents the value.

### 2.2.28 `cht::core::string_value`

This is a subtype of the `cht::core::named_value` that represents a resource value of type `string` and is specified by the following Cheetah entity:

```cpp
entity string_value : named_value {
    attribute string value;
};
```

**value:** A `string` that represents the value.

### 2.2.29 `cht::core::long_value`

This is a subtype of the `cht::core::named_value` that represents a resource value of type `int` and is specified by the following Cheetah entity:

```cpp
entity long_value : named_value {
    attribute int value;
};
```
value: A field of type int that represents the value.

2.2.30  cht::core::longlong_value

This is a subtype of the cht::core::named_value that represents a resource value of type longint and is specified by the following Cheetah entity:

entity longlong_value : named_value {
    attribute longint value;
};

value: A field of type longint that represents the value.

2.2.31  core::lifecycle::state

This specifies the four runtime states for a protocol server process: initializing, running, suspended, and terminating, as specified in section 3.8.1. The FIDL specification for the enumeration is as follows:

enum state {
    initializing, running, suspended, terminating
};
3 Protocol Details

3.1 Common Middleware Details

3.1.1 Abstract Data Model

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

The middleware data model represents defined data structures and values that are implementation-specific, as well as the remote method invocation input values, return values, or any system or user exceptions resulting from the invocation. FSIDL specifies the middleware data model instead of mapping to a specific programming language. Section 3.1.4 contains information about FSIDL specifications, and specifies how to map the FSIDL notation into the corresponding middleware types.

Middleware type

A middleware type is identified by a case-sensitive name, and specifies the structure of data. FSIDL notation refers to the name of the type. The middleware data model supports the BYTE, INT32, and INT64 types specified in [MS-DTYP]. In addition, the middleware data model specifies the following types:

- **Float**: Represents a 32-bit single-precision floating-point value, as specified in section 2.2.1.
- **Void**: A type that specifies no value, as specified in section 2.2.2.
- **LengthPrefixedByteSequence**: A sequence of BYTE values, as specified in section 2.2.3.
- **String**: A sequence of BYTE values that represents a String, as specified in section 2.2.4.
- **LengthPrefixedInt32Sequence**: A sequence of INT32 values, as specified in section 2.2.5.
- **LengthPrefixedInt64Sequence**: A sequence of INT64 values, as specified in section 2.2.6.
- **LengthPrefixedStringSequence**: A sequence of String values, as specified in section 2.2.7.
- **LengthPrefixedFloatSequence**: A sequence of Float values, as specified in section 2.2.8.
- **CheetahValue**: A Cheetah entity, as specified in section 2.2.11.
- **AbstractObjectReference**: An abstract object reference, as specified in section 2.2.14.
- **SystemException**: A system exception, as specified in section 2.2.12.
- **UserException**: A user exception, as specified in section 2.2.13.

Middleware data value

A middleware data value is an instance of a middleware type.

Server interface
This encapsulates a set of method declarations. Multiple versions of the same protocol server interface can be instantiated in the same protocol server.

**Server interface version**

This represents the version of a specific server interface.

**Server object**

An instance of a server interface that is associated with the specified version and server object identifier.

**Server object identifier**

A number that is unique for each server object within the network host where the server object is instantiated.

**Client proxy**

This sends information to the protocol server used to call the remote methods on the server object. The client proxy uses an abstract object reference to refer to the server object.

**Abstract object reference**

This represents a server object that is sent between a protocol client and a protocol server. It contains sufficient information to construct a client proxy that calls remote methods on the server object. More specifically, an abstract object reference for a specified server object is identified by the following:

- **Host name**: The host name where the server object executes.
- **Port number**: The port number associated with the server object.
- **Server interface**: The server interface of the server object.
- **Server interface version**: The server interface version of the server object.
- **Server object identifier**: The identifier of the server object.

The abstract object reference is represented by the *AbstractObjectReference* record specified in section 2.2.14.

**Remote method**

Represents a method that is called remotely and that is declared in a server interface. The specification of a remote method contains the following:

- **Name**: The name of the remote method. A remote method is uniquely identified within a protocol server interface by the remote method name.
- **Arguments**: An ordered collection of arguments, where each argument has a name and a Middleware type.
- **Exceptions**: A collection of user exceptions that the remote method throws. Each user exception has a name.
- **Return type**: The type of the value returned by the remote method.

The remote method request and response consist of the following:
- **Name:** The name of the remote method.

- **Input values:** An ordered collection of values, one for each remote method Argument field. Each value is a Middleware data value. The input values are represented by the CallArguments record specified in section 2.2.15.

- **Return value:** The value that contains the result of the remote method, in an OutputValue record, as specified in section 2.2.9. The OutputValue record contains one of three possible results from a remote method:
  
  A value with the same type as the **Return Type** field returned by the remote method. The value returned from a remote method is represented by the CallResult record specified in section 2.2.10.

  A system exception that represents a processing error associated with a remote method. A system exception contains a human-readable message that specifies the error. If a human-readable message is not possible to infer from the underlying error, an empty message or the message "N/A" is used. A system exception is represented by the SystemException record specified in section 2.2.12.

  A user exception that represents an implementation-specific processing error associated with a remote method. A user exception is represented by the UserException record specified in section 2.2.13.

### 3.1.2 Timers

None.

### 3.1.3 Initialization

None.

### 3.1.4 Message Processing Events and Sequencing Rules

The following specifies the format of FSIDL specifications, how to map FSIDL specifications to remote method specifications, and how data types in FSIDL specifications are mapped to the corresponding middleware types.

#### 3.1.4.1 FSIDL Specifications

Applications use this protocol to specify server interfaces with FSIDL specifications. An implementation of this protocol does not require the use of FSIDL, whose notation specifies the interfaces between protocol clients and protocol servers.

FSIDL specifications resemble MIDL specifications, as described in [MSDN-MIDL], and provide a subset of the OMG IDL language, as described in [CORBA]. The following string pattern, using Augmented Backus-Naur Form (ABNF) syntax specified in [RFC5234], specifies the FSIDL specifications:

```plaintext
FSIDLSpecification = (Cheetah / (Cheetah FSIDL) / FSIDL) LWSP ";
Cheetah = "module" LWSP "cht" LWSP "{" LWSP 1*CheetahModuleDecl LWSP "}" LWSP ""
CheetahModuleDecl = "module" LWSP CheetahModule LWSP "{" LWSP CheetahTypedef LWSP 1*CheetahEntityTypeDef LWSP "}" LWSP ""
CheetahTypedef = "typedef" LWSP "sequence" LWSP "(" LWSP "octet" LWSP ")" LWSP "cheetah" LWSP ";
CheetahEntityTypeDef = "typedef" LWSP "cheetah" LWSP CheetahEntity LWSP ";
CheetahModule = Name
```

[MS-FSMW] — v20120630
Middleware Protocol Specification

Copyright © 2012 Microsoft Corporation.

Release: July 16, 2012
The following uses the notation "FSIDL RuleName" when referring to a rule RuleName that is specified by the previous ABNF grammar.

3.1.4.2 Mapping FSIDL MethodDecl to Remote Method Specifications

The FSIDL MethodDecl specifies the name, arguments, exceptions, and return type of the remote method. The FSIDL ReturnType statement specifies the return type of the remote method, the FSIDL MethodName statement the name of the remote method, the FSIDL arguments specifies the arguments of the remote method, and the FSIDL RaisesExceptions specifies the exceptions of the remote method.
3.1.4.3  Mapping Remote Method Request

The name of the remote method is specified by the FSIDL ArgumentName. The middleware type for each argument of the remote method Arguments is specified by the FSIDL ArgumentType. The input values of the remote method MUST be serialized as a CallArguments record, as specified in section 2.2.15. Each Argument field in the CallArguments record MUST contain the value that corresponds to the type specified by the FSIDL ArgumentType. The input values of the remote method MUST be serialized in same order as the FSIDL Arguments.

