Document Properties

Document Number:	
Document Title:	**NetIDE Intermediate Protocol v1.4.1**
Document Responsible:	Roberto Doriguzzi Corin
Document Editor:	Roberto Doriguzzi Corin
Authors:	Roberto Doriguzzi Corin
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.4.1

Production Properties:

Reviewers:	

Document History:

Revision	Date	Issued by	Description
1.1	2015-08-07	Roberto Doriguzzi Corin	NetIDE Protocol version 1.1
1.2	2015-11-25	Roberto Doriguzzi Corin	NetIDE Protocol version 1.2
1.3	2016-03-08	Roberto Doriguzzi Corin	NetIDE Protocol version 1.3
1.4.1	2016-09-09	Roberto Doriguzzi Corin	NetIDE Protocol version 1.4.1

Disclaimer:

This document has been produced in the context of the NetIDE Project. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7) under grant agreement n° 619543.

REVIEW	Main reviewer	Second reviewer
Summary of suggested changes	• one • two	
Recommendation	( ) Accepted ( ) Major revision ( ) Minor revision	
Re-submitted for review	day/Month/year	
Final comments Approved	day/Month/year	

Abstract:

The NetIDE Intermediate protocol implements the following functions: (i) to carry management messages between the Network Engines layers (Core, Shim and Backend); e.g., to exchange information on the supported SBI protocols, to provide unique identifiers for application modules, implement the fence mechanism, (ii) to carry event and action messages between Shim, Core, and Backend, properly demultiplexing such messages to the right module based on identifiers, and (iii) to encapsulate messages specific to a particular SBI protocol version (e.g., OpenFlow 1.X, NETCONF, etc.) with proper information to recognize these messages as such.
This document provides the specification of the NetIDE Intermediate Protocol v1.4.1.

Keywords:

NetIDE, Network Engine, Network protocol

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

JSON JavaScript Object Notation

NETCONF Network Configuration Protocol

SBI Southbound Interface

XML eXtensible Markup Language

List of Corrections

1 Introduction

The NetIDE Network Engine (Fig. 1.1) integrates a client controller layer that executes the modules that compose a Network Application and interfaces with a server SDN controller layer that drives the underlying infrastructure. In addition, it provides a uniform interface to common tools that are intended to allow the inspection/debug of the control channel and the management of the network resources.
The Network Engine provides a compatibility layer capable of translating calls of the network applications running on top of the client controllers, into calls for the server controller framework. The communication between the client and the server layers is achieved through the so-called NetIDE intermediate protocol, which is an application-layer protocol on top of TCP that transmits the network control/management messages from the client to the server controller and vice-versa.
Between client and server controller sits the Core Layer which also "speaks" the intermediate protocol. The core layer implements three main functions: (i) interfacing with the client backends and server shim, controlling the lifecycle of controllers as well as modules in them, (ii) orchestrating the execution of individual modules (in one client controller) or complete applications (possibly spread across multiple client controllers), (iii) interfacing with the tools.

