OpenFlow Cookbook

Moving on from the traditional, configuration-driven network, OpenFlow paves the way to an open, centrally programmable structure providing standard interfaces between the controller and the actual packet forwarding entity.

This book has been divided into two parts to provide you with everything you need to know to develop either an OpenFlow switch or an OpenFlow controller. Starting from the basics of establishing communication channels between the controller and switch, this book explains in detail about the various tables, table operations, and switch and controller procedures.

What this book will do for you...

- Create, maintain, and close an OpenFlow communication channel between the switch and controller
- Manage multiple switches from a single controller and vice versa: manage a single switch from multiple controllers with different controller roles
- Configure an OpenFlow switch using standard OpenFlow controller and switch procedures
- Explore tables present in OpenFlow switches such as flow tables, group tables, and meter tables
- Using controller and switch procedures, program tables within the switch, such as flow tables, group tables, and meter tables

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

Over 110 recipes to design and develop your own OpenFlow switch and OpenFlow controller
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'OpenFlow Channel Connection Establishment (Part 1)'
- A synopsis of the book’s content
- More information on OpenFlow Cookbook

About the Author

Kingston Smiler. S is a seasoned professional with 11 years of experience in software development and pre-sales, encompassing a wide range of skill sets, roles, and industry verticals. He has solid expertise in data communication networking, software-based switching and routing solutions, and virtualization platforms such as OpenStack, OpenDaylight Controller, Docker containers, CoreOS, and so on. He is currently working as an advisor and technical consultant for networking companies in the development of Layer 2 and Layer 3 IP protocols. Kingston also works on various technologies such as MVC-based web and Windows applications, e-commerce frameworks, open source IoT frameworks, single-board computers such as Raspberry Pi and Intel Galileo, and so on.

He is active in various networking standard bodies such as IETF, IEEE, and ONF. He has proposed a couple of drafts in the MPLS working group of IETF. With the current surge in SDN, virtualization, and NFV, his primary focus is in these areas. Kingston completed a BE in computer science from Madras University.
OpenFlow Cookbook

With the tremendous growth in data traffic due to the increase in the number of devices connected to networks, and more and more applications moving into the cloud, service providers are moving towards virtualizing their infrastructure. With storage and computing virtualization, there is a clear necessity to provide virtualization in networks and a standardized way of programming network devices. With large data centers, there is also a clear necessity to change the traditional control plane and data plane paradigm to move towards flow-based switching, which is more generic and can be adapted for different avenues. Today's networks have evolved into complex proprietary systems, with little chance of offering any possibility to carry out experiments on new ideas or protocols. The Open Network Foundation (ONF) has come up with a specification to meet these requirements and has called it OpenFlow. The latest version of this specification is OpenFlow 1.4.

OpenFlow paves the way for the traditional network to be a programmable network, with standard interfaces between the controller (traditionally called a control plane) and the actual packet forwarding entity (traditionally called a data plane). OpenFlow was started as a means to try experimental ideas/protocols on large-scale networks by providing a standard, flow-based table called an OpenFlow table and separating the control plane from the data plane. This offers the possibility to program the forwarding decisions of those network devices supporting the OpenFlow specification in a flexible manner. There are two main components of the OpenFlow specification, the switch and the controller. This book describes in brief both the OpenFlow switch and controller, their operations, and so on.

An OpenFlow switch consists of one or more flow tables and group tables that perform packet lookups and forwarding. Each flow table in the switch contains a set of flow entries. Each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets. The switch communicates with the controller and the controller manages the switch via the OpenFlow channel, using the OpenFlow protocol.

The OpenFlow controller is a software application that manages the switch using the OpenFlow protocol. The controller can add, update, and delete flow entries in flow tables, both reactively (in response to packets) and proactively.

This Cookbook provides an easy, more user-friendly and step-by-step approach to develop and understand OpenFlow switch and controller. Organized into two parts, part 1 explains OpenFlow from the switch point of view and part 2 explains it from the controller’s point of view. You will see a 1:1 mapping between part I and part 2 chapters and recipes to provide a clear demarcation of the operations and responsibilities of the switch and controller.
What This Book Covers

Chapter 1, OpenFlow Channel Connection Establishment

Part 1 describes the various steps and mechanisms involved in establishing an OpenFlow channel from the switch. An OpenFlow channel is used to exchange an OpenFlow message between the switch and the controller.

