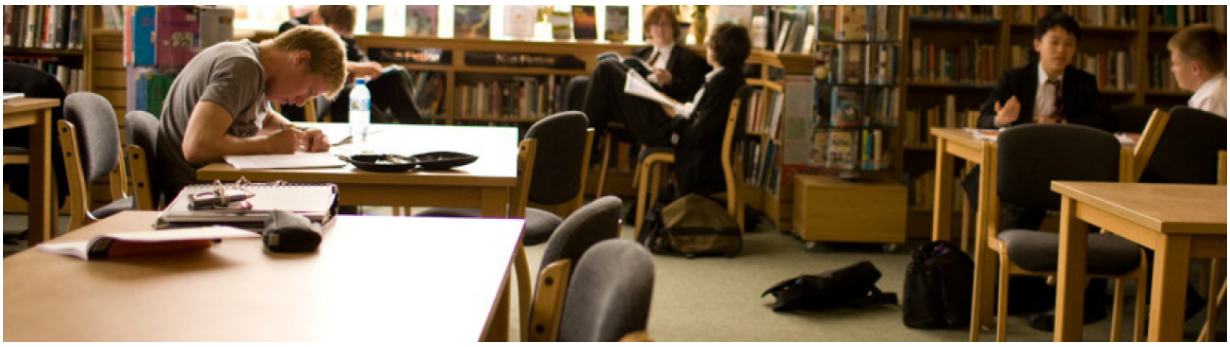
Cisco Data in Motion Application Programming Interface Reference Guide



Release 1.0.1

June 23, 2015

This document provides the information that is required to understand and use the Cisco Data in Motion (DMo) RESTful application programming interface (API).

This document is intended for developers who want to use the Cisco Data in Motion RESTful API to control various DMo features and capabilities and to access DMo resources and data. It assumes the developers have knowledge or experience with Cisco Data in Motion, a web-level programming language, and the following:

- Representational State Transfer (REST) Architecture
- JavaScript Object Notation (JSON)
- Web Services
- Hyper Text Transport Protocol (HTTP) and Secure HTTP (HTTPS)
- Web Distributed Authoring and Versioning (WebDAV)

Contents

Introduction

In the evolution of the Internet what is being termed as the Internet of Things (or IoT) refers to the unique identifiable physical objects and their virtual representations interconnected in a vast network environment. It is also referred to as Internet of Everything (IoE) within certain contexts with distinctions between the two concepts. Cisco defines IoE as the networked connection of people, process, data and things. In contrast, IoT only involves the networked connections of physical objects and data representations and does not include the people and process components. Hence, IoT is a subset of IoE, where IoE comprises of multiple dimensions of technology transitions, including IoT.

Today, more than 99% of things in the physical world are still not connected to the Internet. As sensor devices and nodes are attached to the Internet, they will generate vast amount of data that will need to be processed. The amount of data generated will dwarf the already huge amount of Internet traffic generated today. From research predictions, more than 30 billion devices will be connected to the Internet by 2020.

To address these forthcoming IoT and IoE challenges, Data in Motion (DMo) is a Cisco software technology that provides data management and first-order analysis at the network edge. The primary objective of Data in Motion is to be built into product solutions by Cisco, Cisco customers and partners. DMo rethinks the way data is acquired and managed. It instructs edge devices, such as sensors and sensor gateways as to what data is of interest and what aspects of it to capture. From a user's perspectives, not all data is of interest and the ability to set rules and policies on the edge devices along with capabilities to search the data in real-time and trigger subsequent context-aware actions will inspire powerful technical advantages. The current underlining technical approach is to "store first, analyze later" where all the data is processed in the cloud and backend servers at a later time may not be feasible not only due to the large amounts of data; but also the need to take actions in real-time based on the streaming data (i.e. on data that is in motion and not static). Hence, the key benefits of Data in Motion would be dramatic bandwidth reduction as we reduce the amount of data being sent back by eliminating data that is not needed. Secondly, devices at the edge may have the ability to understand the data and the ability to query the devices for semantic information available in sensor data will be invaluable.

The technical approach towards evangelizing Data in Motion to the masses is to provide an Eclipse Foundation Application Programming Interface (API) for building it into an application ecosystem by Cisco partners and customers. DMo is configurable by these applications to analyze data as it enters and travels over the network. Examples of the analysis can include: (1) finding specific data of interest, (2) summarizing the data in a defined way, (3) processing the data to generate new result data and other application defined analysis. DMo is designed to be deployed and to operate in a distributed and heterogeneous manner in order to manage and control the flood of data from IoT and IoE sensors and data sources.

Another advantage of DMo is to offer different processing and analysis at each location within the network. DMo targets many different heterogeneous Cisco platforms, such as c819 ruggedized router, Connected Grid Router (CGR), Wireless Access Point, processing blade in a router and in a Virtual Machine (VM) in an Unified Computing System (UCS) and within a data center. The many instances of Data in Motion in the network will work together to deliver an integrated data analysis "fabric" for large-scale IoT and IoE.

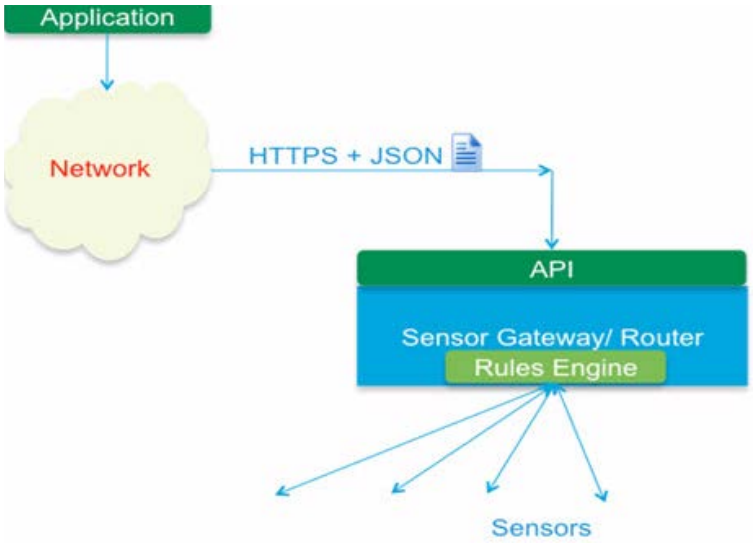
Overview

A top-down system overview is to consider an architecture based on a publish/subscribe mechanism where rules and policies are registered on devices/gateways that have visibility into and communicate with sensors. The rules can be used to describe what data should be acquired. For example, we can acquire data based on network parameters, such as protocol, IP address or port number. The rule can also specify that content payload that matches certain criteria should be processed or extracted. For example, a rule can attempt to acquire data from sensors where the temperature is within a certain range.

