Document Properties

Document Number:	
Document Title:	**NetIDE Intermediate Protocol v1.3**
Document Responsible:	Roberto Doriguzzi Corin
Document Editor:	Roberto Doriguzzi Corin
Authors:	Roberto Doriguzzi Corin
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.3

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

Disclaimer:

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 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.
This document provides the specification of the NetIDE Intermediate Protocol v1.3.

Keywords:

NetIDE, Network Engine, Network protocol

Contents

List of Figures

List of Tables

List of Acronyms

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.

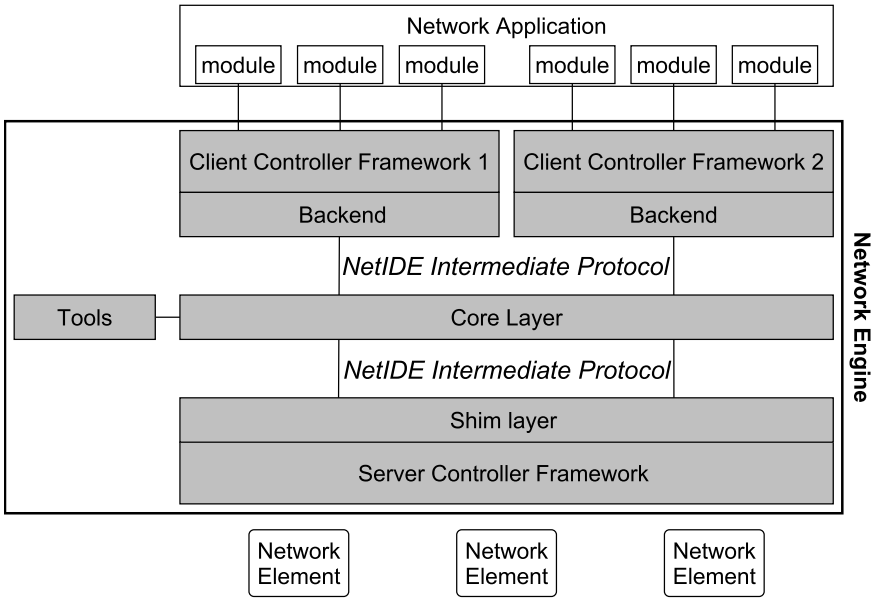


Figure 1.1: The NetIDE Network Engine.

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

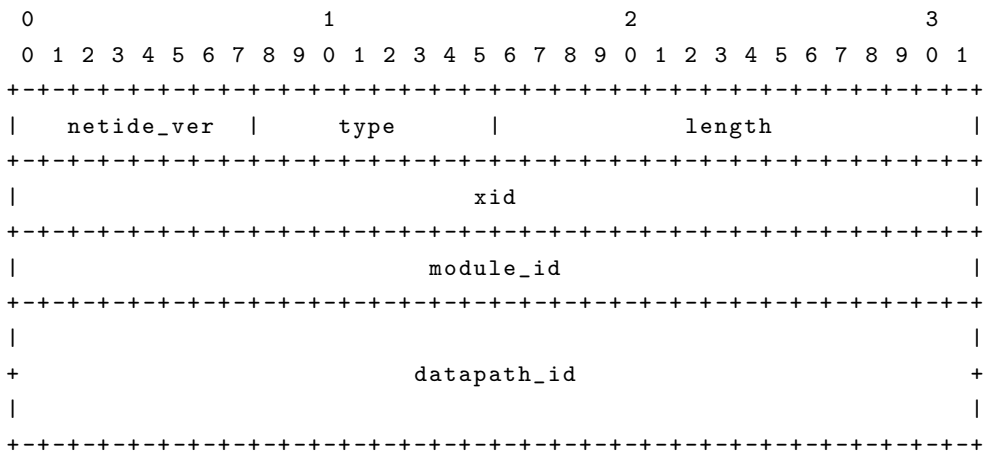
demultiplexing such messages to the right module based on identifiers, (iii) encapsulate messages specific to a particular SBI protocol version (e.g., OpenFlow 1.X, NETCONF, etc.) towards the client controllers with proper information to recognize these messages as such.
In the remainder of this document we provide details of specification v1.3 of the NetIDE Intermediate Protocol, which updates the specification v1.2 reported in Milestone MS4.1 [1] by introducing the FENCE mechanism (Section 2.5).