Part 2 shows the various steps and mechanisms involved for the controller to establish a channel to the switch. An OpenFlow channel is used to exchange OpenFlow messages between the switch and the controller.

Chapter 2, Symmetric Messages and Asynchronous Messages

Part 1 covers the list of symmetric and asynchronous messages sent and received by the OpenFlow switch, along with the procedure for handling these messages.

Part 2 contains the list of symmetric and asynchronous messages sent and received by the controller and the procedure for handling these messages.

Chapter 3, Flow Table and Flow Entry Modification Messages

Part 1 describes in detail the flow table, flow table entries, and procedures for handling flow table messages and flow entry messages, such as adding a new flow, deleting an existing flow, modifying a flow, and so on.

Part 2 demonstrates the procedure to modify the flow table; add, delete, and modify a flow entry in the flow table; and handle the response message from the switch.

Chapter 4, Group Table and Meter Table Modification Messages

Part 1 explains the group table and its entries, the meter table and its entries, and the procedure for handling group and meter modification messages, such as add, delete, modify, and so on.

Part 2 explains the procedure to add, delete, and modify the group and a meter entry, along with the procedure to handle response messages from the switch after sending these messages.

Chapter 5, Handling Multipart Statistics Messages

Part 1 covers multipart messages, in particular multipart statistic messages to get flow statistics, port statistics, group statistics, meter statistics, and so on. This chapter also contains the procedure required for handling multipart statistics messages.

Part 2 shows the procedure to get statistical information, such as flow statistics, port statistics, group statistics, meter statistics, and so on from the switch using multipart statistics messages, along with the procedure for handling response messages from the switch after sending these messages.
Chapter 6, Handling Multipart State Information Messages

Part 1 illustrates multipart messages, in particular multipart state information messages used to get the port description, table description, queue description, and so on, and the procedure for handling multipart state information messages in the switch.

Part 2 describes in detail the procedure for getting state information, such as the port description, table description, queue description, and so on from the switch using multipart state information messages. This chapter also covers the procedure for handling response messages from the switch after sending these messages.

Chapter 7, Handling Bundle Messages

Part 1 describes in detail bundle messages and the procedure required for handling them in switch, such as creating a bundle, opening a bundle, adding a message to a bundle, and so on.

Part 2 illustrates the controller procedure for creating a bundle, opening a bundle, adding a message to a bundle, and so on, along with the procedure for handling the response messages from the switch after sending the bundle messages.

Appendix, Common OpenFlow Headers, Structures, and Error Code, talks about some common message headers, structures, and procedures defined in the OpenFlow specifications.
OpenFlow Channel Connection Establishment (Part 1)

In this chapter we will cover the following topics:

- Connection setup on TCP & TLS
- Connection setup with multiple controllers
- Setting the role of a communication channel towards a controller
- Establishing an auxiliary connection to a controller
- Handling handshake message from a controller
- Handling switch configuration messages from a controller
- Connection interruption procedures
Introduction

OpenFlow channel is a communication medium that is used to send and receive OpenFlow messages between the switch and controller. The switch must create an OpenFlow channel by initiating a connection to the controller. An OpenFlow switch can establish multiple connections to the same or different controllers in parallel, among which one of the controller's connections acts as master and the other controller's connections acts either as slave / equal. For detailed information regarding the controller roles, refer to the recipe, *Setting the role of a controller's communication channel in Chapter 1, OpenFlow Channel Connection Establishment (Part 2).*

This chapter describes the steps and mechanisms involved in establishing an OpenFlow channel from switch to controller along with the handling of different messages related to OpenFlow channels and controller roles.

Connection setup on TCP and TLS

The switch must be able to establish communication with a controller at a configurable IP address, using either a user-specified transport port or a default transport port. The communication protocol can either be TCP or TLS and the default port for both of these protocols is 6653. The previous versions of OpenFlow used the default port of either 6633 or 976. However IANA has allocated port number 6653 to Open networking foundation (ONF) for OpenFlow protocol.