3.1.4.4  Mapping Remote Method Reply

This consists of a CallResult record which contains a value, a system exception, or a user exception. The FSIDL ReturnType specifies the return type for the remote method Return Value. The remote method Return Value MUST be serialized as a CallResult record, as specified in section 2.2.10. The ResultContents field MUST contain the serialized Return Value.

3.1.4.5  Mapping FSIDL AtomicType

The following table specifies the mapping between types specified by the FSIDL AtomicType and the corresponding protocol types.

For FSIDL BooleanType, the BYTE value 0 represents false, and the BYTE value 1 represents true.

For FSIDL StringType, an empty string MUST be serialized as a String record, as specified in section 2.2.4, with the Length field set to 0.

<table>
<thead>
<tr>
<th>FSIDL atomic type</th>
<th>Protocol type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>BYTE, as specified in [MS-DTYP].</td>
</tr>
<tr>
<td>octet</td>
<td>BYTE, as specified in [MS-DTYP].</td>
</tr>
<tr>
<td>boolean</td>
<td>BYTE, as specified in [MS-DTYP].   The BYTE value 0 represents false, and the BYTE value 1 represents true.</td>
</tr>
<tr>
<td>string</td>
<td>String, as specified in section 2.2.4.</td>
</tr>
<tr>
<td>long</td>
<td>INT32, as specified in [MS-DTYP].</td>
</tr>
<tr>
<td>long long</td>
<td>INT64, as specified in [MS-DTYP].</td>
</tr>
<tr>
<td>float</td>
<td>Float, as specified in 2.2.1.</td>
</tr>
<tr>
<td>void</td>
<td>Void, as specified in section 2.2.2.</td>
</tr>
</tbody>
</table>

3.1.4.6  Mapping FSIDL SequenceType

The following table specifies the mapping between types specified by the FSIDL SequenceType and the corresponding protocol types.

<table>
<thead>
<tr>
<th>FSIDL Sequence Type</th>
<th>Protocol Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;char&gt;</td>
<td>LengthPrefixedByteSequence, as specified in section 2.2.3</td>
</tr>
</tbody>
</table>
### 3.1.4.7 Mapping FSIDL EnumName

This specifies an integer whose values are associated with unique names specified by a FSIDL Enum. A value MUST be serialized as an INT32 value that begins with the value 0, increasing by 1 in the order specified by the FSIDL EnumList.

### 3.1.4.8 Mapping FSIDL CheetahEntityName

The FSIDL CheetahEntityName specifies a Cheetah entity and MUST be serialized as a CheetahValue record, as specified in section 2.2.11.

### 3.1.4.9 Mapping FSIDL InterfaceName

The FSIDL InterfaceName specifies a server interface as a remote method argument, or return type. It is serialized as an AbstractObjectReference record, as specified in section 2.2.14, where the InterfaceType field contains the FSIDL InterfaceName, and the InterfaceVersion field contains the FSIDL InterfaceValue.

### 3.1.4.10 Mapping FSIDL ExceptionName

This specifies a user exception for a server interface. The FSIDL ExceptionName MUST be serialized into the UserException record as specified in section 2.2.13. The FSIDL ExceptionAttributes MUST map to the Attribute fields of the UserException record in the same order they occur in the FSIDL ExceptionDecl.

### 3.1.5 Timer Events

None.

### 3.1.6 Other Local Events

None.

### 3.2 Middleware Server Details

#### 3.2.1 Abstract Data Model

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

This table associates server object URIs with the corresponding server objects.

Channel table

This contains an association between a channel URI and a server object table. A channel URI is an entry point from which a protocol server receives connection requests from a protocol client. A protocol server hosts one or more channel URIs. The channel URI is specified with a URI scheme, a network host name, and a port number. The channel URI scheme field MUST contain a value of either "http" or "https".

3.2.2 Timers

None.

3.2.3 Initialization

The protocol server channel table MUST be initialized and associated with a server object table by the higher layer. Higher-level protocols MUST specify the following information about the transport:

- If an authentication mechanism such as Kerberos or NTLM is required, the required credentials MUST be specified. If HTTPS is required, a certificate MUST be specified.
- The maximum size of HTTP response and requests, specified in octets.

For each channel URI, the protocol server listens at the network host name and port number specified by the channel URI, and uses the transport specified by the channel URI scheme.

The host network interface, port number, channel URI scheme, credentials, and certificate are specified with command line options on the protocol server, or they can be specified in a configuration file.

When the higher layer registers a server object, a server object URI represented by the ServerObjectURI in section 2.2.16 MUST be constructed. The higher layer MUST provide values for the InterfaceType and the InterfaceVersion fields. The protocol server MUST generate a server object identifier for the ServerObjectId field. The protocol servers generate the server object identifier by concatenating a random number, the number of registered server objects with this server interface in the protocol server, and the thread identifier for the protocol server.

After the protocol server constructs the server object URI, it performs the following:

- If the server object URI does not exist in the server object table, the protocol server adds the server object URI and server object to the server object table.
- If the server object URI is already present in the server object table, the protocol server notifies the higher layer about the error.

When the higher layer unregisters a server object using the server object URI, a protocol server MUST do the following:

- Remove the entry with the server object URI from the server object table.
- The requests that are being processed, if any, will finish processing. Because request processing does not always finish in a timely manner, completing such requests is implementation-specific.
If the server object URI is not found in the server object table, the higher layer is notified about the error.

3.2.4 Message Processing Events and Sequencing Rules

There are no sequencing rules in this protocol. This section specifies the methods described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Method Call</td>
<td>Specifies how to receive, process, and respond to a remote method request.</td>
</tr>
<tr>
<td>__ping</td>
<td>Determines whether a server object is responding.</td>
</tr>
</tbody>
</table>

### 3.2.4.1 Remote Method Invocation

A protocol server MUST perform the following actions when it receives a message from the protocol client:

1. Receive the HTTP request.
2. Look up the server object.
3. De-serialize the request message content.
4. Validate and dispatch the call.
5. Serialize the response-message content.
6. Send the response.

**Receive HTTP request**

A protocol server MUST determine the channel URI associated with the received message in an implementation-specific way. The protocol client request MUST be mapped to an HTTP request message. The protocol server MUST accept HTTP request messages that are sent using HTTP/1.1. If the HTTP method is not POST or if the Content-Type is not "application/octet-stream", a protocol server sends a transport fault to the protocol client. The protocol server MUST format the transport fault as follows:

- HTTP status code of the response is set to 200.
- The body of the response contains an `OutputValue`, with a b field whose value is 50 and a `MessageContent` field of type `SystemException`.

If an HTTP request is not received before an implementation-specific time-out has elapsed, the protocol server cancels the request.

**Look up the server object**

The message contains an HTTP Request-URI that specifies the protocol server method to which to route the message, as specified in section 2.2.17. The server object URI is a prefix of the protocol server method URI. A protocol server looks up the server object URI in the server object table. If the server object URI is found in the **Server Object Table**, then the corresponding server object in the table is used to dispatch the call.
If the server object URI is not found, then the protocol server sends an HTTP status code 404 to the protocol client.

**De-serialize the request message content**

The message content is de-serialized from the CallArguments record, as specified in section 2.2.15. The remote method name MUST be parsed from the ServerMethodURI, as specified in section 2.2.17.

If the message content does not conform to the expected message format or the association from the serialization stream to the middleware data model resulted in an error, then this is a malformed message. If the protocol server receives a malformed message, it constructs a SystemException, and sends it back to the protocol client in the MessageContent field of an OutputValue. The Description field MUST either convey the nature of the structural error or be a zero-length string.

**Validate and dispatch the call**

The protocol server locates the remote method in the server object using the name of the remote method. If the remote method is not found, the protocol server constructs a SystemException and sends it back to the protocol client in the MessageContent field of an OutputValue. The Description field MUST either convey the nature of the error or be a zero-length string.

If the remote method can be located in the server object, a protocol server MUST call the remote method with the remote method Input Values resulting from de-serializing the CallArguments record.

**Serialize the response message content**

The completion of a remote method yields a return value, a user exception or a system exception. A protocol server constructs an OutputValue record that contains either a CallResult record as specified in section 2.2.10 based on the return value, a UserException record as specified in 2.2.13 based on the user exception, or a SystemException based on the system exception. The values are serialized into the MessageContent field of the OutputValue, and the protocol server sends the OutputValue to the protocol client.

