Instituto Politécnico Nacional

Escuela Superior de Cómputo

TAREA NO. 3

Spanning Tree

Redes de Computadora

Profesor: Axel Ernesto Moreno Cervantes

Grupo: 2CM10

Fecha: 11 / Junio /2018

Alumno:

Calva Hernández José Manuel 2017630201

Índice

Introducción

En teoría de grafos, un árbol de expansión, árbol generador o árbol recubridor T de un grafo conexo, no dirigido G es un árbol compuesto por todos los vértices y algunas (quizá todas) de las aristas de G. Informalmente, un árbol de expansión de G es una selección de aristas de G que forman un árbol que cubre todos los vértices. Esto es, cada vértice está en el árbol, pero no hay ciclos. Por otro lado, todos los puentes de G deben estar contenidos en T.

Un árbol de expansión o árbol recubridor de un grafo conexo G puede ser también definido como el mayor conjunto de aristas de G que no contiene ciclos, o como el mínimo conjunto de aristas que conecta todos los vértices.

En ciertos campos de la teoría de grafos es útil encontrar el mínimo árbol de expansión de un grafo ponderado. También se han abordado otros problemas de optimización relacionados con los árboles de expansión, como el máximo árbol de expansión, el máximo árbol que cubre al menos k vértices, el mínimo árbol de expansión con k aristas por vértice como máximo (árbol de expansión de mínimo grado, MDST por sus siglas en inglés), el árbol de expansión con el máximo número de hojas (estrechamente relacionado con el problema del menos conjunto dominante y conexo), el árbol de expansión con el menor número de hojas (relacionado con el problema del camino hamiltoniano), el árbol de expansión de mínimo diámetro o el árbol de expansión de la mínima dilación.

Protocolo STP

STP (del inglés Spanning Tree Protocol) es un protocolo de red de nivel 2 del modelo OSI (capa de enlace de datos). Su función es la de gestionar la presencia de bucles en topologías de red debido a la existencia de enlaces redundantes (necesarios en muchos casos para garantizar la disponibilidad de las conexiones). El protocolo permite a los dispositivos de interconexión activar o desactivar automáticamente los enlaces de conexión, de forma que se garantice la eliminación de bucles. STP es transparente a las estaciones de usuario.

Funcionamiento

El algoritmo transforma una red física con forma de malla, en la que existen bucles, por una red lógica en forma de árbol (libre de bucles). Los puentes se comunican mediante mensajes de configuración llamados Bridge Protocol Data Units (BPDU).

El protocolo establece identificadores por puente y elige el que tiene la prioridad más alta (el número más bajo de prioridad numérica), como el puente raíz (Root Bridge). Este puente raíz establecerá el camino de menor coste para todas las redes; cada puerto tiene un parámetro configurable: el Span path cost. Después, entre todos los puentes que conectan un segmento de red, se elige un puente designado, el de menor coste (en el caso que haya el mismo coste en dos puentes, se elige el que tenga el menor identificador "dirección MAC"), para transmitir las tramas hacia la raíz. En este puente designado, el puerto que conecta con el segmento es el puerto designado y el que ofrece un camino de menor coste hacia la raíz, el puerto raíz. Todos los demás puertos y caminos son bloqueados, esto es en un estado ya estacionario de funcionamiento.

Elección del puente raíz

La primera decisión que toman todos los switches de la red es identificar el puente raíz ya que esto afectará al flujo de tráfico. Cuando un switch se enciende, supone que es el switch raíz y envía las BPDUs que contienen la dirección MAC de sí mismo tanto en el BID raíz como emisor. El BID es el Bridge IDentifier: Bridge Priority + Bridge Mac Address. El Bridge Priority es un valor configurable que por defecto está asignado en 32768. El Bridge Mac Address es la dirección MAC (única) del Puente.

Cada switch reemplaza los BID de raíz más alta por BID de raíz más baja en las BPDU que se envían. Todos los switches reciben las BPDU y determinan que el switch que cuyo valor de BID raíz es el más bajo será el puente raíz. En caso de empate, el switch root sería el que menor MAC tuviera. El administrador de red puede establecer la prioridad de switch en un valor más pequeño que el del valor por defecto (32768), el nuevo valor debe ser múltiplo de 4096, lo que hace que el BID sea más pequeño. Esto sólo se debe implementar cuando se tiene un conocimiento profundo del flujo de tráfico en la red

Elección de los puertos raíz

Una vez elegido el puente raíz hay que calcular el puerto raíz para los otros puentes que no son raíz. El procedimiento a seguir para cada puente es el mismo: entre todos los puertos del puente, se escoge como puerto raíz el puerto que tenga el menor costo hasta el puente raíz. En el caso de que haya dos o más puertos con el mismo costo hacia el puente raíz, se utiliza la dirección MAC que tenga menor valor para calcular el costo y establecer el puerto raíz.