The connection from switch to controller is identified by the switch Datapath ID and an auxiliary ID. The Auxiliary ID of the main connection is always 0.

Getting started

To establish a communication channel with the controller, the user should configure the IP address, port number (optional) and the transport protocol (TCP/TLS) of the controller. The switch should provide a mechanism to configure these parameters via a standard Command-line Interface (CLI) or use a configuration file or other mechanism.

How to do it...

Based on the configured transport path protocol the switch should follow either the TCP procedure or the TLS procedure explained in this section for establishing connectivity to the controller.
TCP Procedure

Standard TCP/IP socket procedure should be used to establish the OpenFlow communication channel between the switch and controller. The switch should create a TCP socket and try to connect to the configured IP address and port number of the controller using the connect system call provided by the underlying operating system. The procedure in C on Unix-based operating systems is as follows.

```c
struct sockaddr_in controller_address;
memset(&controller_address, '0', sizeof(controller_address));
controller_address.sin_family = AF_INET;
controller_address.sin_addr.s_addr = inet_addr("10.0.0.10");
controller_address.port = htons(6653);
if (socket (AF_INET, SOCK_STREAM, 0) < 0)
{
    printf("\n Error : Could not create socket \n");
    return 1;
}
if ((connect (s, (struct sockaddr*)&controller_address,
            sizeof(controller_address)) < 0))
{
    printf("\n Error : Connect Failed \n");
    return 1;
}
```

TLS Procedure

In order to establish a TLS connection, the switch and controller should authenticate each other mutually, by exchanging certificates which are signed using a private key. The switch must be configured with two certificates, one for authenticating the controller (for example, the controller certificate) and the other for authenticating the controller from the switch (for example, the switch certificate).
For establishing secure a communication channel across the controller and switch using TLS, OpenSSL library can be used. OpenSSL is an open-source implementation of basic cryptographic functions and utilities written in the C programming language and provides complete implementation of the SSL and TLS protocols.

As OpenSSL requires a TCP connection between the client and server, the first step is to create the TCP sockets as mentioned in the TCP procedure section of this recipe. Once the TCP connection is established, the procedure in C on Unix-based operating systems for establishing a secure communication channel using OpenSSL is as follows:

```c
//Register the error strings libssl
SSL_load_error_strings();

// Register the available ciphers and digests
SSL_library_init();

// New context with mode as client and version as SSL 2 or 3
sslContext = SSL_CTX_new(SSLv23_client_method());
if (sslContext == NULL)
{
    printf("\n Error: Could not create SSL context\n");
    return 1;
}

// Create a new SSL struct sslHandle = SSL_new (sslContext);
if (sslHandle == NULL)
{
    printf("\n Error: Could not create SSL handle\n");
    return 1;
}

// Bind the SSL struct to the TCP connection
if (!SSL_set_fd (sslHandle, socket))
{
    printf("\n Error: Could not set socket \n");
    return 1;
}

// Initiate SSL handshake
if (SSL_connect (sslHandle) != 1)
{
    printf("\n Error: Could not connect using SSL\n");
    return 1;
}
```
There's more...

When the connection between the switch and controller is first established, either side of the connection must immediately send the hello message. The procedure to send and receive the hello message is explained in detail in Sending and processing hello message recipe of Chapter 2, Symmetric Messages and Asynchronous Messages.

The OpenFlow specification doesn't mandate any failure handling while establishing the communication channel. However it is recommended for the switch to re-initiate the connection periodically until the connection is successful. If the switch is unable to establish a connection with any of the configured controllers and the switch is a hybrid switch, then the switch can operate in non-openflow mode until the connection to any one of the controllers is successful.