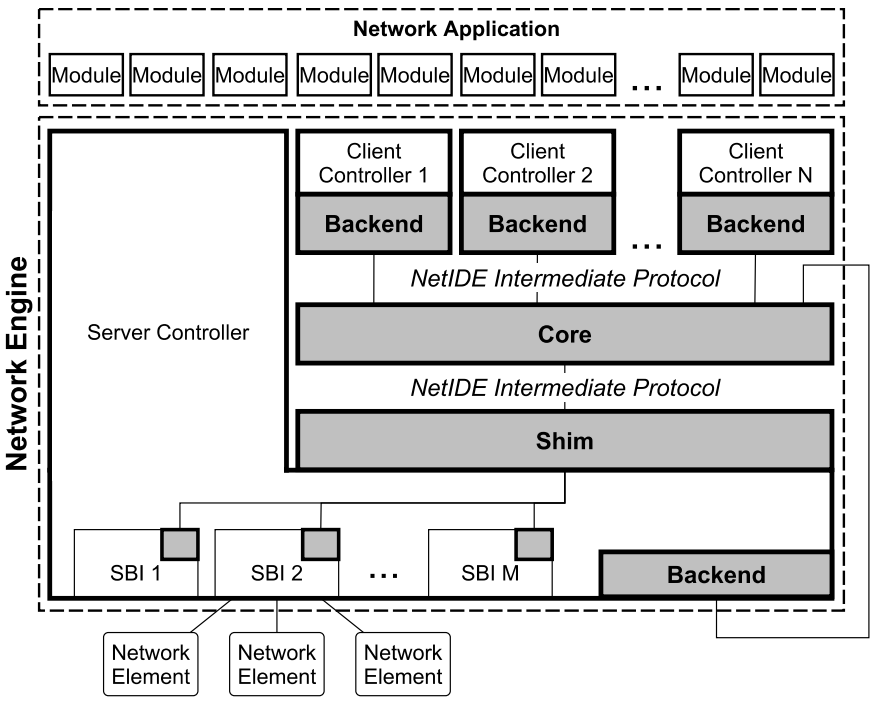


Figure 1.1: The NetIDE Network Engine.

The Intermediate Protocol serves several needs, it has to: (i) carry control messages between

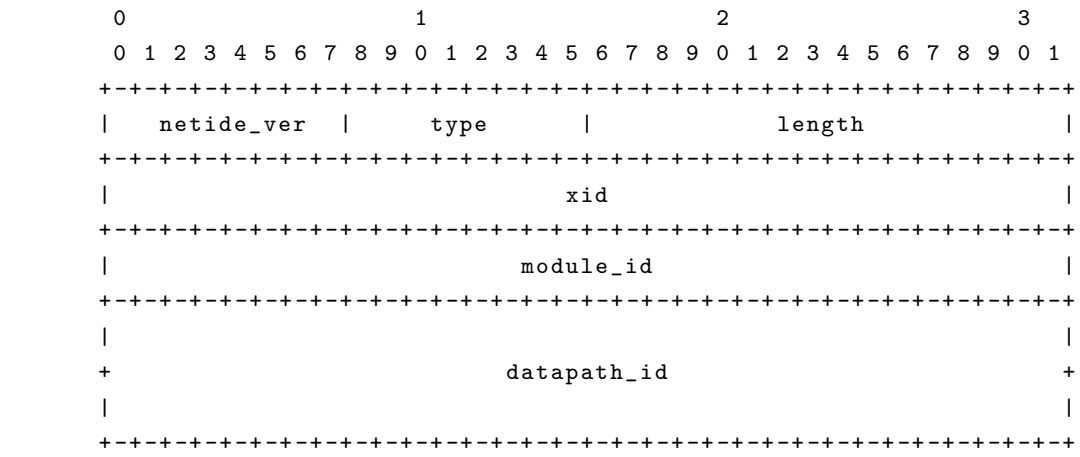
core and shim/backend, e.g., to start up/take down a particular module, providing unique identifiers for modules, (ii) carry event and action messages between shim, core, and backend, properly demultiplexing such messages to the right module based on identifiers, (iii) encapsulate messages specific to a particular SBI protocol version (e.g., OpenFlow 1.X, NETCONF, etc.) towards the client controllers with proper information to recognize these messages as such.
In the remainder of this document we provide details of specification v1.4.1 of the NetIDE Intermediate Protocol.

2 Protocol specification

Messages of the NetIDE protocol contain two basic elements: the NetIDE header and the data (or payload). The NetIDE header, described below in Section 2.1, is placed before the payload and serves as the communication and control link between the different components of the Network Engine. The payload carries management messages or the SBI messages issued by either client controllers or network elements.

