SDN-WISE Core

This documentation explains in details how SDN-WISE works. We suggest to get started by downloading our code from GitHub:

```
git clone https://github.com/sdnwiselab/sdn-wise-java.git
```

SDN-WISE is written in Java, but we also provide a C porting for Contiki. For this reason depending on the context we will discuss the relevant Java/C code.

First of all let's start with the sdn-wise-java Maven project. You can read more on Maven here https://maven.apache.org/ (https://maven.apache.org/)

This project is divided into three modules:

- Core
- Control
- Data

In this document we will focus on the Core module and in particular on SDN-WISE packets. Architectural details can be found in the papers section of the website so we will take for granted that you are already familiar with the concepts of Control and Data plane, WISE Flow Table, and the messages that SDN-WISE nodes exchange.

The Core module is divided into four packages:

- com.github.sdnwiselab.sdnwise.packet
- com.github.sdnwiselab.sdnwise.flowtable
- com.github.sdnwiselab.sdnwise.function
- com.github.sdnwiselab.sdnwise.util

Packet contains the classes that model the messages exchanged by SDN-WISE nodes, Flowtable contains the classes related to the WISE Flow Table, Function is used to reprogram "Over The Air" a node and Util contains all the stuff that did not fit in the previous packages, basically a class that represent the address of a node, the log formatter used in the project, a class that models the concept of neighbor and a utility class that contains byte manipulation functions.

Packets

The most important class in this package is NetworkPacket. It is extended by all the classes in the package and model the header of the SDN-WISE packets. It contains the common constructors used to instantiate packets, and the relative methods to interact with the header's fields.

All the SDN-WISE Packets have the following header fields:

Byte(s)	Name	Description
0	NET	Identifier of the network
1	LEN	Total length of the packet
2-3	DST	Destination address
4-5	SRC	Source address
6	TYP	Packet type
7	TTL	No. of hops remaining
8-9	NXH	Next hop address

A detailed description of the fields is provided in the original paper on SDN-WISE. For the NetworkPacket class, as well as all the following classes, there is always a getter and a setter for the entries reported in the Packet Layout table. These methods are usually called getName/setName where Name is the value reported in the Name column written in CamelCase. E.g. getNet and setNet, getLen and setLen, etc.

Data

For this kind of packets the field TYP is set to 0. Except the header, this packet is just payload.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10-…	Payload	Data content

Relevant Java Methods

To create a Data packet use the DataPacket class. Its main constructor is:

```
public DataPacket(int net, NodeAddress src, NodeAddress dst, byte[] payload)
```

As specified before there is a setter and a getter for the payload and they are called respectively getPayload and setPayload

Beacon

TYP = 1. The Destination is always broadcast. In this case NXH contains the address of the sink for the source node. This packet reports the distance from the sink of the source node and its battery level.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10	Distance	Distance from the sink in no. of hops
11	Battery	Battery level. Full = 0xFF, Empty = 0x00

Relevant Java Methods

To create a Beacon packet use the BeaconPacket class. Its main constructor is:

```
public BeaconPacket(int net, NodeAddress src, NodeAddress sink, int distance, int battery)
```

there is a getter and a setter for both Distance and Battery, and a getter and a setter for the SinkAddress of the node.

```
public BeaconPacket setSinkAddress(NodeAddress addr)
public NodeAddress getSinkAddress()
```

Report

TYP = 2. Contains the list of the nodes at 1 hop distance from the source node. There is a maximum of 35 neighbors for each Report packet. This packet also reports informations on the distance from the sink of the source node and its battery level. By default this packet is always routed towards the sink of the receiving node. After each node in the list of neighbors there is a link quality indicator for the link between the source node and the node in the list.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10	Distance	Distance from the sink in no. of hops
11	Battery	Battery level. Full = 0xFF, Empty = 0x00
12	NeigborsSize	The no. of nodes in the neighbors' list
13-14	NeighborAddress 1	Address of the 1st neighbor in the list
15	LinkQuality 1	Rssi between the 1st neighbor and the source
…-…	NeighborAddress n	Address of the n-th neighbor in the list
…	LinkQuality n	Rssi between the n-th neighbor and the source

Relevant Java Methods

To create a Report packet use the ReportPacket class.

```
public ReportPacket(int net, NodeAddress src, NodeAddress dst, int distance, int battery)
```

This class extends BeaconPacket and adds a getter and a setter for the number of nodes in the neighbors' list plus different methods to deal with this list.

```
public int getNeigborsSize()
public ReportPacket setNeigborsSize(int value)
public NodeAddress getNeighborAddress(int i)
public ReportPacket setNeighborAddressAt(NodeAddress addr, int i)
public int getLinkQuality(int i)
public ReportPacket setLinkQualityAt(byte value, int i)
public HashMap<NodeAddress, Byte> getNeighbors()
public ReportPacket setNeighbors(HashMap<NodeAddress, Byte> map)
```