See also

- For more information regarding the procedure to send and receive the OFPT_HELLO message, refer to the Sending and processing hello message recipe of Chapter 2, Symmetric Messages and Asynchronous Messages

Connection setup with multiple controllers

When a switch is configured to connect to more than one controller, then the switch should establish a communication channel with all the configured controllers. A switch may connect to multiple controllers for the following reasons:

- To improve the reliability of the system
- To provide load balancing across controllers depending on the role

When a switch is connected to more than one controller then the controller can take either the master role, slave role or the equal role. For more information regarding the controller role, refer to the recipe, Multiple controllers managing switch with different roles in Chapter 1, OpenFlow Channel Connection Establishment (Part 2).

The switch maintains the role of the controller with respect to the controller's connection, as the switch identifies the controller with the channel ID, which is the combination of the Datapath ID and the Auxiliary ID.
A switch should be connected to only one controller in its master state. However, it may be connected to multiple controllers in an equal state or in the slave state.

**How to do it...**

During the initialization of the OpenFlow switch, it should initiate a connection to all the configured controllers and should maintain the connectivity to all the controllers concurrently. The procedure to establish an OpenFlow communication channel to multiple controllers is similar to that of establishing a connection to a single controller as was explained in the *Connection setup on TCP & TLS* recipe. However, as the switch has to process requests concurrently from multiple controllers, the switch could employ mechanisms to read and process the message from multiple channels. This can be done, either by having different threads or different processes, each handling requests from multiple controllers, or should use some of the constructs provided by the operating system such as the `select()` system call.

**How it works...**

In a steady state, the switch should be able to send asynchronous messages to all controllers through the channels associated with the controller. Similarly, the switch should be able to process the OpenFlow request message from any of the connected controllers. When a switch receives the OpenFlow request message from any one of the controllers, the switch should send the response only to the channel which is associated with that controller.

**There's more...**

The OpenFlow specification doesn't mandate any failure handling while establishing the communication channel. However, it is recommended for the switch to try to re-initiate the connection periodically until the connection is successful.

**See also**

- For more information about establishing the connection to a controller, refer to the previous recipe in this chapter. The next recipe describes how to set the role of the controller in detail.
Setting the role of the communication channel towards a controller

Role-request messages are sent from the controller to set the role of its OpenFlow channel, or query that role. The controller's role in the switch is constantly changed as a result of a request from the controller. The switch cannot change the role of a controller channel on its own.

The controller may send the role request message to communicate its channel role at any time and the switch must remember the role of each controller connection.

Refer to the Setting the role of a controller's communication channel recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2) for more information on when and why the controller sends the role request message to the switch.

How to do it...

The message format used by the controller to send OFPT_ROLE_REQUEST messages is defined in the Setting the role of a controller's communication channel recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2). The switch should use the same message format for sending the reply message back to the controller.

If the role requested from the controller is master or slave, then the switch must validate the generation_id to check for stale messages. This is required because, if the stale messages are processed by the switch, then there is a possibility of one or more controllers and the switch might be out of sync with respect to the role.

Once the switch receives a role request message, it must return an OFPT_ROLE_REPLY message, if there is no error encountered while processing this role request message. The structure of this reply message is exactly same as the OFPT_ROLE_REQUEST message.

The field role should be set with the current role of the controller.

The field generation_id should be set to the current generation_id (the generation_id associated with the last successful role request). If the current generation_id was never set, the generation_id in the reply must be set to the maximum field value (the unsigned equivalent of -1).
If the validation of generation_id fails, the switch must discard the role request message and return an error message with type `OFPET_ROLE_REQUEST_FAILED` and the code as `OFPRRFC_STALE`.

The procedure to handle a role request message from the controller is as follows:

```c
handle_role_request_message (struct ofp_role_request role_request) {
    struct ofp_role_request role_reply;
    /* pseudo-code to validate the generation_id. Here the
     * generation_is_defined and cached_generation_id are global
     * variables */
    if (generation_is_defined && (int64_t)
        (role_request.generation_id - cached_generation_id) < 0)
    {
        send_error_message(OFPET_ROLE_REQUEST_FAILED, OFPRRFC_STALE);
    }
    else
    {
        cached_generation_id = role_request.generation_id;
        generation_is_defined = true;
        /* Here connection is the connection data structure which is
         * maintained by the switch */
        connection.role = role_request.role;
        role_reply.role = role_request.role;
        role_reply.generation_id = cached_generation_id;
        send_openflow_message (connection, role_reply);
    }
}
```

The procedure to send error messages is explained in the Setting the role of a controller's communication channel recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2).