2.1 The NetIDE protocol header

The NetIDE header is defined as follows:



where each tick mark represents one bit position. Alternatively, in a `C-style` coding format, the NetIDE header can be represented with the following structure:

```
struct netide_header{
        uint8_t netide_ver;
        uint8_t type;
        uint16_t length;
        uint32_t nxid
        uint32_t module_id
        uint64_t datapath_id
};
```

The `netide_ver` is the version of the NetIDE protocol (the current version v1.4.1 which is identified with value 0x05), `length` is the total length of the payload in bytes and `type` contains a code that indicates the type of the message according with the following values[1]:

[1] `NETIDE_MGMT` and `NETIDE_TOPOLOGY` message types are not documented in the current specification. They have been introduced to allow future extensions of the Network Engine capabilities.

```
enum type{
        NETIDE_HELLO            = 0x01,
        NETIDE_ERROR            = 0x02,
        NETIDE_MGMT             = 0x03,
        NETIDE_MODULE_ANN       = 0x04,
        NETIDE_MODULE_ACK       = 0x05,
        NETIDE_HEARTBEAT        = 0x06,
        NETIDE_TOPOLOGY         = 0x07,
        NETIDE_FENCE            = 0x08,
        NETIDE_OPENFLOW         = 0x11,
        NETIDE_NETCONF          = 0x12,
        NETIDE_OPFLEX           = 0x13,
        NETIDE_OFCONFIG         = 0x14,
        NETIDE_OTHER            = 0xFF
};
```

`datapath_id` is a 64-bits field that uniquely identifies the network elements. `module_id` is a 32-bits field that uniquely identifies Backends and application modules running on top of each client controller. The composition mechanism in the Core layer leverages on this field to implement the correct execution flow of these modules. Finally, `nxid` is the transaction identifier associated to the each message. Replies must use the same value to facilitate the pairing.

2.2 Module announcement

The Core executes composition and conflict resolution operations based on a configuration file which specifies how the applications modules cooperate in controlling the network traffic. In particular, configuration parameters determine the way the Core handles the messages received from the applications modules running on top of the client controllers. To this purpose, each message is encapsulated with the NetIDE header containing a `module_id` value that identifies the module that has issued the message.
`module_id` values are assigned by the Core during the modules announcement/acknowledge process described in this Section. As a result of this process, each Backend and application module can be recognized by the Core through an identifier (the `module_id`) placed in the NetIDE header.

As a first step, Backends register themselves by sending a module announcement message (message type `NETIDE_MODULE_ANN`) to the Core containing a human-readable identifier such as: `backend-<platform_name>-<pid>`. Where `platform_name` is the name of the client controller platform (*ryu*, *onos*, *odl* and *floodlight* can be used) and `pid` is the process ID of the instance of the client controller which is performing the registration. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_MODULE_ANN
        length                  = len("backend-<platform_name>-<pid>")
        nxid                    = 0
```

```
        module_id               = 0
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

The answer generated by the Core (message type NETIDE_MODULE_ACK) includes a module_id value and the Backend name in the payload (the same indicated in the NETIDE_MODULE_ANN message):

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_MODULE_ACK
        length                  = len("backend-<platform_name>-<pid>")
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

After this step, all the messages generated by the Backend (e.g., heartbeat and hello messages described in the following Sections) will contain the BACKEND_ID value in the module_id field of the NetIDE header. Furthermore, BACKEND_ID is used to register the application modules that are running on top of the client controller:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_MODULE_ANN
        length                  = len("module_name")
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = "module_name"
}
```

where module_name is the name of the module under registration. The module's name can be assigned by the Backend or retrieved from the module itself via Application Programming Interface (API) calls. The Core replies with:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_MODULE_ACK
        length                  = len("module_name")
        nxid                    = 0
        module_id               = MODULE_ID
        datapath_id             = 0
        data                    = "module_name"
}
```

After this last step, the Backend allows the application modules to control the network. In particular, network commands sent towards the network (e.g. OpenFlow FLOW_MODs, PACKET_OUTs, FEATURES_REQUESTs) are intercepted by the Backend, which encapsulates them with the NetIDE

header containing the `MODULE_ID` value. Such a value is then used by the Core to recognize the sender of the message and to properly feed the composition and conflict resolution operators.