Request

TYP = 3. This packet encapsulates a packet that has no match in a Flow Table. By default a Request packet is always routed toward the sink of the receiving node. If the unmatched packet fits in a single Request packet then Part is set to 0 and Total = 1. Otherwise, it is broken into two pieces and encapsulated in two different Request packets. In this case the first part will have Part = 0 and Total = 2, while the second part will have Part = 1 and Total = 2. Each request has an ID. An ID is repeated after 255 requests. The unmatched packet starts at byte 13.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10	Id	Identifier of the request
11	Part	Which part of the Request packet has been sent
12	Total	The total no. of parts of the Request packet
13-…	Unmatched Packet	The unmatched packet

Relevant Java Methods

To create a Request packet use the following static method of the RequestPacket class:

```
public static RequestPacket[] createPackets(int net, NodeAddress src, NodeAddress dest, byte id, byte[] buf)
```

This method returns an array of RequestPackets depending on the size of the unmatched packet. To merge two Request Packet and get the unmatched packet use:

```
public static NetworkPacket mergePackets(RequestPacket rp0, RequestPacket rp1)
```

Response

TYP = 4. A Response packet contains a Flow Table entry sent from the Control Plane. For further details on the format of an entry and how to create it, please check the FlowTable section of this document. In future version of SDN-WISE this packet may be replaced by a Config packet.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10-…	Rule	The rule sent from the Control Plane

Relevant Java Methods

To create a Request packet use the following constructor of the ResponsePacket class:

```
public ResponsePacket(int net, NodeAddress src, NodeAddress dst, FlowTableEntry entry)
```

OpenPath

TYP = 5. The OpenPath packet is used to reduce the number of control messages sent from the control plane. In particular it creates a path in the network. The content of the packet is basically a set of windows and a list of addresses. The list of addresses creates the path in the network. After receiving this kind of packet, each node in the path will be able to reach the first and the last node in the path, by sending its packets to the previous (for the first) or the next node (for the last) in the list. As an example a node at position n will learn two rules:

- IF DST == last_node_in_the_path THEN forward to node_n+1
- IF DST == first_node_in_the_path THEN forward to node_n-1

It also possible to add some more conditions to the previous rules by adding windows in the packet. For example it is possible to specify a condition on the payload

- IF DST == last_node_in_the_path && Payload[0] == 10 THEN forward to node_n+1
- IF DST == first_node_in_the_path && Payload[0] == 10 THEN forward to node_n-1

To achieve this result we add a window in the OpenPath packet. More details on how to create a window can be found in the Windows section of this document. Finally, when a node has learned its rules, it sends the packet to the following node in the path. This process is repeated until the packet reaches the end of the path.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10	WindowsSize	The no. of windows in the OpenPath packet
11-15	Window 1	The 1st Window added to all the rules
…-…	Window n	The n-th Window added to all the rules

Byte(s)	Name	Description
…-…	Address 1	The 1st Node in the path
…-…	Address k	The k-th Node in the path

Relevant Java Methods

An OpenPath packet can be created using the following constructor:

```
public OpenPathPacket(int net, NodeAddress src, NodeAddress dst, List<NodeAddress> path)
```

Methods are provided to set and get the path and to set and get the list of windows:

```
public OpenPathPacket setPath(List<NodeAddress> path)
public List<NodeAddress> getPath()
public OpenPathPacket setWindows(List<Window> conditions)
public final List<Window> getWindows()
```

Config

TYP = 6. The Config packet is used to read/write configuration parameters. These parameters are:

ID	Name	Description
0	RESET	Hardware reset
1	MY_NET	Node Network ID
2	MY_ADDRESS	The SDN-WISE address of the node
3	PACKET_TTL	The default TTL for a new packet
4	RSSI_MIN	The minimum RSSI value to consider a link reliable
5	BEACON_PERIOD	A Beacon packet is sent every BEACON_MAX seconds
6	REPORT_PERIOD	A Report packet is sent every REPORT_MAX seconds
7	RESET_PERIOD	NXTHOP vs Sink is deleted after RESET_PERIOD reports
8	RULE_TTL	A flow table entry is deleted after ENTRY_TTL seconds
9	ADD_ALIAS	Add a node address alias
10	REM_ALIAS	Remove a node address alias
11	GET_ALIAS	Get the node address alias at a certain position
12	ADD_RULE	Add a flow table entry
13	REM_RULE	Remove a flow table entry at a certain position
14	GET_RULE	Get the flow table entry at a certain position
15	ADD_FUNCTION	Add a function
16	REM_FUNCTION	Remove the function at a certain position
17	GET_FUNCTION	Get the function ID at a certain position

The Config packet contains as payload the ID of the parameters to read/write, a bit indicating if the packet is a “read” or a “write” and a field of variable size containing the value to write or the value read.

- READ The Control plane creates a Config Packet, with the read/write bit set to 0, and specifies the ID of the parameter. The node will send back the response with the reading appended in the packet.
- WRITE The Control plane creates a Config Packet, with the read/write bit set to 1, and specifies the ID of the parameter and the new value. The node receives the packet and updates its values accordingly.
- GET/ADD/REMOVE If the Control plane wants to GET or REMOVE an entry/alias/function it has to specify the index of the item to get/remove or the item itself if it wants to add it.