If there is any error during serialization, then a protocol server constructs a SystemException record, and sends it back to the protocol client. The SystemException is serialized into the MessageContent field of the OutputValue. The Description field MUST either convey the nature of the error or be a zero-length string.

A value that represents a server object or a client proxy MUST be sent as an AbstractObjectReference record, as specified in section 2.2.14.

The implementation MUST provide a valid abstract object reference so that the server object can construct the AbstractObjectReference record.

**Send the response**

The protocol server maps the remote method response to an HTTP response, which MUST contain the following HTTP header fields:

- The Content-Type entity header of the response contains a value of "application/octet-stream".
- The Content-Length entity header of the response contains the length of the response body, specified in decimal number of octets.
- The HTTP status code of the response is set to 200.
The HTTP Reason-Phrase of the response contains a value of "OK".

The response body of the HTTP response message MUST contain an OutputValue record. SystemException records are specified in section 2.2.12, and OutputValue records are specified in section 2.2.9.

### 3.2.4.2 __ping

The __ping method MUST be implemented by all server objects, and is used by protocol clients to determine whether specific server objects respond to requests. The method signature is specified by the following FSIDL:

```plaintext
void __ping(void);
```

**Input values**

**Void:** No input values.

**Return value**

**Void:** No return value.

**Exception:** No exceptions are raised other than system exceptions.

Client proxies that call the __ping method assume the server object is responding if the method does not return a SystemException record, as specified in section 2.2.12. The protocol server that hosts a server object can be subject to a transient network failure, or process slowly because of excessive load. Therefore, the outcome of a single __ping message is not always sufficient to establish whether a server object is responding.

### 3.2.5 Timer Events

None.

### 3.2.6 Other Local Events

None.

### 3.3 Middleware Client Details

#### 3.3.1 Abstract Data Model

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

**Client Proxy table**

The client proxy table associates a client proxy with a specific server object, transport, serialization format, and the network address of the server object. It contains an entry for each client proxy instance. Each entry contains the following items:

- **Client Proxy:** The instance of the client proxy to locate in the table.
- **Abstract Object Reference**: An abstract object reference for the server object.

### 3.3.2 Timers

None.

### 3.3.3 Initialization

When a higher-level implementation requests a client proxy, it MUST provide an abstract object reference.

The higher layer requests a `cht::nameservermsg::aor` Cheetah entity from a name server by calling the `resolve` method, and converting the resulting `cht::nameservermsg::aor` Cheetah entity to an `AbstractObjectReference` record. Otherwise, the higher layer provides the information required for an abstract object reference in a way that is implementation-specific. Some protocol servers specify the abstract object reference for a name server with command line arguments to the Middleware Protocol clients and protocol servers, or in a configuration file.

If the abstract object reference is not well-formed, the client proxy MUST raise a system exception for the implementation, and include a description of the structural error.

The protocol client creates a new client proxy, and adds the client proxy and the abstract object reference to the client proxy table. The protocol client can call the `__ping` method to validate that the server object associated with the client proxy is responding, before it adds the client proxy to the client proxy table. However, calling the `__ping` method is not required.

The abstract object reference is specified in section 2.2.14, the `cht::nameservermsg::aor` Cheetah entity and the mapping table are specified in section 2.2.18, the `resolve` method is specified in section 3.4.4.1, and the `__ping` method is specified in section 3.2.4.2.

### 3.3.4 Message Processing Events and Sequencing Rules

#### 3.3.4.1 Remote Method Invocation

When a higher layer calls a remote method using a client proxy that sends the name and call arguments of the remote method, the protocol client MUST serialize the request, send the request to the protocol server, read the response message, de-serialize the response message, and send the de-serialized values to the caller.

**Serialize the request**

A protocol client looks up the abstract object reference in the client proxy table. If the client proxy is not in the table, then the higher layer MUST be reported using an implementation-specific procedure.

The abstract object reference, represented by the `AbstractObjectReference` record, is used to create a `ServerObjectURI`, as specified in section 2.2.16. The `ServerObjectURI` MUST use the `InterfaceType`, `InterfaceVersion`, and `ServerObjectId` fields from the `AbstractObjectReference` record. The protocol client uses the resulting `ServerObjectURI` and the remote method name to create a `ServerMethodURI`, as specified in section 2.2.17.

A protocol client creates a `CallArguments` record, as specified in section 2.2.15, based on the remote method input values received from the higher layer. If the type of the values contained in the remote method input values does not match the type of the remote method `Arguments`, the higher-layer MUST be reported using an implementation-specific procedure.
Send the request to the protocol server

The protocol client MUST construct an HTTP Request-URI using the ServerMethodURI and the AbstractObjectReference constructed in the previous paragraph, where the host name and port fields of the Request-URI are the same as the AbstractObjectReference.

The protocol client maps the remote method request to an HTTP request, which MUST contain the following HTTP header values:

- An implementation MUST use HTTP/1.1.
- The HTTP Method MUST be a POST.
- The Request-URI of the HTTP request message MUST be the ServerMethodURI of the remote method, as specified in section 2.2.17.
- The Content-Length entity header MUST contain the length of the request body in decimal number of octets.
- The Content-Type entity header MUST be "application/octet-stream".
- The body of the HTTP request MUST be a CallArguments record, as specified in section 2.2.15.

Read the response from the connection

If the protocol client does not receive a response within a specified amount of time after sending a request, it MUST cancel the request, raise a system exception with a description of the time-out, and send the message to the higher layer. The timeout MUST be defined by the higher layer.

If the status code of the HTTP response is one of the successful codes as specified in [RFC2616] section 10.2, the protocol client MUST de-serialize the response message. If the status code is a protocol client-error code as specified in [RFC2616] section 10.4, a protocol server-error code as specified in [RFC2616] section 10.5 or an unknown error code, the protocol client MUST stop processing the response, and instead, use an implementation-specific procedure to notify the higher layer of the error.

De-serialize the response message

The response message MUST contain an OutputValue record, as specified in section 2.2.9. The protocol client de-serializes the OutputValue record to obtain the remote method return value, system exception, or user exception. If the message content does not match the abstract data model, then the protocol client stops processing the message and notifies the higher layer about the error.

Return the de-serialized values to the caller

The protocol client MUST return the de-serialized return value to the calling application, or it MUST raise a system exception or user exception. If the type of the de-serialized values does not match the type of the remote method return type, then the higher-layer is also notified of the error.

3.3.5 Timer Events

None.

3.3.6 Other Local Events

None.
3.4 Name Server Server Details

3.4.1 Abstract Data Model

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

Before a protocol client can communicate with a protocol server, the protocol client requires the abstract object reference for the protocol server. The name server provides a mapping from a logical name to an abstract object reference. A protocol client can then locate a specific server object by querying the name server protocol server for a logical name that the protocol client and protocol server have agreed to use for this server object.

Logical Name

The Logical Name is a triple that uniquely represents an abstract object reference in the name server, and consists of the following three entries:

- **Name**: A symbolic name that represents the abstract object reference.
- **Server interface**: The server interface name of the abstract object reference.
- **Server Interface Version**: The server interface version of the abstract object reference.

Name Server Abstract Object Reference table

The name server AOR table associates logical name entries to AORs. The name server stores AORs so that the protocol clients can find an AOR based on the logical name of that AOR. For each unique logical name, there MUST be only one AOR.

3.4.2 Timers

None.

3.4.3 Initialization

A server object URI that represents the name server server object MUST be created and a mapping from the server object URI to the name server server object MUST be inserted in the server object table of the Middleware protocol server that hosts the name server protocol server.

The server object URI is represented concretely by the ServerObjectURI record specified in section 2.2.16, where the InterfaceType field MUST be “nameservice::nameserver”, the ServerObjectId field MUST be 0, and the InterfaceVersion field MUST be 1.0.