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 can contain management messages, used by the components of the Network Engine to exchange relevant information, or control/configuration messages (such as OpenFlow, NetConf, etc.) crossing the Network Engine generated by either network application modules or by the 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 xid
        uint32_t module_id
        uint64_t datapath_id
};
```

The `netide_ver` is the version of the NetIDE protocol (the current version v1.3 which is identified with value 0x04), `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:

```
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
};
```

`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, `xid` is the transaction identifier associated to the each message. Replies must use the same value to facilitate the pairing[1].

2.2 Module announcement

The first operation performed by a Backend is registering itself and the modules that it is running to the Core. This is done by using the `MODULE_ANNOUNCEMENT` and `MODULE_ACKNOWLEDGE` message types. 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.
First, a Backend registers itself by using the following schema: `backend-<platform_name>-<pid>`. E.g. a Ryu Backend will register by using the following name in the message `backend-ryu-12345` where `12345` is the process ID of the registering instance of the Ryu platform. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = MODULE_ANNOUNCEMENT
        length                  = len("backend-<platform_name>-<pid>")
        xid                     = 0
        module_id               = 0
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

The answer generated by the Core will include a module ID number and the Backend name in the payload (the same indicated in the MODULE_ANNOUNCEMENT message):

```
struct NetIDE_message{
```

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

```
        netide_ver              = 0x04
        type                    = MODULE_ACKNOWLEDGE
        length                  = len("backend-<platform_name>-<pid>")
        xid                     = 0
        module_id               = MODULE_ID
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

Once a Backend has succefully registered itself, it can start registering its modules with the same procedure described above by indicating the name of the module in the `data` (e.g. `data="Firewall"`). From this point on, the Backend will insert its own module ID in the header of the messages it generates (e.g. heartbeat, hello messages, OpenFlow echo messages from the client controllers, etc.). Otherwise, it will encapsulate the control/configuration messages (e.g. FlowMod, PacketOut, FeatureRequest, NetConf request, etc.) generated by network application modules with the specific module IDs.