2.3 Heartbeat

The heartbeat mechanism has been introduced after the adoption of the ZeroMQ messaging queuing library [1] to transmit the NetIDE messages. Unfortunately, the ZeroMQ library does not offer any mechanism to find out about disrupted connections (and also completely unresponsive peers).
This limitation can be an issue for the Core's composition mechanism and for the tools connected to the Network Engine, as they are not able to understand when an client controller disconnects or crashes. As a countermeasure, Backends must periodically send (let's say every 5 seconds) a "heartbeat" message to the Core. If the Core does not receive at least one "heartbeat" message from the Backend within a certain timeframe, the Core considers it disconnected, removes all the related data from its memory structures and informs the relevant tools. In order to minimize the service disruption, the Core applies default policies as specified in the composition specification (e.g. a "drop all" action in case of disconnected firewall module).
The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_HEARTBEAT
        length                  = 0
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = 0
}
```

2.4 Handshake

Upon completion of the connection with the Core (and of the module announcement/acknowledge process for the Backends), Backends must immediately send a `hello` message with the list of the supported control and/or management protocols. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_HELLO
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

Where `data` contains one 2-byte word (in big endian order) for each protocol, with the first byte containing the code of the protocol according to the above `enum`, while the second byte indicates

the version of the protocol (e.g. according to the ONF specification, 0x01 for OpenFlow v1.0, 0x02 for OpenFlow v1.1, etc.). NETCONF version is marked with 0x01 that refers to the specification in the RFC6241 [3], while OpFlex version is marked with 0x00 since this protocol is still in work-in-progress stage [4].
The Shim responds with another `hello` message containing the following:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_HELLO
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

if at least one of the protocols in the request is used between the server controller and the network devices. In particular, `data` contains the codes of the protocols that match the client's request (2-bytes words, big endian order). The `backend_id` value is used in the NetIDE header to allow the Core to forward the reply to the Backend that started the handshake. If none of the requested protocols is supported, the header of the reply is as follows:

```
struct NetIDE_message{
        netide_ver              = 0x05
        type                    = NETIDE_ERROR
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

where the payload of the message `data` contains the codes of all the protocols supported by the server controller (2-bytes words, big endian order).

2.5 The FENCE mechanism

An application module may respond to a given network event (e.g., an OpenFlow `PACKET_IN`) with a set of zero, one or multiple network commands (e.g., OpenFlow `FLOW_MOD`s and `PACKET_OUT`s). The so-called FENCE mechanism is a means for the Core to correlate events and commands and to know when the module has finished processing the input event.
This mechanism is implemented through the message type `NETIDE_FENCE` which is sent by the Backend to the Core once a module has finished processing a network event. Within the same transaction, FENCE message, the network event and related network commands use all the same `module_id` and `nxid` values in the NetIDE header so that the Core can correlate them.

The process is represented in Fig. 2.1, where a `PACKET_IN` event is encapsulated with the NetIDE

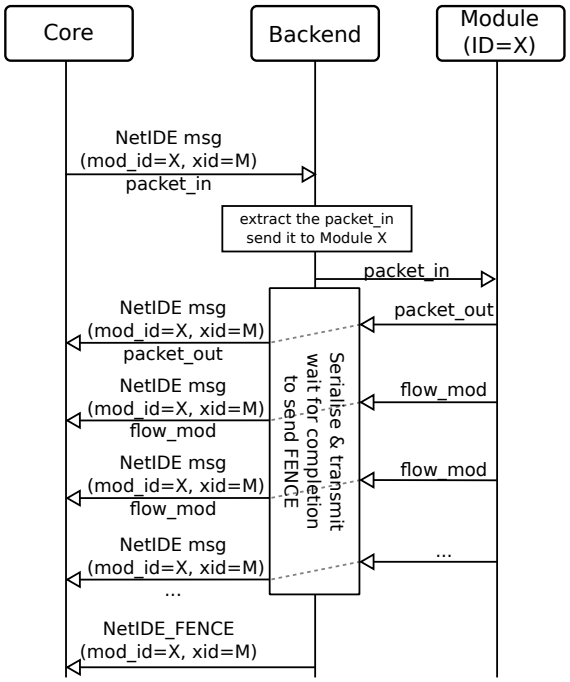


Figure 2.1: Fence mechanism workflow. Both `nxid` and `module_id` refer to the NetIDE header fields.