3.4.4 Message Processing Events and Sequencing Rules

There are no sequencing rules in this protocol. This interface includes the methods described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolve</td>
<td>Looks up the information needed to construct an abstract object reference for a specified server object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>bind</td>
<td>Associates an abstract object reference with a logical name in the name server.</td>
</tr>
<tr>
<td>unbind</td>
<td>Removes the association between an abstract object reference and the associated logical name in the name server.</td>
</tr>
<tr>
<td>list_any</td>
<td>Requests a collection of AORs that match all of the specified input values.</td>
</tr>
<tr>
<td>list_host</td>
<td>Requests a collection of AORs that match a specific host name.</td>
</tr>
<tr>
<td>list_name</td>
<td>Requests a collection of AORs that match a specific logical name prefix.</td>
</tr>
</tbody>
</table>

### 3.4.4.1 resolve

The `resolve` method looks up a specified server object and returns the information needed to construct an abstract object reference. The method signature is specified by the following FSIDL:

```c
cht::nameservermsg::aor resolve(in string name,
    in string interface_type,
    in string version)
raises (resolve_exception);
```

**Input values**

- **name**: A string that represents the Name part of the Logical Name in the name server that is associated with the abstract object reference.
- **interface_type**: A string that represents the server interface of the server object.
- **version**: A string that represents the server interface version of the server object.

**Return value**

- `cht::nameservermsg::aor`: A Cheetah entity, as specified in section 2.2.18, that represents the abstract object reference for the server object. This Cheetah entity can be converted to an AbstractObjectReference record.

**Exceptions**

- `resolve_exception`: No association exists between the requested server object and logical name.

The `resolve` method MUST return an abstract object reference encoded as a `cht::nameservermsg::aor` Cheetah entity. The method looks for the abstract object reference in the name server AOR table by using the input values for the logical name triple. If no abstract object reference is found, the method MUST raise a `nameservice::nameserver::resolve_exception` user exception, as specified in section 2.2.21.

If an abstract object reference is found, a `cht::nameservermsg::aor` Cheetah entity is constructed from the AOR as specified in section 2.2.18 and sent to the protocol client.

### 3.4.4.2 bind

The `bind` method associates an abstract object reference with a logical name in the name server. The method signature is specified by the following FSIDL:
void bind(in cht::nameservermsg::aor the_aor);

**Input values**

the_aor: A Cheetah entity, as specified in section 2.2.18, that represents the abstract object reference for the server object.

**Return value**

void: No value returned.

**Exceptions:** No exceptions are raised other than system exceptions.

The logical name triple MUST be constructed based on the attributes in. An abstract object reference MUST be constructed from the the_aor based on the mapping table in section 2.2.18.

The name server AOR table MUST be updated to specify the mapping from the logical name to the abstract object reference.

### 3.4.4.3 unbind

The unbind method removes the association between an abstract object reference and the associated logical name in the name server. The method signature is specified by the following FSIDL:

```plaintext
void unbind(in string name,
            in string interface_type,
            in string version)
  raises (not_bound_exception);
```

**Input values**

name: A string that represents the name field of the logical name triple that is associated with the AOR in the name server.

interface_type: A string that represents the server interface of the server object.

version: A string that represents the server interface version of the server object.

**Return value**

void: No value returned.

**Exceptions**

not_bound_exception: The logical name is not associated with an abstract object reference.

The abstract object reference MUST be removed from the name server AOR table. The logical name MUST be used to locate and remove the abstract object reference. If no abstract object reference is found, a not_bound_exception user exception, as specified in section 2.2.20, must be raised.

### 3.4.4.4 list_any

The list_any method requests a collection of AORs that matches all of the specified input values. The method signature is specified by the following FSIDL:
cht::nameservermsg::aor_list list_any(in string name_prefix,
   in string interface_type,
   in string version,
   in string host);

Input values

name_prefix: A string that represents a prefix of the name field in the logical name triple in the name server.

interface_type: A string that represents the server interface of the server object.

version: A string that represents the server interface version of the server object.

host: A string that represents the host name of the server object.

Return value

cht::nameservermsg::aor_list: A collection of Cheetah entities, as specified in section 2.2.19, that represents the AORs for the requested server objects.

Exceptions: No exceptions are raised other than system exceptions.

The protocol server MUST traverse the name server AOR table and return a cht::nameservermsg::aor_list Cheetah entity. An entry in the table MUST fulfill the following criteria to be included in the resulting return value:

- If the name_prefix field is not the empty string, the name field in the logical name triple MUST begin with the value of name_prefix.
- If the interface_type field is not the empty string, the server interface of the logical name MUST be equal to the value of interface_type.
- If the version field is not the empty string, the server interface version of the logical name MUST be equal to the value of version.
- If the host field is not the empty string, the host name of the abstract object reference MUST be equal to the value of host.

If one or more of the input values are the empty strings, fields with an empty string matches all entries for that field in the name server AOR table. Thus, if all input values to list_any are the empty strings, all entries of the name server AOR table MUST be included in the resulting cht::nameservermsg::aor_list Cheetah entity.

3.4.4.5 list_host

The list_host method requests a collection of AORs matching a specific host name. The method signature is specified by the following FSIDL:

cht::nameservermsg::aor_list list_host(in string host,
   in string interface_type);

Input values

host: A string that represents the host name of the server object.
**interface_type:** A string that represents the server interface of the server object.

**Return value**

cht::nameservermsg::aor_list: A collection of Cheetah entities, as specified in section 2.2.19, that represent the AORs for the requested server objects.

**Exceptions:** No exceptions are raised other than system exceptions.

If both the **host** and **interface_type** input values are empty strings, a cht::nameservermsg::aor_list Cheetah entity with an empty collection MUST be returned.

The protocol server MUST traverse the name server AOR table and return the cht::nameservermsg::aor_list Cheetah entity.

An entry in the name server AOR table MUST fulfill the following criteria to be included in the resulting return value:

- The host name of the abstract object reference MUST be the same as the value of the **host** field.
- If the **interface_type** field is not the empty string, the server interface of the logical name MUST be equal to the value of **interface_type**.

### 3.4.4.6 list_name

The list_name method requests a collection of AORs matching a specific logical name prefix. The method signature is specified by the following FSIDL:

```
cht::nameservermsg::aor_list list_name(in string name_prefix,
    in string interface_type);
```

**Input values**

**name_prefix:** A string that represents a prefix of the name field in the logical name triple in the name server.

**interface_type:** A string that represents the server interface of the server object.

**Return value**

cht::nameservermsg::aor_list: A collection of Cheetah entities, as specified in section 2.2.19, that represent the AORs for the requested server objects.

**Exceptions:** No exceptions are raised other than system exceptions.

The protocol server MUST traverse the name server AOR table and return the cht::nameservermsg::aor_list Cheetah entity.

An entry in the table MUST fulfill the following criteria to be included in the resulting return value:

- If the **name_prefix** field is not the empty string, the name field in the logical name triple MUST begin with the value of **name_prefix**.
- If the **interface_type** field is not the empty string, the server interface field in the logical name triple MUST be equal to the value of **interface_type**.
If both input values for the `list_name` method are the empty string, all entries of the name server AOR table MUST be included in the resulting Cheetah entity.

3.4.5 Timer Events

None.

3.4.6 Other Local Events

None.

3.5 Name Server Client Details

3.5.1 Abstract Data Model

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

3.5.2 Timers

None.

3.5.3 Initialization

The name server protocol client is a client proxy that calls remote methods on a name server protocol server. A protocol client creates a client proxy, based on an abstract object reference to the name server object, to call remote methods on the name server protocol server as specified in section 3.3.3.

An `AbstractObjectReference` record represents the abstract object reference for the name server object, as specified in section 2.2.14. The implementation MUST specify that the `InterfaceType` field of the `AbstractObjectReference` record is set to "nameservice::nameserver", the `ServerObjectId` field is set to 0, and the `InterfaceVersion` is set to 1.0.

3.5.4 Message Processing Events and Sequencing Rules

Before calling the `bind` remote method (section 3.4.4.2), an implementation MUST first call the `resolve` remote method (section 3.4.4.1), with the same logical name that the `bind` remote method will use.