The provided API interfaces with the user's programming environment. In essence, a user writes a software program that specifies what data s/he is interested in. The API helps the user translate rules in open standard JSON format encapsulated as a REST message that can be understood by the API.

A key part of the system overview is the format of the JSON messages used to express a policy. The API to the edge device of interest using a RESTful communication paradigm then sends this policy. This is the publish part of the architecture.

Figure 1 API Architecture



An edge device supporting the API will listen to the REST messages containing the JSON payloads that express the rules and will register them. A component of the system runs on the edge device and translates the rules from JSON format to the internal format of the device. The device will be responsible for translating the JSON messages into internal representations that it can understand since these representations are specific to each device. Traffic that flows through the device will be searched against the rules. The devices may have the ability to index and search the payload and content in the sensor data. It also can execute ad hoc queries on the payload. In this manner, the data at the edge of the network can be searched in real-time using the API to discover nuggets of information from the mountain of raw data. The rules may also specify what should be done with the matching traffic. Results of successful hits could be sent (again in a RESTful manner) to an endpoint, which will be listening for responses from the edge device. This is the subscribe part of the architecture.

The DMo architecture provides users the capability to transform the raw data generated by sensors into information. DMo provides RESTful APIs for users to communicate any edge supported edge devices, allowing users to write their own applications to acquire data and events.

A typical user will be interested in transforming data from multiple edge devices generating different types of data into events of interest that can be analyzed by more powerful (usually domain specific) analytics engines in the cloud.

The rules can be used to describe what data should be acquired. Data acquisition can be based on network parameters such as protocol, IP address or port. It is also possible to specify that content payload that matches certain criteria should be processed. For example, a rule may be created to acquire data from sensors where the temperature is within a certain range.

Once the rules are pushed to the edge device using the APIs, any traffic that flows through the device will be searched against the rules. DMo allows the edge device to have the ability to index and search the payload and content in the sensor data and also to execute queries on the payload.

In this manner, the data at the edge of the network can be searched in real-time using DMo to discover information from raw data. Once the information is discovered, users will typically send either the whole or part of this information to an endpoint to perform further operations.

Data in Motion Concepts and Terminology

The following sections provide information about Data in Motion concepts and terminology:

- Dynamic Data Definition
- Windowing
- Network Control
- Application Control
- Dynamic Data Stream
- Dynamic Data Collector

Dynamic Data Definition

Cisco Data in Motion's main configurable concepts and terminology is to support a Dynamic Data Definition (D3).

A D3 is a model representation for the underlying data present at the network. It consists of the following parts:

1. Pattern—A pattern pertains to a combination of Protocol Parameters and Content Parameters.
2. Condition—A condition refers to matching a pattern specifically defined within the request payload, such as the JSON body content.
3. Action—An action consists of either a data management transaction relevant to the underlying data/D3 or a nested set of actions/D3s.

D3 takes the form of the following Condition(s)-Action(s) configurable rule syntax:

IF *Protocol Parameters* **AND** *Content Parameters*

THEN *Actions*

Where:

- Protocol and Protocol Parameters—Pertains to a combination of NETWORK protocol, TRANSPORT protocol and APPLICATION protocol parameters. We have a PROTOCOL representation that includes both APPLICATION and NETWORK parameters; the CONTENT representation refers to the APPLICATION payload. In another case, if the PROTOCOL representation refers to 'ONLY' TRANSPORT protocol parameters (e.g. destination port), the CONTENT representation refers to the TRANSPORT payload.

  For example, if a D3 contains both Application and Network Protocol parameters, the Content refers to the Application payload. In another case, if the D3 contains 'ONLY' Transport protocol parameters (e.g. destination port), the Content refers to the transport payload (e.g. TCP or UDP payload).
- Content and Content Parameters—Refers to the payload, specifically, the data that is not part of any protocol header. This refers to the output of operations on the data sets that reside in the payload. This operation is based on natural parsing, decompression, data extraction, data summarization, decoding, data transformation, etc. The output inherits dynamically the underlying data models. For example, the metadata from pair values are naturally extracted as parametric metadata of any stream, or data records are naturally extracted and exposed from an XML, CSV or JSON payload into their underlying data model.
- Condition—Refers to matching a pattern specifically on the payload.
- Action—Could be one of the following: (action is dependent on Condition):
  - A data management transaction relevant to the underlying data.
  - A nested set of Action calls.
- Event-driven actions—An operation that is executed in the event of pattern matches against the real data.
- Timer driven actions—A special type of action. These are scheduled and can be executed periodically based on a timer.
- Context—A collection of related Data Definitions that can be accessed by an authorized group of users. Each user of a context has a separate username and login password for authentication and authorization security purposes. Each context will have a global (equivalent of root or super-user) password.

  All authorized users have the same access rights to the D3's of a context. A D3 under a context will belong to one unique group. If no user-specified "groupid" is given, the "groupid" defaults to the Context name.

Windowing

Windowing defines meta basic information about a D3 concept. The D3 must contain exactly one windowing Meta primitive.

- Id—A unique Identifier that will identify the D3. It is the responsibility of the user to make sure the Id is unique.
- Context—Defines the container within which the D3 must be installed.
- Timer—Defines how often the D3 ACTIONS will be activated (in seconds)
- Cache—Defines how often the D3 ACTIONS will be activated (in buffer size)

Id and Context are required in every D3. The user must make sure that the Id assigned to a new D3 is unique; otherwise, the installation of D3 will fail.

The user must have login credentials to install a D3 within a specific Context.

Timer and Cache are optional primitives. There can only be a Timer primitive or a Cache primitive.

Network Control

Defines network filters and binary packet decoding of traffic. A D3 may contain 0 or 1 Network primitives.

- FilterBy—A primitive that defines the protocol of interest
- Decode—This primitive must be used when a binary packet must be decoded using a model

The FilterBy primitive must always be used in a Network primitive.

The Decode primitive is used only when binary packets must be decoded.

Application Control

Defines the application filters and content queries as follows:

Application—Defines application filters and queries on the content. A D3 may contain 0 or 1 Application primitives.