Elección de los puertos designados

Una vez elegido el puente raíz y los puertos raíz de los otros puentes pasamos a calcular los puertos designados de cada segmento de red. En cada enlace que exista entre dos switches habrá un puerto designado, el cual será el puerto del switch que tenga un menor coste para llegar al puente raíz, este coste administrativo será un valor que estará relacionado al tipo de enlace que exista en el puerto (Ethernet, FastEthernet, GigabitEthernet). Cada

tipo de enlace tendrá un coste administrativo distinto, siendo de un coste menor el puerto con una mayor velocidad. Si hubiese empate entre los costes administrativos que tienen los dos switches para llegar al root bridge, entonces se elegirá como Designated Port, el puerto del switch que tenga un menor Bridge ID (BID).

Puertos bloqueados

Aquellos puertos que no sean elegidos como raíz ni como designados deben bloquearse. Estos puertos evitan los lazos.

Mantenimiento del Spanning Tree

El cambio en la topología puede ocurrir de dos formas:

- El puerto se desactiva o se bloquea
- El puerto pasa de estar bloqueado o desactivado a activado

Cuando se detecta un cambio el switch notifica al puente raíz dicho cambio y entonces el puente raíz envía por broadcast dicho cambio. Para ello, se introduce una BPDU especial denominada notificación de cambio en la topología (TCN). Cuando un switch necesita avisar acerca de un cambio en la topología, comienza a enviar TCN en su puerto raíz. La TCN es una BPDU muy simple que no contiene información y se envía durante el intervalo de tiempo de saludo. El switch que recibe la TCN se denomina puente designado y realiza el acuse de recibo mediante el envío inmediato de una BPDU normal con el bit de acuse de recibo de cambio en la topología (TCA). Este intercambio continúa hasta que el puente raíz responde.

Unidades de datos del protocolo puente

Las reglas anteriores describen una forma de determinar qué árbol de expansión será calculado por el algoritmo, pero las reglas como están escritas requieren el conocimiento de toda la red. Los puentes tienen que determinar el puente raíz y calcular las funciones de los puertos (de raíz, designados o bloqueados) con sólo la información que tienen. Para asegurarse de que cada puente tiene suficiente información, los puentes utilizan tramas de datos especiales llamados Unidades de Datos de Protocolo Puente (BPDU) para intercambiar información acerca de los identificadores de puentes y costes de la ruta raíz.

Un puente envía una trama de BPDU usando la dirección MAC única del propio puerto como dirección de origen, y una dirección de destino la dirección multicast STP 01:80:C2:00:00:00.

Hay dos tipos de BPDU en la especificación original STP (The Rapid Spanning Tree (RSTP)) utiliza una BPDU RSTP específica):

- Configuración de BPDU (CBPDU), utilizado para el cálculo de árbol de expansión
- Notificación de cambio de topología (TCN) BPDU, utilizado para anunciar los cambios en la topología de red

BPDU se intercambian regularmente (cada 2 segundos de forma predeterminada) y permiten a los switches, realizar un seguimiento de cambios en la red y para iniciar y detener el reenvío de puertos según sea necesario.

Cuando un dispositivo se conecta primero a un puerto de un switch, no iniciará de inmediato a enviar datos. En su lugar, pasará por una serie de estados mientras procesa BPDU y determina la topología de la red. Cuando un host se une, como un ordenador, impresora o un servidor, el puerto siempre va a entrar en el estado de envío, aunque con un retraso de unos 30 segundos, mientras que pasa a través de la escucha y de los estados de aprendizaje (ver más abajo). El tiempo pasado en los estados de escucha y aprendizaje está determinado por un valor conocido como el retardo de envío (por defecto 15 segundos y fijado por el puente de la raíz).

Sin embargo, si en lugar otro switch está conectado, el puerto puede permanecer en modo de bloqueo, si se determina que causaría un bucle en la red. Notificación de cambio de topología (TCN) BPDU se utilizan para informar a otros conmutadores de puerto cambia. TCN se inyectan en la red mediante un switch que no sea root y propagado a la raíz. Tras la recepción de la TCN, el switch de la raíz establecerá una bandera de cambio de topología en su BPDU normal. Esta flag se propaga a todos los otros switches para instruirlos a envejecer rápidamente sus entradas de la tabla de reenvío.

Estado de los puertos

Los estados en los que puede estar un puerto son los siguientes:

- Bloqueo: En este estado se pueden recibir BPDU's pero no las enviará. Las tramas de datos se descartan y no se actualizan las tablas de direcciones MAC (mac-address-table). Los switch comienzan en este estado ya que si realizan envíos (forwarding) podrían estar generando un loop o bucle.
- Escucha: A este estado se llega desde Bloqueo. En este estado, los switches determinan si existe alguna otra ruta hacia el puente raíz. En el caso que la nueva ruta tenga un coste mayor, se vuelve al estado de Bloqueo. Las tramas de datos se descartan y no se actualiza la tabla de direcciones MAC (mac-address-table). Se procesan las BPDU.
- Aprendizaje: A este estado se llega desde Escucha. Las tramas de datos se descartan, pero ya se actualizan las tablas de direcciones MAC (aquí es donde se aprenden por primera vez). Se procesan las BPDU.
- Envío: A este estado se llega desde Aprendizaje, en este estado el puerto puede enviar y recibir datos. Las tramas de datos se envían y se actualizan las tablas de direcciones MAC (mac-address-table). Se procesan las BPDU.
- Desactivado: A este estado se llega desde cualquier otro. Se produce cuando un administrador deshabilita el puerto o éste falla. No se procesan las BPDU.