If the `resolve` method returns a `cht::nameservermsg::aor` (section 2.2.18), the protocol server MUST create a client proxy as specified in section 3.3.3, and then call the `__ping` method (section 3.2.4.2) using the client proxy.

If the `__ping` method does not raise a system exception or a user exception, then another server object has previously registered with the same logical name, and therefore the protocol server MUST NOT call the `bind` method.

3.5.5 Timer Events

None.
3.5.6 Other Local Events

None.

3.6 core::fds_component Server Details

This interface is implemented by all protocol servers and is used by protocol clients to request status and resource usage information from the protocol servers.

3.6.1 Abstract Data Model

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

Debug State Table

This represents the correspondence between a debug module and a debug level:

- **Debug Module:** A name that identifies an implementation-specific part of the protocol server.
- **Debug Level:** A number that specifies an implementation-specific debug level.

The protocol server uses the debug level to decide whether to output debug messages. Some protocol servers configure debugging output through implementation-specific command line arguments that override the debug module and debug level settings.

Resource Scope Table

A Resource Scope is an implementation-specific scope name defined over a well-defined set of program statements, typically the program statements defining a function or method. The resource scope table is a mapping between a scope name and an entry consisting of **Current Invocations**, **Total Invocations**, **Minimum Duration**, **Average Duration**, and **Maximum Duration** fields.

- **Scope Name:** A name that uniquely identifies the resource scope.
- **Current Invocations:** The number of threads currently executing the resource scope.
- **Total Invocations:** The number of times the resource scope has been executed.
- **Minimum Duration:** The minimum amount of time spent executing the resource scope in milliseconds.
- **Average Duration:** The average amount of time spent executing the resource scope in milliseconds.
- **Maximum Duration:** The maximum amount of time spent executing the resource scope in milliseconds.

Resource Allocation Table

A Resource Allocation is an implementation-specific counter used to count named resources such as files or memory units. The Resource Allocation table is a mapping between an **Allocation Name** field and an entry consisting of **Current Allocations** and **Total Allocations** fields.
• **Allocation Name:** A name that uniquely identifies the resource allocation.

• **Current Allocations:** Contains the current count for the specified Allocation Name field.

• **Total Allocations:** Contains the total count for the specified Allocation Name field.

**Resource Value Table**

A Resource Value is an implementation-specific value that is associated with a unique name. More specifically, the resource value table specifies a mapping between a unique Value Name field and the implementation-specific resource value.

### 3.6.2 Timers

None.

### 3.6.3 Initialization

The `core::fds_component` server object MUST be initialized by a higher-level implementation that uses the protocol server. The protocol server MUST call the **bind** method, as specified in section 3.4.4.2.

The input values to the **bind** method is a `cht::nameservermsg::aor` Cheetah entity, as specified in section 2.2.18.

- **name:** A string value supplied by the higher-level application.
- **object_id:** A value that is implementation-specific, that is, determined by the higher-level application.
- **host:** A string that contains the host name of the server object on the protocol server. The value is implementation-specific and determined by the higher-level application.
- **port:** The port number used by the protocol server. It is implementation-specific and determined by the higher level application.
- **interface_type:** A string value that MUST be "core::fds_component ".
- **interface_version:** A string value that MUST be "5.1".

The debug level MUST be initialized by the higher-level implementation.

### 3.6.4 Message Processing Events and Sequencing Rules

There are no sequencing rules in this protocol. This interface includes the methods described in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_hostname</td>
<td>Return the host name used for the channel URI of the protocol server.</td>
</tr>
<tr>
<td>get_resource_report</td>
<td>Returns the content of the resource allocation table.</td>
</tr>
<tr>
<td>uptime</td>
<td>Return the number of seconds that elapsed after the protocol server process started.</td>
</tr>
<tr>
<td>get_version</td>
<td>Return a string that represents the server version for a protocol server.</td>
</tr>
</tbody>
</table>

---

[MS-FSMW] — v20120630
Middleware Protocol Specification

Copyright © 2012 Microsoft Corporation.

Release: July 16, 2012
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_model_version</td>
<td>Return a version string for the protocol server interface versions in the protocol server.</td>
</tr>
<tr>
<td>get_fds_version</td>
<td>Return a version string that identifies the product version.</td>
</tr>
<tr>
<td>get_middleware_port</td>
<td>Return the port number used for the channel URI of the protocol server.</td>
</tr>
<tr>
<td>set_tracelevel</td>
<td>Sets the debug log level for an implementation-specific module in the protocol server.</td>
</tr>
</tbody>
</table>

#### 3.6.4.1 get_hostname

The **get_hostname** method returns the host name for the channel URI of the protocol server. The method signature is specified by the following FSIDL:

```c
string get_hostname(void);
```

**Input values**

**Void:** No input values.

**Return value**

**string:** A string that represents the host name for the channel URI of this server object.

**Exceptions:** No exceptions are thrown other than system exceptions.

Return the host name of the protocol server. The host name MUST be specified by the higher-level implementation as part of the configuration.

#### 3.6.4.2 get_resource_report

The **get_resource_report** method retrieves the contents of the resource allocation table. The method signature is specified by the following FSIDL:

```c
cht::core::resource_report get_resource_report(void);
```

**Input values**

**Void:** No input values.

**Return value**

**cht::core::resource_report:** A Cheetah entity, as specified in section 2.2.22, that represents a resource report from a protocol server.

**Exceptions:** No exceptions are raised other than system exceptions.

Returns a **cht::core::resource_report** Cheetah entity. The **cht::core::resource_report** is a Cheetah entity that contains three collections with the type **cht::core::alloc** specified in section 2.2.23, **cht::core::scope** specified section 2.2.24, and **cht::core::named_value** specified in section 2.2.25.
The method constructs a `cht::core::alloc` Cheetah entity for each entry in the resource allocation table, where the `name` attribute maps to the Allocation Name field, the `current` attribute maps to the Current Allocations field, and the `total` attribute maps to the Total Allocations field.

The method constructs a `cht::core::named_value` Cheetah entity for each entry in the resource value table, where the `name` attribute maps to the Value Name field, and the `value` attribute maps to the corresponding value.

The method constructs a `cht::core::scope` Cheetah entity for each entry in the resource scope table, where the `name` attribute maps to scope name, the `current` attribute maps to the Current Invocations field, the `total` attribute maps to the Total Invocations field, the `min_time` attribute maps to the Minimum Duration field, the `max_time` field maps to the Maximum Duration field, and the `avg_time` field maps to the Average Duration field.

The method creates a `cht::core::resource_report` Cheetah entity with the `cht::core::alloc`, `cht::core::named_value` and `cht::core::scope` collections. The `when` attribute of the `cht::core::resource_report` contains the number of seconds since January 1, 1970.

3.6.4.3 uptime

The `uptime` method returns the number of seconds elapsed after the protocol server process was started. The method signature is specified by the following FSIDL:

```cpp
long uptime(void);
```

Input values

**Void:** No input values.

Return value

**long:** A long that represents the number of seconds elapsed after the protocol server process was started.

Exceptions: No exceptions are raised other than system exceptions.

Return the number of seconds elapsed after the protocol server process was started.

3.6.4.4 get_version

The `get_version` method retrieves the server version for a protocol server. The method signature is specified by the following FSIDL:

```cpp
string get_version(void);
```

Input values

**Void:** No input values.

Return value

**string:** A string that represents an implementation-specific version for a protocol server. An empty string or the string "N/A" is used when a meaningful value cannot be determined by the implementation.
Exceptions: No exceptions are raised other than system exceptions.

Return an implementation-specific string that represents the version number for the protocol server that contains the server object for the `core::fds_component` server interface.

### 3.6.4.5 get_model_version

The `get_model_version` method retrieves the version string for the protocol server. The method signature is specified by the following FSIDL:

```fsidl
string get_model_version(void);
```

**Input values**

**Void:** No input values.

**Return value**

**string:** This represents an implementation-specific version for all interfaces implemented by this protocol server. An empty string or the string "N/A" is used when a meaningful value cannot be determined by the implementation.