header by the Core with values `module_id=X` and `nxid=M` and finally sent to the Backend. The Backend removes the NetIDE header and forwards the `PACKET_IN` the the application module `X`. The module reacts with zero, one or multiple network commands (represented by OpenFlow messages `PACKET_OUT` and `FLOW_MOD` in the figure). Each network command is encapsulated by the Backend with the NetIDE header re-using the same `module_id` and `nxid` values received from the Core with the `PACKET_IN`. Therefore, the Core uses the `nxid` value to pair the network commands generated by the module and the previous network event.
Once the module's event handling function returns, the Backend issues a FENCE message to signal the completion of the transaction to the Core.

2.6 The OpenFlow protocol

In this specification, the support for all versions of OpenFlow is achieved with the following:

```
struct netide_message{
        struct netide_header header;
        uint8 data[0]
};
```

Where `header` contains the following values: `netide_ver=0x05`, `type=NETIDE_OPENFLOW` and `length` is the size of the original OpenFlow message which is contained in `data`.

2.6.1 Properly handling reply messages

The NetIDE protocol helps the Network Engine in pairing OpenFlow reply messages with the corresponding requests issued by the application modules running on top of it (e.g. statistics, feature requests, configurations, etc., thus the so-called "controller-to-switch" messages defined in the OpenFlow specifications). In this context, the `xid` field in the OpenFlow header is not helpful, as may happen that different modules use the same values.

In the proposed approach, represented in Fig. 2.2, the task of pairing replies with requests is performed by the Core which replaces the `xid` of the OpenFlow requests with new unique values and stores the original `xid` and the `module_id` it finds in the NetIDE header. As the network elements use the same `xid` values in the replies, the Core can easily pair requests and replies and can use the saved `module_id` to send the reply to the right application module.

The diagram in Fig. 2.2 shows how the Network Engine handles the controller-to-switch OpenFlow messages. The workflow starts with an application module that issues an OpenFlow request with `xid=N`. The Backend relays the message to the Core by encapsulating it with the NetIDE header by using `module_id=X` previously assigned to the application module (see Section 2.2). Once the Core receives this message, it computes a new OpenFlow `xid` value `M` (e.g. by using a hashing algorithm) and ensures that such a value is not being used in other existing transactions.