Ejemplo Implementación (Python)

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.



import datetime
import logging


from ryu.base import app_manager
from ryu.controller import event
from ryu.controller import handler
from ryu.controller import ofp_event
from ryu.controller.handler import set_ev_cls
from ryu.exception import RyuException
from ryu.exception import OFPUnknownVersion
from ryu.lib import hub
from ryu.lib import mac
from ryu.lib.dpid import dpid_to_str
from ryu.lib.packet import bpdu
from ryu.lib.packet import ethernet
from ryu.lib.packet import llc
from ryu.lib.packet import packet
from ryu.ofproto import ofproto_v1_0
from ryu.ofproto import ofproto_v1_2
from ryu.ofproto import ofproto_v1_3



MAX_PORT_NO = 0xfff


# for OpenFlow 1.2/1.3
BPDU_PKT_IN_PRIORITY = 0xffff
```

```
NO_PKT_IN_PRIORITY = 0xfffe



# Result of compared config BPDU priority.
SUPERIOR = -1
REPEATED = 0
INFERIOR = 1


# Port role
DESIGNATED_PORT = 0  # The port which sends BPDU.
ROOT_PORT = 1  # The port which receives BPDU from a root bridge.
NON_DESIGNATED_PORT = 2  # The port which blocked.


""" How to decide the port roles.
     Root bridge:
       a bridge has smallest bridge ID is chosen as a root.
       it sends original config BPDU.
     Non Root bridge:
       forwards config BPDU received from the root bridge.
                   +-----------------------+
                   |      Root bridge      |
                   +-----------------------+
                     (D)               (D)
                      |                 |
                      |                 |
                     (R)               (R)
        +-----------------+          +-----------------+
        | Non Root bridge |(D)---(ND)| Non Root bridge |
        +-----------------+          +-----------------+
     ROOT_PORT(R):
       the nearest port to a root bridge of the bridge.
       it is determined by the cost of the path, etc.
     DESIGNATED_PORT(D):
       the port of the side near the root bridge of each link.
       it is determined by the cost of the path, etc.
     NON_DESIGNATED_PORT(ND):
       the port other than a ROOT_PORT and DESIGNATED_PORT.
"""
```

```
# Port state
#  DISABLE: Administratively down or link down by an obstacle.
#  BLOCK  : Not part of spanning tree.
#  LISTEN : Not learning or relaying frames.
#  LEARN  : Learning but not relaying frames.
#  FORWARD: Learning and relaying frames.
PORT_STATE_DISABLE = 0
PORT_STATE_BLOCK = 1
PORT_STATE_LISTEN = 2
PORT_STATE_LEARN = 3
PORT_STATE_FORWARD = 4


# for OpenFlow 1.0
PORT_CONFIG_V1_0 = {PORT_STATE_DISABLE: (ofproto_v1_0.OFPPC_NO_RECV_STP
                                         | ofproto_v1_0.OFPPC_NO_RECV
                                         | ofproto_v1_0.OFPPC_NO_FLOOD
                                         | ofproto_v1_0.OFPPC_NO_FWD),
                    PORT_STATE_BLOCK: (ofproto_v1_0.OFPPC_NO_RECV
                                       | ofproto_v1_0.OFPPC_NO_FLOOD
                                       | ofproto_v1_0.OFPPC_NO_FWD),
                    PORT_STATE_LISTEN: (ofproto_v1_0.OFPPC_NO_RECV
                                        | ofproto_v1_0.OFPPC_NO_FLOOD),
                    PORT_STATE_LEARN: ofproto_v1_0.OFPPC_NO_FLOOD,
                    PORT_STATE_FORWARD: 0}


# for OpenFlow 1.2
PORT_CONFIG_V1_2 = {PORT_STATE_DISABLE: (ofproto_v1_2.OFPPC_NO_RECV
                                         | ofproto_v1_2.OFPPC_NO_FWD),
                    PORT_STATE_BLOCK: (ofproto_v1_2.OFPPC_NO_FWD
                                       | ofproto_v1_2.OFPPC_NO_PACKET_IN),
                    PORT_STATE_LISTEN: ofproto_v1_2.OFPPC_NO_PACKET_IN,
                    PORT_STATE_LEARN: ofproto_v1_2.OFPPC_NO_PACKET_IN,
                    PORT_STATE_FORWARD: 0}


# for OpenFlow 1.3
PORT_CONFIG_V1_3 = {PORT_STATE_DISABLE: (ofproto_v1_3.OFPPC_NO_RECV
                                         | ofproto_v1_3.OFPPC_NO_FWD),
```