2.3 Heartbeat

The heartbeat mechanism has been introduced after the adoption of the ZeroMQ messaging queuing library [2] 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 of the ZeroMQ library can be an issue for the Core's composition mechanism and for the tools connected to the Network Engine, as they cannot understand when an client controller disconnects or crashes. As a consequence, 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. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_HEARTBEAT
        length                  = 0
        xid                     = 0
        module_id               = backend-id
        datapath_id             = 0
        data                    = 0
}
```

2.4 Handshake

Upon a successful connection with the Core, the client controller must immediately send a `hello` message with the list of the control and/or management protocols needed by the applications deployed on top of it.

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

The `header` contains the following values: `netide_ver=0x04`, `type=NETIDE_HELLO` and `length=2*NR_PROTOCOLS`. `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 Core relay `hello` messages to the server controller which responds with another `hello` message containing the following: `netide_ver=0x04`, `type=NETIDE_HELLO` and `length=2*NR_PROTOCOLS` if at least one of the protocols requested by the client is supported. In particular, `data` contains the codes of the protocols that match the client's request (2-bytes words, big endian order). If the handshake fails because none of the requested protocols is supported by the server controller, the header of the answer is as follows: `netide_ver=0x04`, `type=NETIDE_ERROR` and `length=2*NR_PROTOCOLS` and the payload of the message `data` contains the codes of all the protocols supported by the server controller (2-bytes words, big endian order). In this case, the TCP session is terminated by the server controller just after the answer is received by the client.

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 know when the module has finished processing the input event and to pair events and commands.
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, network event and related network command use all the same `mod_id` and `xid` 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 header by the Core with values `mod_id=X` and `xid=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 `mod_id` and `xid` values received from the core with the `PACKET_IN`. Therefore, the Core uses the `xid` value to pair the network commands generated by the module and the previous network event.
As the Backend can understand when a module completes the processing of a network event (i.e.,

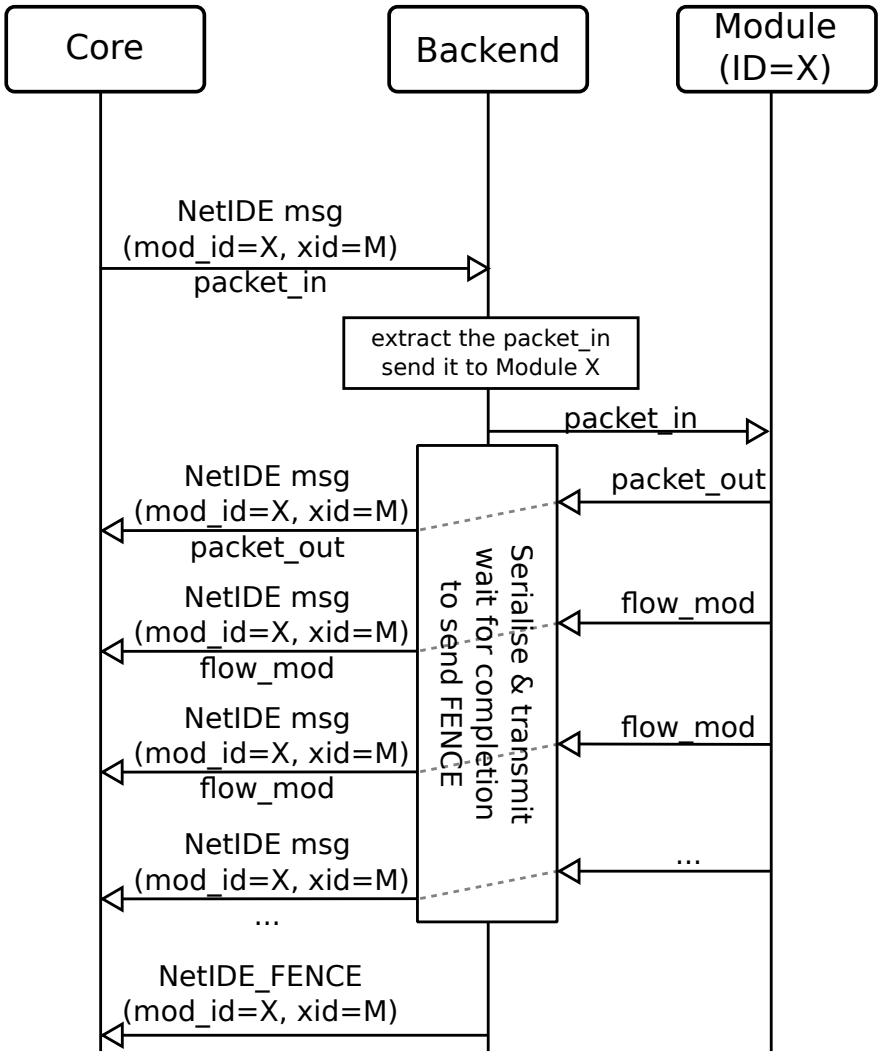


Figure 2.1: Fence mechanism workflow. Both `xid` and `mod_id` refer to the NetIDE header fields.

when the event handling function returns), it generates 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=0x04`, `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

When an application module sends a request to the network (e.g. flow statistics, features, etc.), the Network Engine must be able to correctly drive the corresponding reply to such a module. This is not a trivial task, as many modules may compose the network application running on top of the Network Engine, and there is no way for the Core to pair replies and requests. The transaction IDs (`xid`) in the OpenFlow header are unusable in this case, 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 Shim Layer which replaces the original `xid` of the OpenFlow requests coming from the Core with new unique `xid` values. The Shim also saves the original OpenFlow `xid` value and the `module_id` it finds in the NetIDE header. As the network elements must use the same `xid` values in the replies, the Shim layer can easily pair a reply with the correct request as it is using unique `xid` values.