**Exceptions:** No exceptions are raised other than system exceptions.

Return an implementation-specific string that represents the server interface version for all server interfaces instantiated by the protocol server.

### 3.6.4.6 get_fds_version

The `get_fds_version` method retrieves a version string that identifies the product version. The method signature is specified by the following FSIDL:

```fsidl
string get_fds_version(void);
```

**Input values**

**Void:** No input values.

**Return value**

**string:** This represents an implementation-specific version for all protocol servers constituting the system. An empty string or the string "N/A" is used when the implementation can not determine a meaningful value.

**Exceptions:** No exceptions are raised other than system exceptions.

Return an implementation-specific version string that identifies product version.

### 3.6.4.7 get_middleware_port

The `get_middleware_port` method retrieves the port number used for the channel URI in the protocol server. See section 3.2.3 for details. The method signature is specified by the following FSIDL:
long get_middleware_port(void);

**Input values**

**Void:** No input values.

**Return value**

**long:** The port number used by the channel URI of this protocol server.

**Exceptions:** No exceptions are raised other than system exceptions.

Return the port number used for all AORs contained in the protocol server. The port number MUST be specified by the higher-level implementation as part of the configuration.

### 3.6.4.8 set_tracelevel

The `set_tracelevel` method sets the debug log level for an implementation-specific module in the protocol server. The method signature is specified by the following FSIDL:

```plaintext
void set_tracelevel(in string module_name, in long level);
```

**Input values**

**module_name:** A string that represents an implementation-specific module within the protocol server.

**level:** A long that represents the debug level for the module specified by the module_name input value. Debug logging is disabled when this field is set to 0. The verbosity of debug logging increases with the level number.

**Return value**

**Void:** No return value.

**Exceptions:** No exceptions are raised other than system exceptions.

This method MUST set the debug state for the **Module Name** entry that matches `module_name` in the debug state table to the value represented by `level`.

### 3.6.5 Timer Events

None.

### 3.6.6 Other Local Events

None.

### 3.7 core::fds_component Client Details

#### 3.7.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the
3.7.2 Timers

None.

3.7.3 Initialization

The protocol client calls the resolve method, as specified in section 3.4.4.1 on a name server protocol server with the following input values:

- **name**: Specified by the higher-level implementation.
- **interface_type**: A string with the value "core::fds_component".
- **version**: A string with the value "5.1".

An abstract object reference is created based on the cht::nameservermsg::aor Cheetah entity returned by the resolve method. A client proxy for the core::fds_component object is created based on the abstract object reference, as specified in section 3.3.3.

3.7.4 Message Processing Events and Sequencing Rules

None.

3.7.5 Timer Events

None.

3.7.6 Other Local Events

None.

3.8 core::lifecycle Server Details

3.8.1 Abstract Data Model

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

**Lifecycle State**

This contains the current state for the protocol server. The value MUST be one of the values specified by the core::lifecycle::state enumeration, as specified in section 2.2.31.

A higher-level implementation can modify the lifecycle state according to the runtime state of the process hosting the protocol server. Some protocol servers do not adjust the runtime state of the process correctly, and a malfunctioning protocol server can be prohibited from executing the code that updates the lifecycle state.
3.8.2 Timers

None.

3.8.3 Initialization

The server object for `core::lifecycle` MUST be initialized by a higher-level implementation using the protocol server. The protocol server calls the `bind` method, as specified in section 3.4.4.2.

The input parameter to the `bind` method is a `cht::nameservermsg::aor` Cheetah entity, as specified in section 2.2.18. The `cht::nameservermsg::aor` Cheetah entity sets the `interface_type` attribute to "core::lifecycle" and the `interface_version` attribute to "5.1".

The Lifecycle State field MUST be set to initializing, as specified by the enum in section 2.2.31.

3.8.4 Message Processing Events and Sequencing Rules

This interface includes the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>Sets the state of the protocol server to terminating.</td>
</tr>
<tr>
<td>resume</td>
<td>Sets the state of the protocol server to running.</td>
</tr>
<tr>
<td>suspend</td>
<td>Sets the state of the protocol server to suspended.</td>
</tr>
<tr>
<td>get_state</td>
<td>Return the value of the Lifecycle State field.</td>
</tr>
</tbody>
</table>

The `resume` method is called only after the `suspend` method has been called. No method in this interface is called after the `stop` method has been called. Protocol servers that do not adjust the runtime state of the process hosting the protocol server are not required to follow the sequencing rules.

3.8.4.1 stop

The `stop` method sets the state of the protocol server to terminating. The method signature is specified by the following FSIDL:

```plaintext
void stop(void);
```

**Input values**

**Void:** No input values.

**Return value**

**Void:** No return value.

**Exceptions:** No exceptions are raised other than system exceptions.

The method sets the Lifecycle State field to the value represented by the terminating enum specified in section 2.2.31. The protocol server can call the necessary procedures to terminate the process hosting the protocol server, although this is implementation-specific. Some protocol servers ignore terminating the process hosting the protocol server.
3.8.4.2 resume

The resume method sets the state of the protocol server to running. The method signature is specified by the following FSIDL:

```c
void resume(void);
```

**Input values**

**Void:** No input values.

**Return value**

**Void:** No return value.

**Exceptions:** No exceptions are raised other than system exceptions.

This method sets the Lifecycle State to the value represented by the running constant of the enum specified in section 2.2.31. The protocol server can call the necessary procedures to resume the execution of the process hosting the protocol server, although this is implementation-specific.

3.8.4.3 suspend

The suspend method sets the state of the protocol server to suspended. The method signature is specified by the following FSIDL:

```c
void suspend(void);
```

**Input values**

**Void:** No input values.

**Return value**

**Void:** No return value.

**Exceptions:** No exceptions are raised other than system exceptions.

This method sets the Lifecycle State to the value represented by the suspended constant of the enum specified in section 2.2.31. The protocol server can call the necessary procedures to suspend the execution of the process hosting the protocol server, although suspending is implementation-specific.

3.8.4.4 get_state

The get_state method retrieves the state of the protocol server. The method signature is specified by the following FSIDL:

```c
state get_state(void);
```

**Input values**

**Void:** No input values.
Return value

state: An enumerated type that represents the state of the protocol server, as specified in section 2.2.31.

Exceptions: No exceptions are raised other than system exceptions.

Returns the value of the Lifecycle State.

3.8.5 Timer Events

None.

3.8.6 Other Local Events

None.

3.9 core::lifecycle Client Details

3.9.1 Abstract Data Model

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

3.9.2 Timers

None.

3.9.3 Initialization

The protocol client MUST call the resolve method, as specified in section 3.4.4.1, on a name server protocol server with the following input values:

name: Specified by the higher-level implementation.

interface_type: A string with the value "core::lifecycle".

version: A string with the value "5.1".

An abstract object reference MUST be created based on the cht::nameservermsg::aor Cheetah entity returned by the resolve method. A protocol server creates a client proxy for the core::lifecycle server object based on the abstract object reference, as specified in section 3.3.3.

3.9.4 Message Processing Events and Sequencing Rules

None.

3.9.5 Timer Events

None.
3.9.6 Other Local Events

None.
4 Protocol Examples

The following two samples shows the remote methods that create a client proxy based on an abstract object reference retrieved from a name server protocol server, similar to the procedure described in section 3.3.3.

4.1 Resolve an Abstract Object Reference

This sample calls the resolve remote method, as described in section 3.4.4.1, on a name server protocol server that implements the nameservice::nameserver interface, as described in section 3.4. The relevant parts of the nameservice::nameserver interface are specified by the following FSIDL specification:

```idl
module interfaces {
    module nameservice {
        exception resolve_exception { };

        interface nameserver {
            #pragma version nameserver 1.0
            cht::nameservermsg::aor resolve(in string name,
                        in string interface_type,
                        in string version)
                raises (resolve_exception);
        }
    }
}
```

The cht::nameservermsg::aor Cheetah entity is described in section 2.2.18.

```idl
root entity aor {
    attribute string host;
    attribute int port;
    attribute string interface_type;
    attribute string interface_version;
    attribute longint object_id;
    attribute string bound_name;
};
```

The protocol client calls the resolve method with the following parameters:

```idl
name = "esp/subsystems/processing/dispatcher/0"
interface_type = "core::fds_component"
version = "5.1"
```

The server object for the nameservice::nameserver interface is located at "http://www.cohowinery.com:16099/nameservice::nameserver/1.0/0".