```
                    PORT_STATE_BLOCK: (ofproto_v1_3.OFPPC_NO_FWD
                                       | ofproto_v1_3.OFPPC_NO_PACKET_IN),
                    PORT_STATE_LISTEN: ofproto_v1_3.OFPPC_NO_PACKET_IN,
                    PORT_STATE_LEARN: ofproto_v1_3.OFPPC_NO_PACKET_IN,
                    PORT_STATE_FORWARD: 0}


""" Port state machine
    +------------------------<--------------------------+
    |                                                   |*2
    +--> [BLOCK] -----+--> [LISTEN] ----> [LEARN] ------+----> [FORWARD]
                  *3  |        |    15sec    |    15sec   *1       |
                      |        |*3           |*3                   |*3
                      +----<---+------<------+----------<----------+
          *1 if port role == DESIGNATED_PORT or ROOT_PORT
          *2 if port role == NON_DESIGNATED_PORT
          *3 re-calculation of Spanning tree occurred.
      When bridge has started, each port state is set to [LISTEN]
       except port configuration is disable.
      If port configuration is disable or link down occurred,
       the port state is set to [DISABLE]
"""



# Throw this event when network topology is changed.
# Flush filtering database, when you receive this event.
class EventTopologyChange(event.EventBase):
    def __init__(self, dp):
        super(EventTopologyChange, self).__init__()
        self.dp = dp



# Throw this event when port status is changed.
class EventPortStateChange(event.EventBase):
    def __init__(self, dp, port):
        super(EventPortStateChange, self).__init__()
        self.dp = dp
        self.port_no = port.ofport.port_no
        self.port_state = port.state
```

```
# Event for receive packet in message except BPDU packet.
class EventPacketIn(event.EventBase):
    def __init__(self, msg):
        super(EventPacketIn, self).__init__()
        self.msg = msg



# For Python3 compatibility
# Note: The following is the official workaround for cmp() in Python2.
# https://docs.python.org/3.0/whatsnew/3.0.html#ordering-comparisons
def cmp(a, b):
    return (a > b) - (a < b)



class Stp(app_manager.RyuApp):
    """ STP(spanning tree) library. """

    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION,
                    ofproto_v1_2.OFP_VERSION,
                    ofproto_v1_3.OFP_VERSION]

    def __init__(self):
        super(Stp, self).__init__()
        self.name = 'stplib'
        self._set_logger()
        self.config = {}
        self.bridge_list = {}

    def close(self):
        for dpid in self.bridge_list:
            self._unregister_bridge(dpid)
```

```
def _set_logger(self):
    self.logger.propagate = False
    hdlr = logging.StreamHandler()
    fmt_str = '[STP][%(levelname)s] dpid=%(dpid)s: %(message)s'
    hdlr.setFormatter(logging.Formatter(fmt_str))
    self.logger.addHandler(hdlr)


def set_config(self, config):
    """ Use this API if you want to set up configuration
        of each bridge and ports.
       Set configuration with 'config' parameter as follows.
        config = {<dpid>: {'bridge': {'priority': <value>,
                                      'sys_ext_id': <value>,
                                      'max_age': <value>,
                                      'hello_time': <value>,
                                      'fwd_delay': <value>}
                           'ports': {<port_no>: {'priority': <value>,
                                                 'path_cost': <value>,
                                                 'enable': <True/False>},
                                     <port_no>: {...},,,}}
                  <dpid>: {...},
                  <dpid>: {...},,,}
         NOTE: You may omit each field.
               If omitted, a default value is set up.
              It becomes effective when a bridge starts.
         Default values:
         ---------------------------------------------------
         | bridge | priority   | bpdu.DEFAULT_BRIDGE_PRIORITY |
         |        | sys_ext_id | 0                            |
         |        | max_age    | bpdu.DEFAULT_MAX_AGE         |
         |        | hello_time | bpdu.DEFAULT_HELLO_TIME      |
         |        | fwd_delay  | bpdu.DEFAULT_FORWARD_DELAY   |
         |--------|------------|------------------------------|
         | port   | priority   | bpdu.DEFAULT_PORT_PRIORITY   |
         |        | path_cost  | (Set up automatically        |
         |        |            |   according to link speed.)  |
         |        | enable     | True                         |
         ---------------------------------------------------
    """
    assert isinstance(config, dict)
    self.config = config
```

```
@set_ev_cls(ofp_event.EventOFPStateChange,
            [handler.MAIN_DISPATCHER, handler.DEAD_DISPATCHER])
def dispacher_change(self, ev):
    assert ev.datapath is not None
    if ev.state == handler.MAIN_DISPATCHER:
        self._register_bridge(ev.datapath)
    elif ev.state == handler.DEAD_DISPATCHER:
        self._unregister_bridge(ev.datapath.id)


def _register_bridge(self, dp):
    self._unregister_bridge(dp.id)


    dpid_str = {'dpid': dpid_to_str(dp.id)}
    self.logger.info('Join as stp bridge.', extra=dpid_str)
    try:
        bridge = Bridge(dp, self.logger,
                        self.config.get(dp.id, {}),
                        self.send_event_to_observers)
    except OFPUnknownVersion as message:
        self.logger.error(str(message), extra=dpid_str)
        return


    self.bridge_list[dp.id] = bridge


def _unregister_bridge(self, dp_id):
    if dp_id in self.bridge_list:
        self.bridge_list[dp_id].delete()
        del self.bridge_list[dp_id]
        self.logger.info('Leave stp bridge.',
                         extra={'dpid': dpid_to_str(dp_id)})