- FilterBy—A primitive that defines the application protocol of interest.

The FilterBy primitive must always be used in an Application primitive.

Content Query—Defines queries on the content. A D3 may contain 0 or 1 Content primitives.

- Query—This primitive must be used when a query must be run on the content of the payload of interest.

The Query primitive must always be used in a Content primitive.

Dynamic Data Stream

A Dynamic Data Stream (DDS) is a user-defined stream that transmits the output of a D3 to a user-defined endpoint. A DDS can be defined to be a raw TCP stream or a HTTP stream depending on the requirements of the user. The output of a D3 is pipelined to the endpoint on a single connection.

If configured to be a HTTP or HTTPS stream a DDS performs a HTTP or HTTPS PUT operation on the endpoint.

Endpoint - This primitive specifies the destination Uniform Resource Identifier (URI) to be used by the action.

Dynamic Data Collector

A Dynamic Data Collector (DDC) is a special action that can be used to collect information from a device by initiating a connection to it at an endpoint address specified by the user. A DDC can be configured to connect to the device either using a raw TCP connection or using a HTTP or HTTPS GET depending on the capability of the device.

Endpoint—This primitive specifies the source Uniform Resource Identifier (URI) to be used by the action.

RESTful API

The following sections provide information about using the RESTful API:

- WebDAV Methods
- Data in Motion Configuration using JSON Format
- Primitives for Decoding Raw Data
- Data Models

Primitives for Applying a Content-Type to Raw TCP/UDP Data

The JSON payload syntax for applying a content type to raw TCP/UDP payloads is as follows:

```
"network": {
                .
                .
                .
        "content": MIME_TYPE
    }
```

The MIME_TYPE is a string that is one of the following supported MIME types –

1) application/json
2) text/xml
3) text/csv
4) text/html
5) text/plain

This primitive is useful when messages of one of the above formats are sent as a raw TCP/UDP payloads. By identifying the appropriate content type in the rule the user can help DMo invoke the appropriate parser for the payload.

Note	The default mime type for raw TCP/UDP payloads is ‘text/plain’.

By selecting an appropriate mime type the user can then apply a content query on the payload, as detailed in the following section, to improve the granularity of the event.

- Primitives for Content-Query Condition
- Query Operators
- Query Algebra

Data in Motion Configuration uses RESTful service API implemented using HTTP methods (e.g. GET, PUT, POST, DELETE) and REST principles (e.g. hypertext driven URI). It also leverages from the Web-based Distributed Authoring and Versioning (WebDAV) set of extensions to the HTTP protocol to allow users to manage web resources (e.g. content, documents, videos, images, etc.) intuitively for collaborative editing and managing files on remote web servers. WebDAV is part of the Internet Engineering Task Force (IETF)

approved protocol standard RFC 2518 and the latest proposed 4918 (See http://www.webdav.org/specs/ for additional information).

The API is designed to be RESTful with a JSON over HTTP data format. It supports four major categories of operations to manage D3 entries, such as Create, Read, Update, and Delete (CRUD):

- Create a new D3 in a Context
- Read by listing all existing D3
- Update an existing D3 in a Context
- Delete an existing D3

Data in Motion RESTful-enabled HTTP server supports the WebDAV (RFC 2518 specifications) methods and commands that the WebDAV Methods describes.

You should have received a Cisco license key with your product. The license file must be located under the /home/localadmin/DMRoot/_lic_ directory for Data in Motion services to be enabled. Optionally, the license file can be installed by using the WebDAV HTTP PUT command to upload to the Data in Motion enabled device. Only the admin user can upload/update using WebDAV HTTP PUT command and the HTTP header "Content-Type" MUST be set as "application/lic".

WebDAV Methods

The following sections describe the WebDAV methods:

- PROPFIND
- PUT
- DELETE
- GET

PROPFIND

Retrieves the properties of a D3 (e.g. href, author, creationdate, displayname, etc.). PROPFIND returns a multi-status server-side response with XML body content as follows:

Example 1 PROPFIND Client Request (e.g. Retrieves all properties)

```
PROPFIND / HTTP/1.1
Host: www.cisco.com
Accept-Language: en-us
Authorization: Basic cm9vdDpyb290(NOTE: Basic Authentication with Base64 username:password
encoding)
Connection: Keep-Alive
Content-Language: en-us
Content-Length: xxx
Content-Type: text/xml
Depth: 0
Translate: f
User-Agent: Microsoft Data Access Internet Publishing Provider DAV

<?xml version="1.0" encoding="utf-8" ?>
   <D:propfind xmlns:D="DAV:">
     <D:allprop/>
   </D:propfind>
```

Example 2 PROPFIND Server Response (e.g. Retrieves all properties)

```
HTTP/1.1 207 Multi-Status
Date: Mon, 10 Sep 2007 22:08:38 PDT
Server: DM Version 1.0(Unix)
DAV: 2
MS-Author-Via: DAV
Stream-Length: -1
Content-Type: text/xml; charset=ISO-8859-1
Connection: close

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
<D:response>
<D:href>https://www.cisco.com/Definitions.json</D:href>
<D:propstat>
           <D:prop>
           <D:creationdate></D:creationdate>
           <D:getlastmodified>Sat, 21 Jul 20013 PDT</D:getlastmodified>
           <D:resourcetype>
               <D:collection></D:collection>
</D:resourcetype>
           <D:contenttype></D:contenttype>
                                  <D:getcontentlength>4096</D:getcontentlength></D:prop>
           <D:status>HTTP/1.1 200 OK</D:status>
       </D:propstat>
   </D:response>
</D:multistatus>
```

PUT

Upload a WebDAV resource, such as a document or image through request URI.

Example 3 PUT Client Request

```
PUT /myContext.json HTTP/1.1
Host: www.cisco.com
Accept-Language: en-us
Authorization: Basic cm9vdDpyb290
Connection: Keep-Alive
Content-Language: en-us
Content-Length: 123
Content-Type: application/config
User-Agent: Microsoft Data Access Internet Publishing Provider DAV
```

Example 4 PUT Server Response

```
HTTP/1.1 201 Created
Date: Mon, 10 Sep 2007 22:08:38 PDT
Server: DM Version 1.0(Unix)
Content-Type: text/xml; charset=ISO-8859-1
Connection: close
```

DELETE

Removes a resource (e.g. collection or file).