Fig. 2.2, shows how the Network Engine should handle the controller-to-switch OpenFlow messages. The diagram shows the case of a request message sent by an application module to a network element, i.e. the Backend inserts the `module_id` of the module in the NetIDE header (X in the Figure). For other messages generated by the client controller platform (e.g. echo requests) of by the Backend, the `module_id` or the Backend is used (Y in the Figure).

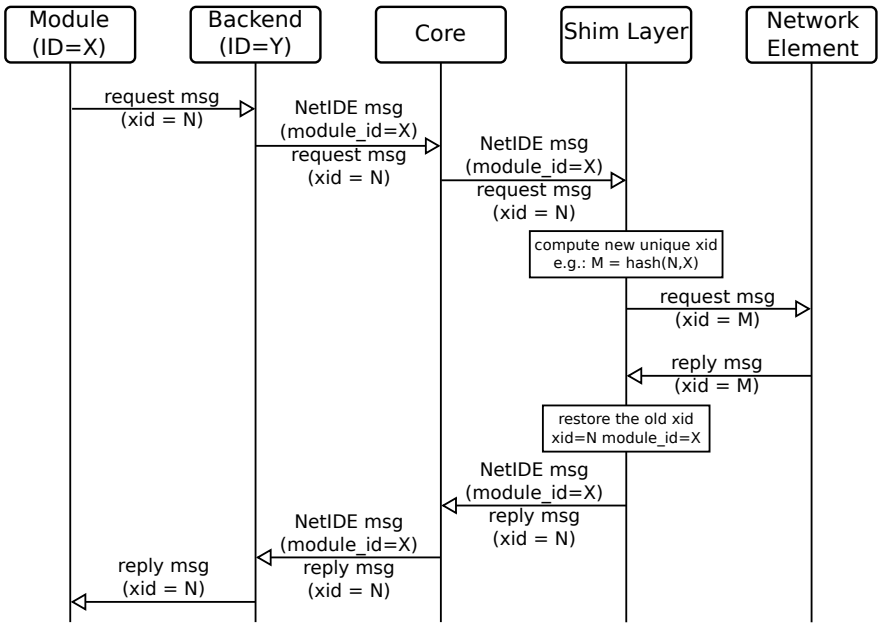


Figure 2.2: Request/reply message handling. `xid` refers to the OpenFlow header field.

Before sending the reply to the Core, the Shim 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 can insert the `module_id` previously saved in the NetIDE header. The Core will use this information to forward the message to the right application module.

Asynchronous OpenFlow messages generated by the network devices should be ignored by this tracking mechanism in the Shim Layer, as they cannot be considered as *reply messages*. They are

simply relayed to the Core that eventually forwards them to the relevant application modules on the basis of the composition and topology specifications. 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 The NETCONF protocol

In this first version, the NetIDE intermediate protocol supports the NETCONF management protocol as specified in the RFC6241 [3]. In particular, after a successful handshake, the client controller can start sending commands to the switches with the following message structure:

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

Where `header` contains the following values: `netide_ver=0x04`, `type=NETIDE_NETCONF` and `length` is the size of the original NETCONF message contained in `data` plus 1 byte of the `format` field. The value of `format` indicates the format of the message (either eXtensible Markup Language (XML) or JavaScript Object Notation (JSON)) and can be used by external components connected to the Network Engine (such as the Logger or the Debugger) to correctly decode the message contained in the `data` field.

2.8 The OpFlex protocol

OpFlex [4] is an open and extensible policy protocol developed by Cisco and other partners for transferring abstract policies in XML or JSON between a network policy controller and any device, including hypervisor switches and physical switches. Similarly to NETCONF configurations, OpFlex policies can be transmitted through the following NetIDE message:

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

Where `header` contains the following values: `netide_ver=0x04`, `type=NETIDE_OPFLEX` and `length` is the size of the original OpFlex message contained in `data` plus 1 byte of the `format` field. The value of `format` indicates the encoding format of the message (XML, JSON or a binary encoding).

3 Bibliography

[1] The NetIDE consortium, "MS4.1 - Network Engine Progress Report," The European Commission, Tech. Rep., 2015.

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

[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