@set_ev_cls(ofp_event.EventOFPPacketIn, handler.MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    if ev.msg.datapath.id in self.bridge_list:
        bridge = self.bridge_list[ev.msg.datapath.id]
        bridge.packet_in_handler(ev.msg)
```

```
@set_ev_cls(ofp_event.EventOFPPortStatus, handler.MAIN_DISPATCHER)
def port_status_handler(self, ev):
    dp = ev.msg.datapath
    dpid_str = {'dpid': dpid_to_str(dp.id)}
    port = ev.msg.desc
    reason = ev.msg.reason
    link_down_flg = port.state & 0b1


    if dp.id in self.bridge_list:
        bridge = self.bridge_list[dp.id]


        if reason is dp.ofproto.OFPPR_ADD:
            self.logger.info('[port=%d] Port add.',
                             port.port_no, extra=dpid_str)
            bridge.port_add(port)
        elif reason is dp.ofproto.OFPPR_DELETE:
            self.logger.info('[port=%d] Port delete.',
                             port.port_no, extra=dpid_str)
            bridge.port_delete(port)
        else:
            assert reason is dp.ofproto.OFPPR_MODIFY
            if bridge.ports_state[port.port_no] == port.state:
                # Do nothing
                self.logger.debug('[port=%d] Link status not changed.',
                                  port.port_no, extra=dpid_str)
                return
            if link_down_flg:
                self.logger.info('[port=%d] Link down.',
                                 port.port_no, extra=dpid_str)
                bridge.link_down(port)
            else:
                self.logger.info('[port=%d] Link up.',
                                 port.port_no, extra=dpid_str)
                bridge.link_up(port)


@staticmethod
def compare_root_path(path_cost1, path_cost2, bridge_id1, bridge_id2,
                      port_id1, port_id2):
```

```
    """ Decide the port of the side near a root bridge.
        It is compared by the following priorities.
         1. root path cost
         2. designated bridge ID value
         3. designated port ID value """
    result = Stp._cmp_value(path_cost1, path_cost2)
    if not result:
        result = Stp._cmp_value(bridge_id1, bridge_id2)
        if not result:
            result = Stp._cmp_value(port_id1, port_id2)
    return result


@staticmethod
def compare_bpdu_info(my_priority, my_times, rcv_priority, rcv_times):
    """ Check received BPDU is superior to currently held BPDU
         by the following comparison.
         - root bridge ID value
         - root path cost
         - designated bridge ID value
         - designated port ID value
         - times """
    if my_priority is None:
        result = SUPERIOR
    else:
        result = Stp._cmp_value(rcv_priority.root_id.value,
                                my_priority.root_id.value)
        if not result:
            result = Stp.compare_root_path(
                rcv_priority.root_path_cost,
                my_priority.root_path_cost,
                rcv_priority.designated_bridge_id.value,
                my_priority.designated_bridge_id.value,
                rcv_priority.designated_port_id.value,
                my_priority.designated_port_id.value)
            if not result:
                result1 = Stp._cmp_value(
                    rcv_priority.designated_bridge_id.value,
                    mac.haddr_to_int(
                        my_priority.designated_bridge_id.mac_addr))
                result2 = Stp._cmp_value(
                    rcv_priority.designated_port_id.value,
                    my_priority.designated_port_id.port_no)
```

```
                    if not result1 and not result2:
                        result = SUPERIOR
                    else:
                        result = Stp._cmp_obj(rcv_times, my_times)
        return result


    @staticmethod
    def _cmp_value(value1, value2):
        result = cmp(value1, value2)
        if result < 0:
            return SUPERIOR
        elif result == 0:
            return REPEATED
        else:
            return INFERIOR


    @staticmethod
    def _cmp_obj(obj1, obj2):
        for key in obj1.__dict__.keys():
            if (not hasattr(obj2, key)
                    or getattr(obj1, key) != getattr(obj2, key)):
                return SUPERIOR
        return REPEATED




class Bridge(object):
    _DEFAULT_VALUE = {'priority': bpdu.DEFAULT_BRIDGE_PRIORITY,
                      'sys_ext_id': 0,
                      'max_age': bpdu.DEFAULT_MAX_AGE,
                      'hello_time': bpdu.DEFAULT_HELLO_TIME,
                      'fwd_delay': bpdu.DEFAULT_FORWARD_DELAY}


    def __init__(self, dp, logger, config, send_ev_func):
        super(Bridge, self).__init__()
        self.dp = dp
        self.logger = logger
        self.dpid_str = {'dpid': dpid_to_str(dp.id)}
        self.send_event = send_ev_func
```