Example 7 DELETE Client Request

```
DELETE /myContext.json HTTP/1.1
Content-Type: application/config
Authorization: Basic cm9vdDpyb290
Accept: */*
Host: www.cisco.com
```

Example 8 DELETE Server Response

```
HTTP/1.1 204 No Content
Date: Mon, 10 Sep 2007 22:08:38 PDT
Server: DM Version 1.0(Unix)
Connection: close
```

GET

Retrieves or download a resource or HTTP resources through Request URI

Example 9 GET Client Request

```
GET /favicon.ico HTTP/1.1
Authorization: Basic cm9vdDpyb290
Host: www.cisco.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR
1.0.3705; .NET CLR 2.0.50727)
```

Example 10 GET Server Response

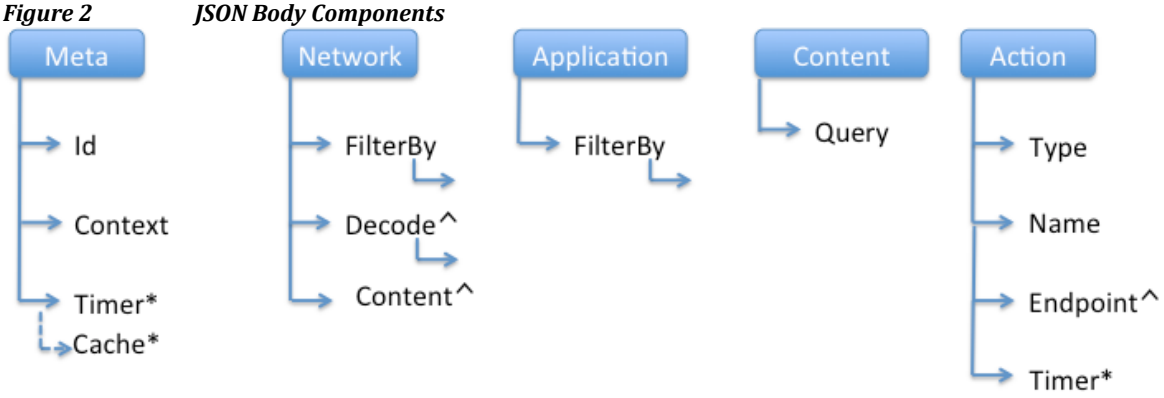
```
HTTP/1.1 200 OK
Date: Mon, 10 Sep 2007 22:08:38 PDT
Server: DM Version 1.0(Unix)
DAV: 2
MS-Author-Via: DAV
Stream-Length: 3064
Content-Type: image/x-icon; charset=ISO-8859-1
Connection: close
```

Data in Motion Configuration using JSON Format

All JSON configurations must be performed using a HTTP PUT operation on the DMo server. The Content-Type of the HTTP PUT message MUST be set to "application/config".

Figure 2 illustrates the components of a JSON body reflecting the parameters needed. These parameters are explained as follows:

Figure 2 *JSON Body Components*



The JSON message that describes a D3 contains the following top-level blocks:

- Meta primitive—This block describes the meta-data about the D3.
- Network primitive—This block describes the network parameters.
- Application primitive—This describes the application level fields of interest.
- Content primitive—This describes what queries are to be run on the content/payload.
- Action primitive—This describes what actions to run and what the triggers are.

Of these blocks, meta and action are required. The other blocks (network, application and content) function mainly as filters for data and may be omitted.

Meta

The meta block contains the following fields:

"*ruleid*"—Name of the D3 and serves as a reference. The ruleid MUST match the JSON filename of the HTTP PUT message.

"*context*"—Specifies the context under which the D3 runs.

"*timer*"—Specifies how frequently the data that has been acquired should be processed. This is specified in milliseconds.

"cache"—Similar to the "meta.timer" field, specifies how frequently (in terms of bytes seen) the data should be processed

Network

The network block describes the network parameters of interest and specifies which data is to be processed. This block contains the following fields:

- "*protocol*"—Specifies the transport protocol we are interested in. e.g. UDP/TCP
- "*decode*"—If payload is encapsulated in TCP or UDP transport without a L7 application header, the decode format string can be used to specify how the payload should be decoded. This allows the user to specify the data model of the payload and to write queries on such payload. It is typically used in situations where the data generator (e.g. sensor) sends a packet with binary payload that needs to be interpreted as integers, characters or other data types.
- "*Filter-by*"—This block specifies the set of 4-tuples (source IP, source port, destination IP, destination port) the user is interested in. This block contains the following fields in the specified formats.
- "*srcaddr*"—ipaddress[/prefix length] : The prefix length can be used to specify a range of IP addresses. Currently, a prefix length of at most 8 (256) addresses is supported. Only raw IP addresses are supported.

- "*srcport*"—Specifies the set of port number ranges, as well as, comma-separated values are accepted. e.g. [3212,1214,2118-2144]
- "*dstaddr*"—The format is similar to that of "*srcaddr*".
- "*dstport*"—The format is similar to that of "*srcport*"

Application

This blocks specifies which application-level protocol fields we want to filter by.

"*protocol*"—Specifies the application or L7 protocol for e.g. HTTP, SMTP etc.

"*Filter-by*"—This block matches directly on name-value pairs in the L7 header, where the name is supplied by the user. For example, user may want to match on "Content-Type" of XML from a specific HOST contained in HTTP protocol. This can be described as follows:

```
"application" :
{
   "protocol" : "http",
   "Filter-by" :
      {
         "Content-Type:" : "application/xml",
         "Host:"         : "10.12.13.14",
      },
}
```

Content

This block specifies what queries are to be run on the content/payload.

- "*query*"—Specifies the query to be run. For example, it can be used to find payload where the pressure is greater than a threshold.

  Example:

  ```
  "query" : "pressure > 23"
  ```

Action

This important block specifies what actions are to be taken in response to the filters and policies set earlier in the D3.

- "*type*"—This specifies the type of the action. Actions can be of two types. A D3 can specify either a list of event-driven actions or a single timer-driven action.

Event-driven actions are executed in response to filters or patterns specified in the D3. These actions are triggered when traffic matches the D3 specifications. For example, a D3 may specify that when JSON traffic is seen encapsulated in HTTP, then the original Header and the Original Payload should be processed. Multiple event-driven actions can be chained together. Examples of event-drive actions are:

- GetHeader - Send original HTTP header back.
- GetPayload - Send original HTTP payload back.