### There's more...

If the requested role is `OFPCR_ROLE_MASTER`, then the switch should change the role of the existing master controller to `OFPCR_ROLE_EQUAL`.

If the requested role is `OFPCR_ROLE_SLAVE` and after successfully setting the role of this communication channel as a slave in the switch, the switch:

- Should not send asynchronous messages.
Should not execute the controller-switch command. For example OFPT_PACKET_OUT, OFPT_FLOW_MOD, OFPT_GROUP_MOD, OFPT_PORT_MOD, OFPT_TABLE_MOD requests, and OFPMP_TABLE_FEATURES multipart requests with a non-empty body must be rejected.

If it receives any controller-switch command then the switch must send the OFPT_ERROR message of the type field and code as OFPET_BAD_REQUEST and OFPBRCC_IS_SLAVE respectively. The procedure to send error messages is explained in the recipe, Setting the role of a controller's communication channel recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2).

See also

For more information about sending the role request message from the controller, refer to the Setting the role of a controller's communication channel recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2)

Establishing an auxiliary connection to the controller

Apart from the main connection, a switch can initiate one or more connections towards the controller to improve its processing performance and exploit the parallelism of most switch implementations. These connections are termed auxiliary connections. For auxiliary connections, the switch should assign the auxiliary ID value as a non-zero value and the Datapath ID the same as that of the main connection towards that controller.

The switch must not initiate an auxiliary connection to a controller until the main connection has been successfully established.

Getting started

To establish an auxiliary connection to the controller, the main connection between the switch and controller should have been successfully established.
How to do it...

The procedure to establish an auxiliary connection to a controller is similar to that of establishing a main connection to the controller which was explained in the first recipe of this chapter. The IP address of the controller used by the switch to establish an auxiliary connection should be the same as that of the main connection. However auxiliary connections are not restricted to use the same transport protocols as those of the main connection. Depending on the configuration, the switch can use TCP, TLS, UDP or DTLS to establish auxiliary connections. When a switch detects that the main connection to a controller is closed or broken, it must close all its auxiliary connections to that controller immediately.

There's more...

The switch must accept any OpenFlow message types and sub-types on any connections. The main connection or an auxiliary connection cannot be restricted to a specific message type or sub-type. However, the processing performance of different connections may be different. The switch may choose to process the different auxiliary connections with different priorities; for example, one auxiliary connection may be dedicated to high priority requests and another may be dedicated to lower priority requests.

The OpenFlow specification provides some guidelines for sending and receiving messages in main and auxiliary connections, as follows:

- All OpenFlow messages which are not Packet in should be sent over the main connection.
- A Mechanism should be provided in the switch to keep track of the packet of the same flow mapped to the same connection, when there are multiple auxiliary connections between the switch and controller.
- An OpenFlow switch which uses an unreliable auxiliary connection should follow recommendations specified in RFC 5405 wherever it is required. RFC 5405 provides guidelines on the usage of UDP protocol such as congestion control, message sizes, reliability, checksums, and middle box traversal, etc.
- If the auxiliary connection from the switch is established in unreliable transport protocols like UDP and DTLS, then only the following message type should be supported to send /receive in an auxiliary connection:
  - OFPT_HELLO
  - OFPT_ERROR
  - OFPT_ECHO_REQUEST
  - OFPT_ECHO_REPLY
Support to send and receive other message types should not be provided

- After establishing the auxiliary connection over unreliable transport protocols, if the switch receives a first message other than OFPT_HELLO then the switch should either:
  - Assume the connection is set up properly and use the version number from that message or it must return an error message of the OFPET_BAD_REQUEST type with the code as OFPBR_BAD_VERSION.

- If the OpenFlow switch receives an error message with the error type OFPET_BAD_REQUEST and the code OFPBR_BAD_VERSION on an unreliable auxiliary connection, then it must either send a new Hello message or terminate the unreliable auxiliary connection.