The HTTP headers of the request are as follows:

```text
POST /nameservice::nameserver/1.0/0/resolve HTTP/1.1
Content-Type: application/octet-stream
User-Agent: Middleware client/1.0
```
The request is an HTTP/1.1 request. The HTTP headers are set as described in section 3.2.4.1. The protocol server method URI is "/nameservice::nameserver/1.0/0/resolve", where the remote method name is "resolve" and the server object URI "/nameservice::nameserver/1.0/0".

The body of the request message is as follows:

```
00 00 00 26 65 73 70 2F ...&esp/
73 75 62 73 79 73 74 65 subsystem
6D 73 2F 70 72 6F 63 65 ms/proces
73 73 69 6E 67 2F 64 69 ssing/di
73 70 61 74 63 68 65 72 spatcher
2F 30 00 00 00 00 13 63 /0....co
72 65 3A 3A 66 73 75 62 re::fds_
6D 73 70
```

The interpretation of the preceding message content is as follows:

```
CallArguments:
name, String:
  Length: 0x00000026
  ByteSequence: subsystems/processing/dispatcher/0
interface_type, String:
  Length: 0x00000013
  ByteSequence: core::fds_component
version, String:
  Length: 0x00000003
  ByteSequence: 5.1
```

The HTTP headers of the response are shown as follows:

```
HTTP/1.1 200 OK
Content-Length: 115
Content-Type: application/octet-stream
Server: Microsoft-HTTPAPI/2.0
Date: Mon, 04 May 2009 10:50:56 GMT
```

The body of the response message is as follows:

```
30 10 8F 02 E8 00 00 00 0.?.&...
00 00 00 00 12 77 77 77 ....www
2E 63 6F 6E 70 75 62 73 79 73 ./cohowine.
65 72 65 73 75 62 73 79 73 74 65 ery.com.
6E 74 74 74 74 74 74 74 74 74 ...
```

[MS-FSMW] — v20120630
Middleware Protocol Specification

Copyright © 2012 Microsoft Corporation.

Release: July 16, 2012
The interpretation of the preceding message content is as follows:

```
OutputValue:
  ReturnType: 0x30
MessageContent:
  CallResult:
    CheetahValue:
      CheetahValueContents:
        Cheetah checksum: 0x108f02e8
        Entity TypeId: 0x00000000
        host, LengthPrefixString:
          Length: 0x00000012
          Bytes: www.cohowinery.com
        port, INT32: 0x00003ee3
        interface_type, LengthPrefixString:
          Length: 0x00000012
          Bytes: core::fds_component
        interface_version, LengthPrefixString:
          Length: 0x00000003
          Bytes: 5.1
        object_id, INT64: 0x113D33BE26177801
        bound_name, LengthPrefixString:
          Length: 0x00000026
          Bytes: esp/subsystems/processing/dispatcher/0
```

### 4.2 Call __ping

The next sample uses the `cht::nameservermsg::aor` Cheetah entity returned by the `resolve` method in the previous sample to call the `__ping` remote method on the server object represented by the `cht::nameservermsg::aor`.

The server object represented by the `cht::nameservermsg::aor` Cheetah entity is located at "http://www.cohowinery.com:16099/core::fds_component/5.1/1242205964000000001".

The HTTP headers of the request are shown as follows:

```
POST /core::fds_component/5.1/1242205964000000001/__ping HTTP/1.1
Content-Type: application/octet-stream
User-Agent: Python WinHTTP client/1.0
Host: www.cohowinery.com:16099
Content-Length: 0
Connection: Keep-Alive
```

The request is an HTTP/1.1 request. The HTTP headers are set as described in section 3.2.4.1. The protocol server method URI is "/core::fds_component/5.1/1242205964000000001/__ping", where the remote method name is "__ping" and the server object URI is "/core::fds_component/5.1/1242205964000000001". The server object identifier part of the server object URI, "1242205964000000001", is the hex value 0x113D33BE26177801 converted to base 10. The request has no further message content, as indicated by the `Content-Length` value 0.
The headers of the HTTP response are as follows:

HTTP/1.1 200 OK
Content-Length: 1
Content-Type: application/octet-stream
Server: Microsoft-HTTPAPI/2.0
Date: Mon, 04 May 2009 10:50:56 GMT

The body of the response message is as follows:

30 0

The interpretation of the preceding message content is as follows:

OutputValue:
  ReturnType: 0x30
  MessageContent:
    CallResult:
      Void:
5 Security

5.1 Security Considerations for Implementers

None.

5.2 Index of Security Parameters

<table>
<thead>
<tr>
<th>Security parameter</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP authentication</td>
<td>See section 2.1</td>
</tr>
<tr>
<td>HTTPS</td>
<td>See section 2.1</td>
</tr>
</tbody>
</table>
6 Appendix A: Full FSIDL

For ease of implementation, the full FSIDL and complete listing of Cheetah entities used in this protocol are provided in the following sections.

6.1 FSIDL

For ease of implementation, the following full FSIDL is provided.

```fsidl
module interfaces {
    module core {
        interface fds_component;
        interface lifecycle;

        enum state {
            initializing, running, suspended, terminating
        };

        exception unsupported_guarantee_set {
            string message;
        };

        interface fds_component {
            #pragma version fds_component 5.1
            string get_hostname();
            cht::core::resource_report get_resource_report();
            long uptime();
            string get_version();
            string get_model_version();
            string get_fds_version();
            long get_middleware_port();
            void set_tracelevel(in string module_name, in long level);
        };

        interface lifecycle {
            #pragma version lifecycle 5.1
            void stop();
            void resume();
            void suspend();
            state get_state();
        };
    }

    module nameservice {
        exception not_bound_exception { };
        exception resolve_exception { };
        interface nameserver {
            #pragma version nameserver 1.0
            cht::nameservermsg::aor resolve(in string name, in string interface_type, in string version) raises (resolve_exception);

            void bind(in cht::nameservermsg::aor the_aor);

            void unbind(in string name, in string interface_type, in string version) raises (not_bound_exception);
        }
    }
}
```
cht::nameservermsg::aor_list list_name(in string name_prefix,
    in string interface_type);
cht::nameservermsg::aor_list list_host(in string host,
    in string interface_type);
cht::nameservermsg::aor_list list_any(in string name_prefix,
    in string interface_type,
    in string version,
    in string host);


6.2 Cheetah Entities

root entity aor {
    attribute string host;
    attribute int port;
    attribute string interface_type;
    attribute string interface_version;
    attribute longint object_id;
    attribute string name;
};

root entity aor_list {
    collection aor aors;
};

dentity alloc {
    attribute string name;
    attribute int current;
    attribute int total;
};

dentity scope {
    attribute string name;
    attribute int current;
    attribute int total;
    attribute int min_time;
    attribute int max_time;
    attribute int avg_time;
};

dentity named_value {
    attribute string name;
};

dentity bool_value : named_value {
    attribute bool value;
};

dentity float_value : named_value {
    attribute float value;
};
entity long_value : named_value {
    attribute int value;
};

entity string_value : named_value {
    attribute string value;
};

entity longlong_value : named_value {
    attribute longint value;
};

root entity resource_report {
    attribute longint when;
    collection alloc allocs;
    collection scope scopes;
    collection named_value values;
};
7 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft® FAST™ Search Server 2010

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.
8 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.
9 Index

__ping method__ 34

A

Abstract data model

client (section 3.3.1 34, section 3.5.1 42, section 3.7.1 48, section 3.9.1 52)
client - core::fds_component 48
client - core::lifecycle 52
client - Middleware 34
client - name server 42
core::fds_component client 48
core::fds_component server 43
core::lifecycle client 52
core::lifecycle server 49
Middleware client 34
Middleware common 25
Middleware server 30
name server client 42
name server server 37
server (section 3.2.1 30, section 3.4.1 37, section 3.6.1 43, section 3.8.1 49)
server - core::fds_component 43
server - core::lifecycle 49
server - Middleware 30
server - name server 37
AbstractObjectReference common data type 17
Applicability 11