These actions are triggered when the data/traffic is processed which happens at the frequency of "meta.timer".

Timer-driven actions are built-in actions that are not triggered by D3 matches on traffic but are run periodically at intervals. This interval is specified by "action.period". For example, we may want to specify that every 1000 milliseconds (1 second), Data in Motion should go fetch data from sensors (i.e. run the action FETCHDATA periodically).

Timer-driven actions cannot be chained together and there must be only one specified. Actions such as GPSUPDATE can be triggered by both timer, as well as, by rules and can be considered to be of either type.

- "period"—This is specified only for timer-driven actions and is ignored for event-driven actions. It specifies the frequency of operation for timer-driven actions.
- "endpoint"—This block describes the endpoint URI for where the results of the actions are to be sent back.
- "method"—Specifies the protocol to be used. If "http" is specified, the results will be encapsulated in an HTTP header with meta-data about the results. If left empty, the results will be in TCP payload.
- "addr"—IP address of the destination endpoint.
- "port"—Port number for the destination endpoint.
- "resource"—Resource on the destination endpoint which will process the results.

The strings: method:addr:port/resource forms the full URI. Once a D3 has been registered, the client should listen at the specified URI for results.

FETCHDATA and GPSUPDATE are the only timer actions that are supported as of now. These are described in more detail below.

FETCHDATA: This action performs an HTTP GET on the endpoint. The typical usage is to have a FETCHDATA rule to periodically pull data from sensors and another rule that will actually process the data. DMo will automatically pick up the response to the HTTP GET command. It is not mandatory to have a pair of rules always but this is the expected usage.

GPSUPDATE: This is an example of a timer action where DMo does not process the resulting traffic. In this case, DMo periodically sends GPS info to a server using HTTP PUT.

Primitives for Decoding Raw Data

The JSON payload syntax for decoding raw data is as follows:

```
"network": {
                .
                .
                .
        "decode": DATA-MODEL
    }
```

The DATA-MODEL represents the binary data that will be decoded.

Data Models

The data model is used to interpret the binary data as basic data types. As of version 1.0, Cisco Data in Motion will support the interpretation of the following basic data types, such as char, uint16_t, uint32_t. The user has the option to associate variable names to each one of the interpreted fields and later perform queries using these variable names.

A data model is represented in the following format -

{type, #of elements of this type, {var1, var2, var3, var4 …}}

- type—Defines the basic data type. 1 = char, 2 = uint16_t, 4 = uint32_t
- #of elements—Number of consecutive fields of this type that occur in the binary data
- {var1, var2, var3 … }—Name list field. It is a comma separated list of variable names that need to be associated with the binary fields. This is an optional field.

When it is present, the number of fields it contains must be equal to the # of elements. A user may choose to leave a variable name (field) blank if he does not want to associate a name with the binary field. A user may also choose to completely leave out the name list if there is no need to associate names with the variables.

Example 13

A sensor sends out temperature, pressure and humidity data as a 3byte payload. 1st byte corresponds to temperature, 2nd byte corresponds to the pressure and 3rd byte corresponds to humidity. In this situation the data model would be as follows

{1, 3, {t, pres, hum}}

This indicates that there are 3 consecutive bytes that need to be interpreted as u_char. The first byte will be associated the variable name 't', the second byte with variable name 'pres' and the third with 'hum'.

It is not necessary that all fields must be associated with a variable. For example, if the user chooses to associate only the first byte with a variable name on which he can write a query, the data model could be

{1, 3, {t,,}}

Notice that there must be "#of elements -1" commas in the variable name list field. But if a variable name is left blank then that field is not tracked. In another case, if the user wanted to track only the humidity field, the data model could be written as

{1, 3, {,,hum}}

Example 14

Binary payload with multiple data types are also possible. For example, if the payload consists of a total of 10 bytes with the first 4 bytes to be interpreted as uint32_t, the next 4 bytes to be interpreted as 2 consecutive uint16_t with names 'vara' and 'varb', and the next 2 bytes to be interpreted as u_char with names 'ca' and 'cb', the data model would be as follows -

{4,1,{},2,2,{vara, varb},1,2,{ca,cb}}

Note that we did not need to associate a name with the uint32_t field so the name list was left blank.

Example 15

In certain cases it may be required to skip some bytes as they may not hold any meaning in a particular context. Lets say that there is a payload with 25 bytes. The first 21 bytes have no meaning and the last 4 bytes are to be interpreted as a uint32_t and associated with variable name 'h'. The data model would be

{0,21,4,1{h}}

Note that there is no “variable names” field following “0,21,”, as the 0 type does not take any variable names. In addition, empty braces (as shown in Example 14) must not be used.

Note	There will be strict checks on the format of the JSON and any failure in parsing will result in a HTTP 400 bad request at the time of installing of the rule. However, the correctness of the data model is the responsibility of the user and there will be no logical checks performed.

Primitives for Applying a Content-Type to Raw TCP/UDP Data

The JSON payload syntax for applying a content type to raw TCP/UDP payloads is as follows:

```
"network": {
                .
                .
                .
        "content": MIME_TYPE
    }
```

The MIME_TYPE is a string that is one of the following supported MIME types –

1) application/json
2) text/xml
3) text/csv
4) text/html
5) text/plain

This primitive is useful when messages of one of the above formats are sent as a raw TCP/UDP payloads. By identifying the appropriate content type in the rule the user can help DMo invoke the appropriate parser for the payload.

Note	The default mime type for raw TCP/UDP payloads is ‘text/plain’.

By selecting an appropriate mime type the user can then apply a content query on the payload, as detailed in the following section, to improve the granularity of the event.

Primitives for Content-Query Condition

The JSON payload syntax for a content-based query condition is as follows:

```
"content": {
        "query": QUERY-CONDITION
     }
```

A "Query Condition" is a condition set against the specific fields that are found in the payload. Since the fields are present in real-time in the data stream, the notion of real-time schema becomes a valid concept. The schema is derived from the data itself and thus not predetermined.

To determine the underlying schema, specific drivers are used to parse the data and extract, if possibly, an underlying schema.