- If the switch doesn't receive any message on an auxiliary connection after a chosen amount of time lower than 5 seconds, the switch must either send a new Hello message or terminate the unreliable auxiliary connection.

See also

- For more information about establishing the main connection to the controller, refer to the Connection setup on TCP and TLS recipe of Chapter 1, OpenFlow Channel Connection Establishment (Part 2)

Handling a handshake message from the controller

The handshake messages (OFPT_FEATURES_REQUEST / OFPT_FEATURES_REPLY) are used by the controller to fetch the basic capabilities and features supported by the switch. The switch should respond with supported features via an OFPT_FEATURES_REPLY message.

💡 The OFPT_FEATURES_REQUEST message should be sent from the controller once the connection setup is completed.
Getting started

To get the features supported by switch, the communication channel between the switch and controller should have been successfully established.

How to do it...

On reception of OFPT_FEATURES_REQUEST, the switch should prepare an OFPT_FEATURES_REPLY message and send the reply message to the controller.

The OFPT_FEATURES_REQUEST message sent from the controller doesn’t contain any body other than the OpenFlow header which is defined in the OpenFlow Header section of the Appendix.

The format of OFPT_FEATURES_REPLY is as follows:

```c
/* Switch features. */
struct ofp_switch_features {
    struct ofp_header header;
    uint64_t datapath_id; /* Datapath unique ID. The lower 48-bits are for a MAC address, while the upper 16-bits are implementer-defined. */
    uint32_t n_buffers; /* Max packets buffered at once. */
    uint8_t n_tables; /* Number of tables supported by datapath. */
    uint8_t auxiliary_id; /* Identify auxiliary connections */
    uint8_t pad[2]; /* Align to 64-bits. */
    /* Features. */
    uint32_t capabilities; /* Bitmap of support "ofp_capabilities". */
    uint32_t reserved;
};
```

![Flowchart of OFPT_FEATURES_REPLY structure](image)
Let’s see the brief description of the image:

- **datapath_id**: This field should be filled with `datapath_id` of the connection on which the switch receives this request message. Typically, the `datapath_id` is a 64-bit variable wherein the lower 48 bits are assigned from the switch MAC address, while the top 16 bits are assigned based on the implementer’s logic.

- **n_buffers**: This field should be set with the maximum number of packets that can be buffered by the switch while sending packets to the controller using packet-in messages.

- **n_tables**: This field should be set with the number of tables supported by the switch, each of which can have a different set of supported match fields, actions, and number of entries.

- **auxiliary_id**: This should be set to the auxiliary ID of the connection. For the main connection, this field value is zero, while the auxiliary connection has this field value as a non-zero value.

Based on the capability of the switch, it should set the bitmap with the following flag values:

```c
/* Capabilities supported by the datapath. */
enum ofp_capabilities {
  OFPC_FLOW_STATS = 1 << 0,    /* Value 1. Flow statistics. */
  OFPC_TABLE_STATS = 1 << 1,   /* Value 2. Table statistics. */
  OFPC_PORT_STATS = 1 << 2,    /* Value 4. Port statistics. */
  OFPC_GROUP_STATS = 1 << 3,   /* Value 8. Group statistics. */
  OFPC_IP_REASM = 1 << 5,      /* Value 16. Can reassemble IP fragments. */
  OFPC_QUEUE_STATS = 1 << 6,   /* Value 32. Queue statistics. */
  OFPC_PORT_BLOCKED = 1 << 8   /* Value 64. Switch will block looping ports. */
};
```