B

bind method 38

C

Call __ping example__ 56
CallArguments common data type 18
CallResult common data type 16
Capability negotiation 11
Change tracking 63
CheetahValue common data type 16
c::core::alloc_common data type 21
c::core::bool_value common data type 23
c::core::float_value common data type 23
c::core::long_value common data type 23
c::core::longlong_value common data type 24
c::core::named_value common data type 22
c::core::resource_report common data type 21
c::core::scope_common data type 22
c::core::string_value common data type 23
c::nameservermsg::aor common data type 20
c::nameservermsg::aor_list common data type 20
Client

abstract data model (section 3.3.1 34, section 3.5.1 42, section 3.7.1 48, section 3.9.1 52)
initialization (section 3.3.3 35, section 3.5.3 42, section 3.7.3 49, section 3.9.3 52)

local events (section 3.3.6 36, section 3.5.6 43, section 3.7.6 49, section 3.9.6 53)
message processing (section 3.5.4 42, section 3.7.4 49, section 3.9.4 52)
overview 25
Remote Method Invocation method 35
sequencing rules (section 3.5.4 42, section 3.7.4 49, section 3.9.4 52)
timer events (section 3.3.5 36, section 3.5.5 42, section 3.7.5 49, section 3.9.5 52)
timers (section 3.3.2 35, section 3.5.2 42, section 3.7.2 49, section 3.9.2 52)
Client - core::fds_component

abstract data model 48
local events 49
message processing 49
sequencing rules 49
timer events 49
timers 49
Client - Middleware

abstract data model 34
initialization 35
local events 36
Remote Method Invocation method 35
timer events 36
timers 35
Client - name server

abstract data model 42
initialization 42
local events 43
message processing 42
sequencing rules 42
timer events 42
timers 42
Common - Middleware

abstract data model 25
FSIDL specifications 27
initialization 27
local events 30
mapping FSIDL AtomicType 29
mapping FSIDL CheetahEntityName 30
mapping FSIDL EnumName 30
mapping FSIDL ExceptionName 30
mapping FSIDL InterfaceName 30
mapping FSIDL MethodDecl to remote method specifications 28
mapping FSIDL SequenceType 29
mapping remote method reply 29
mapping remote method request 29
message processing 27
local - name server client 43
local - name server server 42
local - server (section 3.2.6 34, section 3.4.6 42, section 3.6.6 48, section 3.8.6 52)
timer - client (section 3.3.5 36, section 3.5.5 42, section 3.7.5 49, section 3.9.5 52)
timer - client - core::fds_component 49
timer - client - core::lifecycle 52
timer - client - Middleware 36
timer - client - name server 42
timer - Common - Middleware 30
timer - core::fds_component client 49
timer - core::fds_component server 48
timer - core::lifecycle client 52
timer - core::lifecycle server 52
timer - Middleware client 36
timer - Middleware common 30
timer - Middleware server 34
timer - server - client 42
timer - server - core::fds_component 48
timer - server - core::lifecycle 52
timer - server - Middleware 36
timer - server - name server 42
timer - server (section 3.2.5 34, section 3.4.5 42, section 3.6.5 48, section 3.8.5 52)
timer - server - core::fds_component 48
timer - server - core::lifecycle 52
timer - server - Middleware 34
timer - server - name server 42
Examples
call __ping 56
overview 54
resolve an abstract object reference 54

F
Fields - vendor-extendable 11
Float common data type 13
FSIDL 59
Full FSIDL 59

G
get_fds_version method 47
get_hostname method 45
get_middleware_port method 47
get_model_version method 47
get_resource_report method 45
get_state method 51
get_version method 46
Glossary 7

I
Implemener - security considerations 58
Index of security parameters 58
Informative references 8
Initialization
client (section 3.3.3 35, section 3.5.3 42, section 3.7.3 49, section 3.9.3 52)
core::fds_component client 49
core::fds_component server 44
core::lifecycle client 52
core::lifecycle server 50
Middleware client 35
Middleware common 27
Middleware server 31
name server client 42
name server server 37
server (section 3.2.3 31, section 3.4.3 37, section 3.6.3 44, section 3.8.3 50)
server - core::fds_component 44
server - core::lifecycle 50
server - Middleware 31
server - name server 37
Interfaces - server
core::fds_component 43
fundamental 10
Introduction 7

L
LengthPrefixedByteSequence common data type 13
LengthPrefixedFloatSequence common data type 15
LengthPrefixedInt32Sequence common data type 14
LengthPrefixedInt64Sequence common data type 14
LengthPrefixedStringSequence common data type 14
list_any method 39
list_host method 40
list_name method 41
Local events
client (section 3.3.6 36, section 3.5.6 43, section 3.7.6 49, section 3.9.6 53)
core::fds_component client 49
core::fds_component server 48
core::lifecycle client 53
core::lifecycle server 52
Middleware client 36
Middleware common 30
Middleware server 34
name server client 43
name server server 42
server (section 3.2.6 34, section 3.4.6 42, section 3.6.6 48, section 3.8.6 52)
servserver - core::fds_component 48
server - core::lifecycle 52
server - Middleware 34
server - name server 42

M
Message processing
client (section 3.5.4 42, section 3.7.4 49, section 3.9.4 52)
suspend method 51
Server - core::fds_component
abstract data model 43
get_fds_version method 47
getHostname method 45
get Middleware port method 47
get_model_version method 47
get_resource_report method 45
get_version method 46
initialization 44
local events 48
message processing 44
sequencing rules 44
set_tracelevel method 48
timer events 48
timers 44
uptime method 46
Server - core::lifecycle
abstract data model 49
get_state method 51
initialization 50
local events 52
message processing 50
resume method 51
sequencing rules 50
stop method 50
suspend method 51
timer events 52
timers 50
Server - Middleware
__ping method 34
abstract data model 30
initialization 31
local events 34
message processing 32
Remote Method Invocation method 32
sequencing rules 32
timer events 34
timers 31
Server - name server
abstract data model 37
bind method 38
initialization 37
list_host method 40
list_name method 41
local events 42
message processing 37
resolve method 38
sequencing rules 37
timer events 42
timers 37
unbind method 39
ServerMethodURI common data type 19
ServerObjectURI common data type 19
set_tracelevel method 48
Standards assignments 11
stop method 50
String common data type 13
suspend method 51
SystemException common data type 17
T
Timer events
client (section 3.3.5 36, section 3.5.5 42, section 3.7.5 49, section 3.9.5 52)
client - core::fds_component 49
client - core::lifecycle 52
client - Middleware 36
client - name server 42
Common - Middleware 30
core::fds_component client 49
core::fds_component server 48
core::lifecycle client 52
core::lifecycle server 52
Middleware client 36
Middleware common 30
Middleware server 34
name server client 42
name server server 42
server (section 3.3.5 34, section 3.4.5 42, section 3.6.5 48, section 3.8.5 52)
server - core::fds_component 48
server - core::lifecycle 52
server - Middleware 34
server - name server 42
Timers
client (section 3.3.2 35, section 3.5.2 42, section 3.7.2 49, section 3.9.2 52)
client - core::fds_component 49
client - core::lifecycle 52
client - Middleware 35
client - name server 42
Common - Middleware 27
core::fds_component client 49
core::fds_component server 44
core::lifecycle client 52
core::lifecycle server 50
Middleware client 35
Middleware common 27
Middleware server 31
name server client 42
name server server 37
server (section 3.3.2 31, section 3.4.2 37, section 3.6.2 44, section 3.8.2 50)
server - core::fds_component 44
server - core::lifecycle 50
server - Middleware 31
server - name server 37
Tracking changes 63
Transport 12
U
unbind method 39
uptime method 46
UserException common data type 17

V

Vendor-extensible fields 11
Versioning 11
Void common data type 13