```
        # Bridge data
        bridge_conf = config.get('bridge', {})
        values = self._DEFAULT_VALUE
        for key, value in bridge_conf.items():
            values[key] = value
        system_id = list(dp.ports.values())[0].hw_addr


        self.bridge_id = BridgeId(values['priority'],
                                  values['sys_ext_id'],
                                  system_id)
        self.bridge_times = Times(0,  # message_age
                                  values['max_age'],
                                  values['hello_time'],
                                  values['fwd_delay'])
        # Root bridge data
        self.root_priority = Priority(self.bridge_id, 0, None, None)
        self.root_times = self.bridge_times
        # Ports
        self.ports = {}
        self.ports_state = {}
        self.ports_conf = config.get('ports', {})
        for ofport in dp.ports.values():
            self.port_add(ofport)


        # Install BPDU PacketIn flow. (OpenFlow 1.2/1.3)
        if dp.ofproto == ofproto_v1_2 or dp.ofproto == ofproto_v1_3:
            ofctl = OfCtl_v1_2later(self.dp)
            ofctl.add_bpdu_pkt_in_flow()


    @property
    def is_root_bridge(self):
        return bool(self.bridge_id.value == self.root_priority.root_id.value)


    def delete(self):
        for port in self.ports.values():
            port.delete()
```

```
def port_add(self, ofport):
    if ofport.port_no <= MAX_PORT_NO:
        port_conf = self.ports_conf.get(ofport.port_no, {})
        self.ports[ofport.port_no] = Port(self.dp, self.logger,
                                          port_conf, self.send_event,
                                          self.recalculate_spanning_tree,
                                          self.topology_change_notify,
                                          self.bridge_id,
                                          self.bridge_times,
                                          ofport)
        self.ports_state[ofport.port_no] = ofport.state


def port_delete(self, ofp_port):
    self.link_down(ofp_port)
    self.ports[ofp_port.port_no].delete()
    del self.ports[ofp_port.port_no]
    del self.ports_state[ofp_port.port_no]


def link_up(self, ofp_port):
    port = self.ports[ofp_port.port_no]
    port.up(DESIGNATED_PORT, self.root_priority, self.root_times)
    self.ports_state[ofp_port.port_no] = ofp_port.state


def link_down(self, ofp_port):
    """ DESIGNATED_PORT/NON_DESIGNATED_PORT: change status to DISABLE.
        ROOT_PORT: change status to DISABLE and recalculate STP. """
    port = self.ports[ofp_port.port_no]
    init_stp_flg = bool(port.role is ROOT_PORT)


    port.down(PORT_STATE_DISABLE, msg_init=True)
    self.ports_state[ofp_port.port_no] = ofp_port.state
    if init_stp_flg:
        self.recalculate_spanning_tree()


def packet_in_handler(self, msg):
    dp = msg.datapath
    if dp.ofproto == ofproto_v1_0:
```

```
    in_port_no = msg.in_port
else:
    assert dp.ofproto == ofproto_v1_2 or dp.ofproto == ofproto_v1_3
    in_port_no = None
    for match_field in msg.match.fields:
        if match_field.header == dp.ofproto.OXM_OF_IN_PORT:
            in_port_no = match_field.value
            break
if in_port_no not in self.ports:
    return


in_port = self.ports[in_port_no]
if in_port.state == PORT_STATE_DISABLE:
    return


pkt = packet.Packet(msg.data)
if bpdu.ConfigurationBPDUs in pkt:
    # Received Configuration BPDU.
    # - If received superior BPDU:
    #    Re-calculates spanning tree.
    # - If received Topology Change BPDU:
    #    Throws EventTopologyChange.
    #    Forwards Topology Change BPDU.
    (bpdu_pkt, ) = pkt.get_protocols(bpdu.ConfigurationBPDUs)
    if bpdu_pkt.message_age > bpdu_pkt.max_age:
        log_msg = 'Drop BPDU packet which message_age exceeded.'
        self.logger.debug(log_msg, extra=self.dpid_str)
        return


    rcv_info, rcv_tc = in_port.rcv_config_bpdu(bpdu_pkt)


    if rcv_info is SUPERIOR:
        self.logger.info('[port=%d] Receive superior BPDU.',
                         in_port_no, extra=self.dpid_str)
        self.recalculate_spanning_tree(init=False)


    elif rcv_tc:
        self.send_event(EventTopologyChange(self.dp))
```

```
            if in_port.role is ROOT_PORT:
                self._forward_tc_bpdu(rcv_tc)

        elif bpdu.TopologyChangeNotificationBPDUs in pkt:
            # Received Topology Change Notification BPDU.
            # Send Topology Change Ack BPDU and throws EventTopologyChange.
            # - Root bridge:
            #    Sends Topology Change BPDU from all port.
            # - Non root bridge:
            #    Sends Topology Change Notification BPDU to root bridge.
            in_port.transmit_ack_bpdu()
            self.topology_change_notify(None)

        elif bpdu.RstBPDUs in pkt:
            # Received Rst BPDU.
            # TODO: RSTP
            pass

        else:
            # Received non BPDU packet.
            # Throws EventPacketIn.
            self.send_event(EventPacketIn(msg))

    def recalculate_spanning_tree(self, init=True):
        """ Re-calculation of spanning tree. """
        # All port down.
        for port in self.ports.values():
            if port.state is not PORT_STATE_DISABLE:
                port.down(PORT_STATE_BLOCK, msg_init=init)