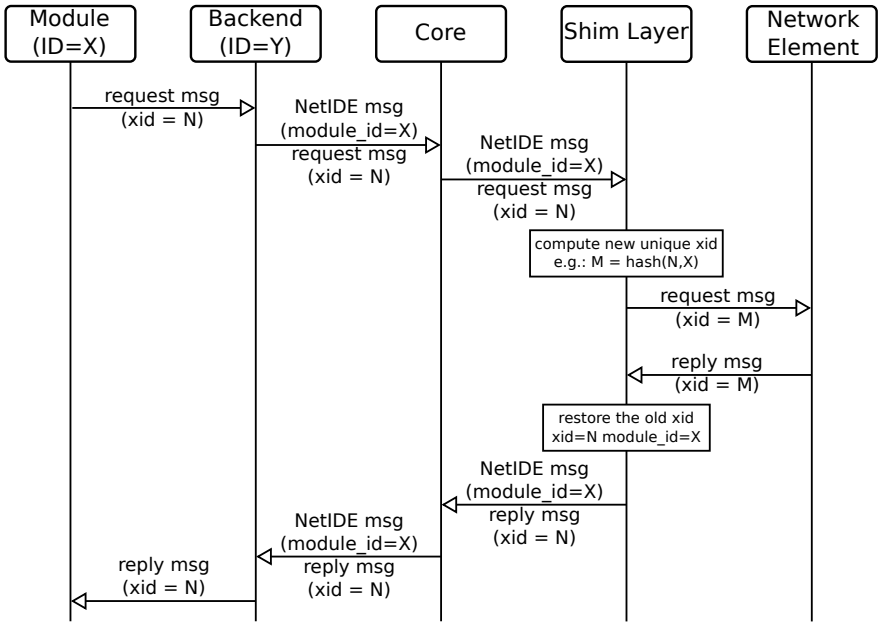


Figure 2.2: Request/reply message handling. `xid` refers to the OpenFlow header field.

Before sending the reply to the Backend, the Core restores the original `xid` in the OpenFlow reply (the application module expects to find in the reply the same `xid` value that was used for the request) and inserts the `module_id` previously saved in the NetIDE header. The Backend will use this information to forward the message to the right application module.

Asynchronous OpenFlow messages generated by the network elements are ignored by the above described tracking mechanism. They are simply relayed to the Backends that eventually forward them to the relevant application modules based on the composition and topology specifications.

Currently defined OpenFlow asynchronous messages are the following:

Message Type	ID	Description	OF Version
OFPT_PACKET_IN	10	New packet received by a switch	1.0-1.5
OFPT_FLOW_REMOVED	11	Flow rule removed from the table	1.0-1.5
OFPT_PORT_STATUS	12	Port added, removed or modified	1.0-1.5
OFPT_ROLE_STATUS	30	Controller role change event	1.4-1.5
OFPT_TABLE_STATUS	31	Changes of the table state	1.4-1.5
OFPT_REQUESTFORWARD	32	Request forwarding by the switch	1.4-1.5
OFPT_CONTROLLER_STATUS	35	Controller status change event	1.5

2.7 Other SBI protocols

The NetIDE intermediate protocol can easily support other SBI protocols, such as NETCONF [3], OF-Config [5] or OpFlex [4].
While OF-Config configurations are only encoded in eXtensible Markup Language (XML) [6], NETCONF and OpFlex specifications are more flexible and support both XML and JavaScript Object Notation (JSON) [7] encoding formats. For this reason, we need an additional field in the NetIDE header to indicate the format of the message contained in `data` and to allow the recipients to correctly handle it. To this purpose, when transmitting NETCONF or OpFlex messages, the sender must set `type=NETIDE_OTHER` in the NetIDE header to indicate the presence of an additional 16-bits field at the end of the header. This field, named `ext_type`, specifies the SBI protocol and the format of the message carried by `data`:

```
struct netide_message{
        struct netide_header header;
        uint16_t ext_type;
        uint8_t data[0];
};
```

Where `header` contains the following values: `netide_ver=0x05`, `type=NETIDE_OTHER` and `length` is the size of the original SBI message carried by `data`. The value of `ext_type` indicates the SBI protocol in the most significant byte (as specified in Section 2.1) and the format of the message (either `0x00` for XML or `0x01` for JSON) in the least significant byte.

3 Bibliography

[1] "ZeroMQ - Distributed messaging," http://zeromq.org/.

[2] "OpenFlow specification 1.5.1," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[3] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)," RFC 6241, IETF, Tech. Rep. 6241, June 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6241.txt

[4] M. Smith et al., "OpFlex Control Protocol," IETF, Tech. Rep., November 2014. [Online]. Available: https://tools.ietf.org/html/draft-smith-opflex-01

[5] "OpenFlow Management and Configuration Protocol (OF-Config 1.1)," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.1.pdf, Jun 2012.

[6] "Extensible Markup Language (XML) 1.0 (Fifth Edition)," https://www.w3.org/TR/xml/.

[7] "Introducing JSON," http://www.json.org/.