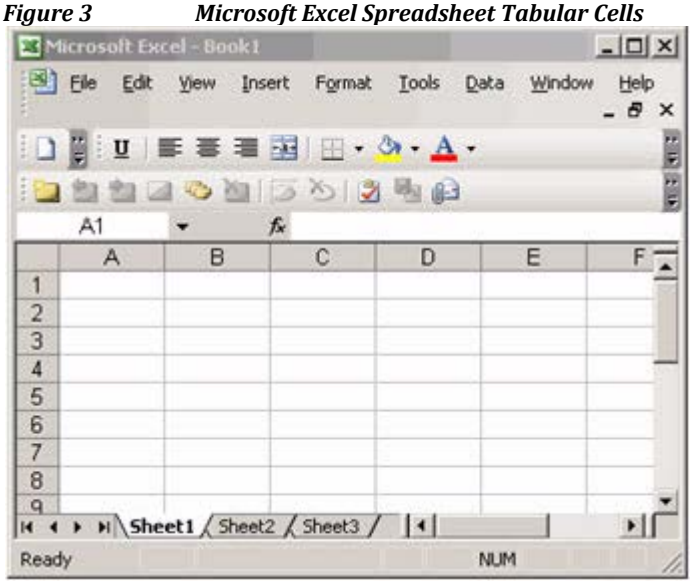
ASCII Streams

The supported formats in the current release are CSV, XML, JSON, TXT, and HTML.

- CSV format—A Comma Separated Values (CSV) does not specify any semantics about the values. The natural parsing of a CSV is that each column is of the same type. Thus, the parser will implicitly assign the letter A for the first column, B for the second column, and so forth, similar to how a Microsoft Excel

spreadsheet represents its default column naming convention. In such an arrangement we can derive implicit attribute names, such as "A=value" or "AA= value" (Figure 2 shows the tabular concept).

Figure 3 Microsoft Excel Spreadsheet Tabular Cells



- XML format—The XML format represents structured and unstructured data. The structured data is represented in the attributes field in the XML tag. The unstructured data are found between begin and end tags as the values.

  The schema representation of an attribute is the attribute itself. In the example below, rdf:about='org:1145' is the value-pair that will be used. Test and Evaluation Department will be used as free text.

```
< pcv:Descriptor rdf:about='org:1145'>
                    <pcv:label>Test and Evaluation Department</pcv:label>
                  </ pcv:Descriptor>
```

- HTML and TXT formats—HTML and TXT formats generate free-text or unstructured data. Each extracted word from HTML and free text format is a valid keyword. The driver parser for HTML will attempt to discard HTML tags.
- JSON format—Key-value pair entry. The schema is inherited dynamically. For nested JSON object, the schema nomenclature follows a typical object model such as in "obj1.obj2.obj3=value."

Binary Streams

Binary requires the schema conversion be defined through a Dynamic Data Definition record - See Theory of Operation document.

Numeric Notations

The system dynamically inherits numeric representation regardless of integer, or floating point representations or data resolution (e.g. 32bits or 64bits).

The "Query Condition" statement is the actual and effective representation of the key-value pair representation. For example, if the data or payload consisted of JSON format, then the object model is used. Let's say the data is given as follows in JSON format:

```
{
   "Sensor":"GPS",
   "Timestamp":1326369894582,
      "State":
       {
       "Longitude":71.30362551,
       "Altitude":25.5,
       "Latitude":42.66195771
       }
}
```

Examples of Data in Motion content query strings in JSON format are as follows:

Example 16 Query for GPS Pair-wise Values

```
"content":  {
   "query": "Sensor=GPS"
}
```

Example 17 Query for a Specific GPS Altitude State Value

```
"content":  {
   "query":  "State.Altitude=25.5"
}
```

Query Operators

Table 1 describes the query operators, followed by examples.

Table 1 Query Operators

Expression	Meaning
Free Text	Key words used as required constant
Text before and after =, <, >	Equality and non-equality operations
Text between =[ ]	Parameter for range evaluation

Example 18 Query is a Simple String

```
"content": {
"query": "Temperature"
}
```

Example 19 Query is a Simple String but Needs a Wild Character

```
"content": {
"query": "Temp*"
}
```

Example 20 Query is an Equality Operation

```
"content": {
"query": "Temperature = 10.2"
}
```

Example 21 Query is a Non-equality Greater-than Operation

```
"content": {
"query": "Temperature > 10"
}
```

Example 22 Query is a Non-equality Less-than Operation

```
"content": {
"query": "Temperature < 10"
}
```

Example 23 Query is a Range Operation

```
"content": {
"query": "Temperature =[10,20]"
}
```

Query Algebra

The notion of algebraic operations is to achieve an operation on the data sets in real-time. The operators allowed are the following "or", "and" and "not". By default, the operator is an "and."

```
"content": {
"query": "Temperature = 10.2F and Pressure = 20PSI"
}
```

```
with an "or" operation
```

```
"content": {
"query": "Temperature = 10.2F or Pressure = 20PSI"
}
```

Sample RESTful API using JSON Data Format Use Cases

Example 24

```
PUT /getData.json  HTTP/1.1
Host: 128.107.165.80 : 8082
Authorization: Basic T3BlbkNvbnRleHQ6Zm9vYmFy
Content-Type: application/config
Content-Length: 727

{
          "meta":  {
                "ruleid" : "getData",
                "context" : "OpenContext",
                "timer" : "3000"
                },
       "network":  {
                "protocol" : "tcp" ,
                 "Filter-by" : {
                          "srcaddr" : "172.27.231.28",
                          "dstaddr" : "10.1.1.2",
                            "sport" : "80",
                            "dport" : "4002",
                         },
                },
    "application   ":  {
                "protocol" : "http" ,
                "Filter-by" : {
                       "Content-Type" : "application/json" ,
                         },
                },
        "action":  {
                "type" : "event",
                "name":  ["GetPayload" ] ,

                "endpoint" :  {
                       "method" : "http" ,
                       "addr" : "172.27.231.28" ,
                       "port" : "5001" ,
                       "resource" : "/process/sensor.pl" ,
                         },
                },
}
```

Example 25

```
PUT /vijay12.json  HTTP/1.1
Host: 128.107.165.80 : 8082
Authorization: Basic T3BlbkNvbnRleHQ6Zm9vYmFy
Content-Type: application/config
Content-Length: 845

{
          "meta":  {
                "ruleid" : "vijay12",
                "context" : "OpenContext",
                "timer" : "3000"
                },
       "network":  {
                "protocol" : "tcp" ,
                 "Filter-by" : {
                          "srcaddr" : "172.27.231.28",
                          "dstaddr" : "10.1.1.2",
                            "sport" : "80",
                            "dport" : "4002",
                         },
                    "decode" : "(2,1,{A}, 4,2,{B,})",
                },

                "content" :  {
                            "query" : "A>23" ,
                        },
                },
        "action":  {
                "type" : "event",
                "name":  ["GetPayload" ] ,
                "endpoint" :  {
                       "method" : "http" ,
                       "addr" : "172.27.231.28" ,
                       "port" : "5001" ,
                         },
                },
}
```