        # Send topology change event.
        if init:
            self.send_event(EventTopologyChange(self.dp))

        # Update tree roles.
```

```
        port_roles = {}
        self.root_priority = Priority(self.bridge_id, 0, None, None)
        self.root_times = self.bridge_times

        if init:
            self.logger.info('Root bridge.', extra=self.dpid_str)
            for port_no in self.ports:
                port_roles[port_no] = DESIGNATED_PORT
        else:
            (port_roles,
             self.root_priority,
             self.root_times) = self._spanning_tree_algorithm()

        # All port up.
        for port_no, role in port_roles.items():
            if self.ports[port_no].state is not PORT_STATE_DISABLE:
                self.ports[port_no].up(role, self.root_priority,
                                       self.root_times)

    def _spanning_tree_algorithm(self):
        """ Update tree roles.
             - Root bridge:
                all port is DESIGNATED_PORT.
             - Non root bridge:
                select one ROOT_PORT and some DESIGNATED_PORT,
                and the other port is set to NON_DESIGNATED_PORT."""
        port_roles = {}

        root_port = self._select_root_port()

        if root_port is None:
            # My bridge is a root bridge.
            self.logger.info('Root bridge.', extra=self.dpid_str)
            root_priority = self.root_priority
            root_times = self.root_times

            for port_no in self.ports:
```

```
                if self.ports[port_no].state is not PORT_STATE_DISABLE:
                    port_roles[port_no] = DESIGNATED_PORT
        else:
            # Other bridge is a root bridge.
            self.logger.info('Non root bridge.', extra=self.dpid_str)
            root_priority = root_port.designated_priority
            root_times = root_port.designated_times

            port_roles[root_port.ofport.port_no] = ROOT_PORT

            d_ports = self._select_designated_port(root_port)
            for port_no in d_ports:
                port_roles[port_no] = DESIGNATED_PORT

            for port in self.ports.values():
                if port.state is not PORT_STATE_DISABLE:
                    port_roles.setdefault(port.ofport.port_no,
                                          NON_DESIGNATED_PORT)

        return port_roles, root_priority, root_times

    def _select_root_port(self):
        """ ROOT_PORT is the nearest port to a root bridge.
            It is determined by the cost of path, etc. """
        root_port = None

        for port in self.ports.values():
            root_msg = (self.root_priority if root_port is None
                        else root_port.designated_priority)
            port_msg = port.designated_priority
            if port.state is PORT_STATE_DISABLE or port_msg is None:
                continue
            if root_msg.root_id.value > port_msg.root_id.value:
                result = SUPERIOR
            elif root_msg.root_id.value == port_msg.root_id.value:
                if root_msg.designated_bridge_id is None:
                    result = INFERIOR
```

```
            else:
                result = Stp.compare_root_path(
                    port_msg.root_path_cost,
                    root_msg.root_path_cost,
                    port_msg.designated_bridge_id.value,
                    root_msg.designated_bridge_id.value,
                    port_msg.designated_port_id.value,
                    root_msg.designated_port_id.value)
        else:
            result = INFERIOR


        if result is SUPERIOR:
            root_port = port


    return root_port


def _select_designated_port(self, root_port):
    """ DESIGNATED_PORT is a port of the side near the root bridge
        of each link. It is determined by the cost of each path, etc
        same as ROOT_PORT. """
    d_ports = []
    root_msg = root_port.designated_priority


    for port in self.ports.values():
        port_msg = port.designated_priority
        if (port.state is PORT_STATE_DISABLE
                or port.ofport.port_no == root_port.ofport.port_no):
            continue
        if (port_msg is None or
                (port_msg.root_id.value != root_msg.root_id.value)):
            d_ports.append(port.ofport.port_no)
        else:
            result = Stp.compare_root_path(
                root_msg.root_path_cost,
                port_msg.root_path_cost - port.path_cost,
                self.bridge_id.value,
                port_msg.designated_bridge_id.value,
                port.port_id.value,
                port_msg.designated_port_id.value)
```