The procedure to handle the feature request message is as follows:

```c
handle_features_request_message (struct ofp_header features_request)
{
  struct ofp_switch_features features_reply;
  features_reply.datapath_id = htonl(datapath_id);
  features_reply.n_buffers = htonl(buffer_size);
  features_reply.n_tables = supported_no_of_flow_tables;
  features_reply.capabilities = htonl (OFPC_FLOW_STATS | OFPC_TABLE_STATS);
  features_reply.auxiliary_id = 0; /* Assuming main connection */
  send_openflow_message (connection, features_reply);
  /* The send_openflow_message function adds the OF header */
}
See also

- For more information about sending the feature request message from the controller, refer to the Sending a handshake message to the switch recipe in Chapter 1, OpenFlow Channel Connection Establishment (Part 2)

Handling a switch configuration message from the controller

Switch configuration messages are used by the controller to set and get switch configuration parameters.

The Switch should be able to handle OFPT_SET_CONFIG and OFPT_GET_CONFIG messages from the controller and should reply back with OFPT_GET_CONFIG_REPLY messages to the controller.

OFPT_SET_CONFIG and OFPT_GET_CONFIG messages are sent from the controller to the switch whereas the OFPT_GET_CONFIG message is sent from the switch to the controller.

How to do it...

On reception of an OFPT_GET_CONFIG_REQUEST message, the switch should prepare an OFPT_GET_CONFIG_REPLY message and send it to the controller.

The OFPT_GET_CONFIG_REQUEST message doesn't contain a body other than an OpenFlow header which is defined in the OpenFlow Header section of the Appendix.

The OFPT_GET_CONFIG_REPLY message uses the following message format:

```c
/* Switch configuration. */
struct ofp_switch_config {
    struct ofp_header header;
    uint16_t flags;           /* Bitmap of OFPC_* flags. */
    uint16_t miss_send_len;   /* Max bytes of packet that datapath should send to the controller. */
};
```
Chapter 1

The flags take the value as:

```c
def enum ofp_config_flags {
    /* Handling of IP fragments. */
    OFPC_FRAG_NORMAL = 0,  /* No special handling for fragments. */
    OFPC_FRAG_DROP = 1 << 0,  /* Drop fragments. */
    OFPC_FRAG_REASM = 1 << 1,  /* Reassemble (only if
                                 OFPC_IP_REASM set). */
    OFPC_FRAG_MASK = 3,
};
```

The value in `miss_send_len` should be set to the switch's current maximum bytes of data that will be sent to the controller while sending buffered packets. By default this value is 128 bits.

The flag value should be set to the configured value of the IP fragment detail.

On reception of `OFPT_SET_CONFIG`, which has the same message format as that of `OFPT_GET_CONFIG_REPLY`, the switch should set the value of `miss_send_len` and IP fragment related details.

The switch need not send a reply message to the `OFPT_SET_CONFIG` message.

The procedure to handle `OFPT_GET_CONFIG_REQUEST` is as follows:

```c
handle_get_config_message (struct ofp_header get_config_request)
{
    struct ofp_switch_config config_reply;
    features_reply.flags = htons(supported_flags);
    features_reply.miss_send_len = 128;  /* default value */
    send_openflow_message (connection, config_reply);
    /* The send_openflow_message function adds the OF header */
}
```
Connection interruption procedures

The OpenFlow switch should maintain the connectivity to all controllers. If for any reason the switch loses the connection to all configured controllers, then the switch should follow the procedure defined in this section.

How to do it...

If a switch loses its connection with all the connected or configured controllers, then the switch should immediately enter either fail secure mode or fail standalone mode depending upon the switch configuration or implementation. The connection loss can be identified by either OpenFlow echo request timeouts or TLS session timeouts or other disconnection procedures such as TCP socket close, etc.

How it works...

When the switch is in fail secure mode, the switch should drop all the packets and messages destined to controllers. However, all the flow entries programmed by the controller should persist in the switch and expire based on the configured timeout value in fail secure mode.

The fail standalone mode is usually possible only on Hybrid switches. When a switch is in fail standalone mode, the switch should act as a legacy Ethernet switch or router and should process all the packets.

There's more...

Upon connecting to a controller again, the existing flow entries remain intact. The controller can delete all the existing flow entries, if desired. The switch can be configured to start up with the operational mode either as fail secure mode or fail standalone mode until it successfully connects to a controller.

See also

- For more information about sending the switch configuration message from the controller, refer to the Sending a switch configuration message to the switch recipe of Chapter 1, OpenFlow Channel Connection Establishment (Part 2)
Where to buy this book

You can buy OpenFlow Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.