Example 26

```
PUT /vi12.json  HTTP/1.1
Host: 128.107.165.80 : 8082
Authorization: Basic T3BlbkNvbnRleHQ6Zm9vYmFy
Content-Type: application/config
Content-Length: 393

{
          "meta":  {
                "ruleid" : "vi12",
                "context" : "OpenContext",
                },
        "action":  {
                "type" : "timer",
                "name":  [ "Fetchdata " ] ,
                "period" : "10",
                "endpoint" :  {
                        "method" : "http" ,
                        "addr" : "172.27.231.28" ,
                        "port" : "5001" ,
                        "resource" : "/process/sensor.pl" ,
                          },
                },
}
```

Guidelines for Rule Creation

This section describes the valid values and error information for creating a new rule using JSON. Each must adhere these guidelines or the rule is considered invalid and is ignored during the rule engine parsing.

- Invalid IP address and port—Enforced only for endpoint specification
- Invalid prefix—Only 25–32 allowed
- Invalid subnet specification
- Context name is incorrect and does not match authorized one
- meta.cache value should be >= 128
- meta.timer value should be >= 1000
- Network.protocol value is not valid
- Combinations of ipaddress and ports are too numerous
- One or more network block parameters are not valid
- Data Model Error—Must conform exactly to spec
- Endpoints can be accessed using HTTP or raw tcp only
- Invalid IP address or format—Note that DNS resolution is not supported
- Endpoint port must be in the range 1–65535
- Rule definition file is too large
- Invalid ruleid—Not same as filename
- Invalid context—Not logged in

- Cannot have application block and decode block in the same rule
- HTTP Header field name specified is incorrect
- Action block is missing in JSON
- Action list unparsable
- No actions specified
- Cannot have GetHeader action for L3 rule
- Unknown action specified
- Timer rule takes only one action
- Timer rule cannot have any other blocks specified
- Timer rule cannot have meta.cache or meta.timer
- Timer rule needs period to be specified
- Timer rule can only have fetchdata or gpsupdate actions
- Invalid Action-type specified
- HTTP Header field value specified is incorrect
- Rule file name does not have .json extension
- License can only be uploaded by admin user

Sensor Demo Example

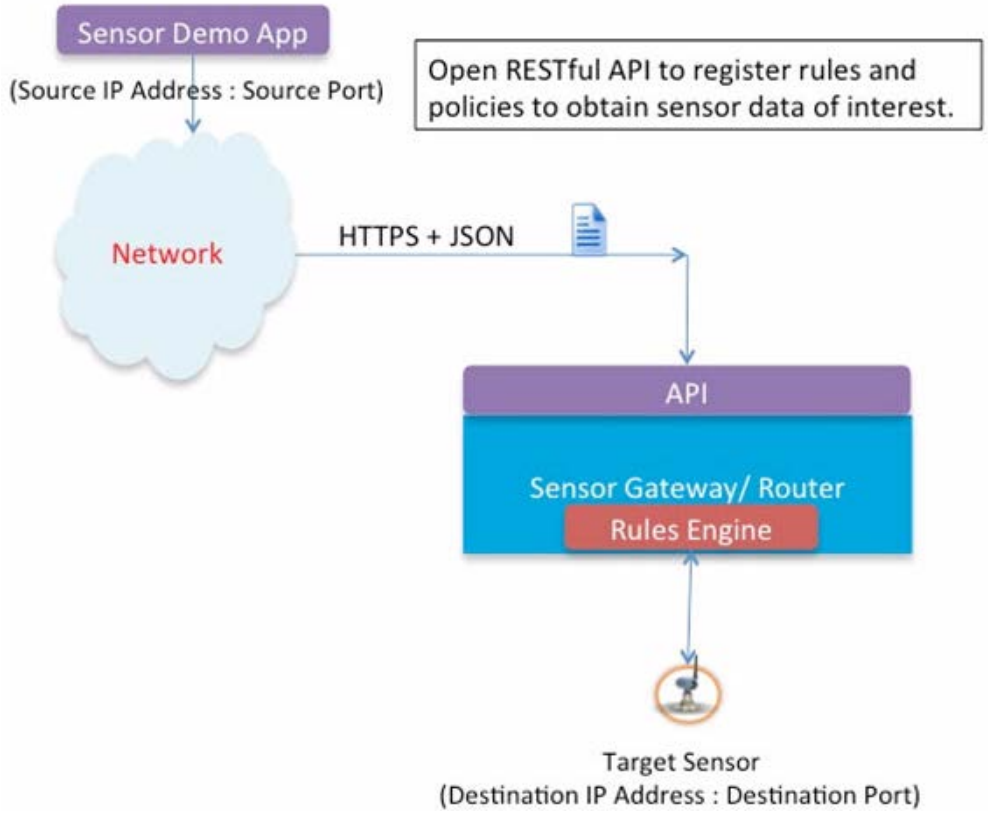
This section provides an end-to-end sensor standalone demonstration application to illustrate the DMo RESTful set of available APIs.

Figure 4 shows the architectural diagram of the sensor demo concepts of operations. It shows how the sensor demo constructs a simple rule called “sensordemo” with the default user context (login userid and password) “DMRootCtx” using JSON format payload and transmitted the encapsulated request over standard HTTPS secure web protocol within the network to register the rule.

The sensor demo runs in server mode in which a destination IP address and a port number are specified as part of the API as an action to send to the target sensor endpoint. In server mode, the sensor demo performs an HTTPS PUT method to upload the rule onto the destination endpoint. The demo can be configured programmatically via the API to operate in client mode, in which the HTTPS GET method is used to fetch the results (for example, querying for specific sensor temperature or pressure values).

The sensor demo is available using either the Java-based or C-based API. There is a default X.509 security certificate and keystore that is included in both sets of APIs (either Java or C-based) to enable HTTPS protocol. For security purposes, only HTTPS is supported in both APIs as the preferred transport protocol.