Anyway, from a user points of view, the AbstractController class contains all the methods needed to read/write/get/add/remove thus no direct interaction with the packets is required. More details will be provided in the Control Plane documentation.

Packet Layout

Byte(s)	Name	Description

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10	ConfigId	R/W bit + ID
11-…	Params(Optional)	The value to be set or read

Relevant Java Methods

To create a Config packet to read a value use the constructor:

```
public ConfigPacket(int netId, NodeAddress src, NodeAddress dst, ConfigProperty read)
```

To create a Config packet to write a value use:

```
public ConfigPacket(int net, NodeAddress src, NodeAddress dst, ConfigProperty write, byte[] value)
```

For more information and some examples on how to use the Config packets check the AbstractController and ControllerInterface classes inside the com.github.sdnwiselab.sdnwise.controller package.

RegProxy

TYP = 7. The RegProxy packet is used to notify the existence of a Sink to the Control Plane and to associate some information with it. Some controllers may register the Sink node as a switch so we provide a Dpid, a MAC address, a physical port, and an InetSocketAddress.

Packet Layout

Byte(s)	Name	Description
0-9	Header	SDN-WISE Packet header
10-17	DPID	DPID of the Sink
18-23	MAC	MAC address of the sink
24-31	Port	Physical port to which the sink is connected
31-34	IP	IP address of the sink
34-35	TCP	TCP port of the sink

Relevant Java Methods

To create a RegProxy packet use the following constructor:

```
public RegProxyPacket(int net, NodeAddress src, String dPid, String mac, long port, InetSocketAddress isa)
```

The only exception to the usual list of setters and getters is for the IP address and TCP port because an InetSocketAddress object is used:

```
public RegProxyPacket setInetSocketAddress(InetSocketAddress isa)
public InetSocketAddress getInetSocketAddress()
```

Work in progress…

FlowTable

The FlowTable of each node consists in a collection of FlowTableEntry objects. Each FlowTableEntry is made of a list of Windows, a list of Actions, and some Statistical information.

Windows

Each com.github.sdnwiselab.sdnwise.flowtable.Window allows to set a condition to be verified in order to execute the actions. A condition is made of three parts a lefthandside, a righthandside, and an operator

For both handsides it is possible to specify the location of the operand (using the method setLhsLocation and setRhsLocation) and the address (using the method setLhs or setRhs).

The loacation of a handsides can be PACKET, STATUS, or CONST (These values are in the com.github.sdnwiselab.sdnwise.flowtable.FlowTableInterface interface) while a the operator can be choosen among a list of possible operators:

```
EQUAL,
GREATER,
GREATER_OR_EQUAL,
LESS,
LESS_OR_EQUAL,
NOT_EQUAL
```

For example: to create a window that checks that the type (i.e. the byte at position 6) of a packet is equal to 10 you can write:

```
new Window()
.setLhsLocation(PACKET)
.setLhs(6)
.setOperator(EQUAL)
.setRhsLocation(CONST)
.setRhs(10)
```

Actions

The actions are the events that are executed when all the conditions are verified. There are different types of actions:

Null

NULL: No Action

Ask

ASK: Send the packet to the controller, asking for a new FlowTableEntry. It does not requires any parameter and can be created using:

```
new AskAction()
```

Drop

DROP: Discard a packet. It does not requires any parameter and can be created using:

```
new DropAction()
```

ForwardBroadcast

FORWARD_B: Forward in broadcast to all the device in range. It does not requires any parameter and can be created using:

```
new ForwardBroadcastAction()
```

ForwardUnicast

FORWARD_U: Forward in unicast to an address. It requires as input the NodeAddress of the device to which the packet is going to be forwarded. As an example, if we want to forward to node 3 then:

```
new ForwardUnicastAction(new NodeAddress(3))
```

Function

FUNCTION: Execute a function installed on the device. It requires the id of the function, that is going to be called and a list of parameters. As an example, if we want to call the function n 5 passing as arguments 10 integers so can use:

```
new FunctionAction(new byte[]{5, 1, 9, 8, 7, 6, 5, 4, 3, 2, 1,}));
```

Match

MATCH: Match the packet against the FlowTable of the node. It does not requires any parameter and can be created using:

```
new MatchAction()
```

Set

SET: Set a value in the status of the node or in the packet. The way a set is executed is similar to the matching in the window, with the difference that the list of operators is the following:

```
ADD = sum
AND = binary AND
DIV = division
MOD = modulo operation
MUL = multiplication
OR = binary OR
SUB = subtraction
XOR = binary XOR
```

It is also possible to specify where the result of the set operation is placed by using the setResLocation function. The possible locations are

```
PACKET
STATUS
```

As an example to set the 13th byte of a packet equal to the sum of the content of byte 12 and byte 13 SET P.11 = P.12 + P.13:

```
new SetAction()
            .setLhsLocation(PACKET)
            .setLhs(12)
            .setOperator(ADD)
            .setRhsLocation(PACKET)
            .setRhs(13)
            .setResLocation(PACKET)
            .setRes(11));
```

Statistics