Figure 4 Sensor Demo Application Architecture



Programmatic Steps

This section shows the programmatic steps for either the Java or C-based API definitions for the sensor demo:

1. Initialize the API library.

   Java

   ```
   api a = api.getInstance(); // Gets an instance of the Java API object
   a.init_api(); // Initializes the API library
   ```

   C

   ```
   init_api(); // Initializes the API library
   ```

2. Create a new request and get a handle reference to it.

   Java

   ```
   h = a.create_request();
   ```

   C

   ```
   h = create_request();
   ```

3. Specific which source IP address and port to register the request.

Java

```
a.set_register_params(h, srcIPAddr, srcPort);
```

C

```
set_register_params(h, argv[1], argv[2]);
// where argv[1] is the first command line argument for the source IP address and argv[2]
is the second argument designating the source port number.
```

4. Check whether the request handle reference is valid. If it is not valid, generate appropriate error message and exit program.

Java

```
if (h == 0) {
         log.severe("Failed to create request\n");
         System.exit(-1);
}
```

C

```
if (!h) {
          fprintf(stderr, "Failed to create request\n");
          exit(-1);
}
```

5. Set context user login credentials (for example, username and password).

Java

```
a.set_login(h, contextUser, contextPasswd); // contextUser=DMRootCtx and contextPasswd=
ciscoDM123
```

C

```
set_login(h, argv[5], argv[6]); // argv[5] designates the contextUser and argv[6]
designates the contextPasswd
```

6. Sets the optional META primitives, such as the timer in seconds or cache size in bytes. NOTE: When cache is set DMo will wait for the output cache to build up to the specified size before sending any output to the destination. When packet payload size of the event is small this can cause a large delay from the time the first event occurred to the time the output is sent to the destination.

Java

```
a.set_meta(h, 0, 2784); // Sets the cache size to 2784 bytes
```

C

```
set_meta(h, 0, 1024); // Sets the cache size to 1024 bytes
```

7. Specify whether to operate in server or client mode. In server mode, specify the destination IP address and port number to send to the target endpoint. In client mode, use the HTTPS GET method to fetch the results from the target location (for example, a web URL address).

Java

```
a.set_action_endpoint(h, "http", destIPAddr, destPort, "");  // server-mode setup for
destination IP address and port number
```

C

```
set_action_endpoint(h, "http", argv[3], argv[4], ""); // where argv[3] is the designated
destination IP address and argv[4] is the designated destination port number
```

8. Optionally specify either TCP or UDP network traffic packets to inspect.

Java

```
a.set_net_filter(h, "udp", "", "", "", ""); // Filter only UDP traffic
```

C

```
set_net_filter(h, "udp", "", "", "", ""); // Filter only UDP traffic
```

9. Optionally specify HTTP network traffic with following content type, content length, and host parameters:

Java

```
a.set_application_proto(h, "http");  // Only interested HTTP network traffic
a.add_application_filter(h, "Content-Type", "multipart/x-mixed-replace;");  // Filter on
Content-Type=multipart/x-mixed-replace
```

C

```
set_application_proto(h, "http");
add_application_filter(h, "Content-Type", " multipart/x-mixed-replace;");
```

10. Add rule called “sensordemo” to the JSON request payload and enable SSL certificate for HTTPS.

Java

```
a.setSSLKeyStore(javaKeyStore);
a.set_operation(h, "ADDRULE");
a.set_ruleid(h,"sensordemo");
a.register_request(h, enable_ssl);
```

C

```
set_operation(h, "ADDRULE");
set_ruleid(h,"sensordemo" );
register_request(h, enable_ssl);
```

Sample Sensor Demo JSON Request Payload:

```
{
    "meta" :        {
                       "ruleid" : "sensordemo",
                       "context" : "DMRootCtx",
                       "cache" : "2784" ,
                    },
    "network" :     {
                       "Filter-by" :    {
                                        },
                    },
    "application" : {
                       "protocol" : "http" ,
                       "Filter-by" : {
                                     "Content-Type" : "multipart/x-mixed-replace;" ,
                                     },
                    },
    "content" :     {
                    },
    "action":       {
                       "type" : "event",
                                "name" : [ "GetPayload" ] ,
                                "endpoint" :  {
                                        "method" : "http" ,
                                        "addr" : "10.0.2.15" ,
                                        "port" : "5001" ,
                                              },
                    },
}
```

Context Management

A context is an associated collection of related Data Definitions that can be accessed by an authorized set of users. Each user of a context has a separate login username and password for security authentication and authorization. Data in Motion offers RESTful APIs for an admin to create/modify/delete a context.

The admin MUST perform an HTTP GET with the corresponding url patterns below and MUST ALSO include the basic authorization header with username and password set to the admin user. Create context allows the DMo Administration to add a context user with a global password and configuration. Update context allows the Dmo Administrator to update username and password within a given context based on existing and new password. Delete context allows the DMo Administrator to disable the context and prevent users from logging in. You may be able to recover the contents of the context after this operation. (Request guidance from Cisco Support) .

The URL pattern for create/modify/delete are formatted in the following ways:

Create Context	**https://<ipaddr>:<configport>/*/C/<context name>/<context_password>**
Update Context	**https://<ipaddr>:<configport>/*/U/<context_name>/<new password>**
Delete Context	**https://<ipaddr>:<configport>/*/D/<context_name>**

<ipaddr> - The ip of the host on which DMo is running

<configport> - Configuration port of DMo. Default is 443. Consult your admin if a custom port has been configured

<context_name> - Name of the context you are creating/updating

<context_password> - Password of the context you are creating

`<new_password> - Must be different from the current password`

Acronyms and Terms

DDC

Dynamic Data Collection is a special action that can be used to collect information from a device by initiating a connection to it at the endpoint address specified by the user.

DDD (D3)—Dynamic Data Definition is a model representation of the underlying data present at the network.

DDS—Dynamic Data Stream is a user-defined data stream that transmits the output of a D3 to a user-defined endpoint.

DMo—Data in Motion reflects the instantiation of TigerMe in the scope of Cisco productization roadmaps.

JSON—JavaScript Object Notation used for standard data interchange format.

TigerMe—Name of a technology that Cisco acquired in 2012.

WebDAV—Web Distributed Authoring and Versioning is a set of standard HTTP extensions that allows users to manage web resources. It is part of the Internet Engineering Task Force (IETF) approved protocol standard RFC 2518.

XML—eXtensible Markup Language (XML) used for standard data interchange format markup language.

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**