```
            if result is SUPERIOR:
                d_ports.append(port.ofport.port_no)

    return d_ports

def topology_change_notify(self, port_state):
    notice = False
    if port_state is PORT_STATE_FORWARD:
        for port in self.ports.values():
            if port.role is DESIGNATED_PORT:
                notice = True
                break
    else:
        notice = True

    if notice:
        self.send_event(EventTopologyChange(self.dp))
        if self.is_root_bridge:
            self._transmit_tc_bpdu()
        else:
            self._transmit_tcn_bpdu()

def _transmit_tc_bpdu(self):
    for port in self.ports.values():
        port.transmit_tc_bpdu()

def _transmit_tcn_bpdu(self):
    root_port = None
    for port in self.ports.values():
        if port.role is ROOT_PORT:
            root_port = port
            break
    if root_port:
        root_port.transmit_tcn_bpdu()

def _forward_tc_bpdu(self, fwd_flg):
    for port in self.ports.values():
```

```
            port.send_tc_flg = fwd_flg



class Port(object):
    _DEFAULT_VALUE = {'priority': bpdu.DEFAULT_PORT_PRIORITY,
                      'path_cost': bpdu.PORT_PATH_COST_10MB,
                      'enable': True}


    def __init__(self, dp, logger, config, send_ev_func, timeout_func,
                 topology_change_func, bridge_id, bridge_times, ofport):
        super(Port, self).__init__()
        self.dp = dp
        self.logger = logger
        self.dpid_str = {'dpid': dpid_to_str(dp.id)}
        self.config_enable = config.get('enable',
                                        self._DEFAULT_VALUE['enable'])
        self.send_event = send_ev_func
        self.wait_bpdu_timeout = timeout_func
        self.topology_change_notify = topology_change_func
        self.ofctl = (OfCtl_v1_0(dp) if dp.ofproto == ofproto_v1_0
                      else OfCtl_v1_2later(dp))


        # Bridge data
        self.bridge_id = bridge_id
        # Root bridge data
        self.port_priority = None
        self.port_times = None
        # ofproto_v1_X_parser.OFPPhyPort data
        self.ofport = ofport
        # Port data
        values = self._DEFAULT_VALUE
        path_costs = {dp.ofproto.OFPPF_10MB_HD: bpdu.PORT_PATH_COST_10MB,
                      dp.ofproto.OFPPF_10MB_FD: bpdu.PORT_PATH_COST_10MB,
                      dp.ofproto.OFPPF_100MB_HD: bpdu.PORT_PATH_COST_100MB,
                      dp.ofproto.OFPPF_100MB_FD: bpdu.PORT_PATH_COST_100MB,
                      dp.ofproto.OFPPF_1GB_HD: bpdu.PORT_PATH_COST_1GB,
                      dp.ofproto.OFPPF_1GB_FD: bpdu.PORT_PATH_COST_1GB,
                      dp.ofproto.OFPPF_10GB_FD: bpdu.PORT_PATH_COST_10GB}
        for rate in sorted(path_costs, reverse=True):
```

```
            if ofport.curr & rate:
                values['path_cost'] = path_costs[rate]
                break
        for key, value in values.items():
            values[key] = value
        self.port_id = PortId(values['priority'], ofport.port_no)
        self.path_cost = values['path_cost']
        self.state = (None if self.config_enable else PORT_STATE_DISABLE)
        self.role = None
        # Receive BPDU data
        self.designated_priority = None
        self.designated_times = None
        # BPDU handling threads
        self.send_bpdu_thread = PortThread(self._transmit_bpdu)
        self.wait_bpdu_thread = PortThread(self._wait_bpdu_timer)
        self.send_tc_flg = None
        self.send_tc_timer = None
        self.send_tcn_flg = None
        self.wait_timer_event = None
        # State machine thread
        self.state_machine = PortThread(self._state_machine)
        self.state_event = None


        self.up(DESIGNATED_PORT,
                Priority(bridge_id, 0, None, None),
                bridge_times)


        self.state_machine.start()
        self.logger.debug('[port=%d] Start port state machine.',
                          self.ofport.port_no, extra=self.dpid_str)


    def delete(self):
        self.state_machine.stop()
        self.send_bpdu_thread.stop()
        self.wait_bpdu_thread.stop()
        if self.state_event is not None:
            self.state_event.set()
            self.state_event = None
        if self.wait_timer_event is not None:
            self.wait_timer_event.set()
```

```
            self.wait_timer_event = None
        self.logger.debug('[port=%d] Stop port threads.',
                          self.ofport.port_no, extra=self.dpid_str)


    def up(self, role, root_priority, root_times):
        """ A port is started in the state of LISTEN.  """
        self.port_priority = root_priority
        self.port_times = root_times


        state = (PORT_STATE_LISTEN if self.config_enable
                 else PORT_STATE_DISABLE)
        self._change_role(role)
        self._change_status(state)


    def down(self, state, msg_init=False):
        """ A port will be in the state of DISABLE or BLOCK,
             and be stopped.  """
        assert (state is PORT_STATE_DISABLE
                or state is PORT_STATE_BLOCK)
        if not self.config_enable:
            return


        if msg_init:
            self.designated_priority = None
            self.designated_times = None


        self._change_role(DESIGNATED_PORT)
        self._change_status(state)


    def _state_machine(self):
        """ Port state machine.
             Change next status when timer is exceeded
             or _change_status() method is called."""
        role_str = {ROOT_PORT: 'ROOT_PORT          ',
                    DESIGNATED_PORT: 'DESIGNATED_PORT    ',
                    NON_DESIGNATED_PORT: 'NON_DESIGNATED_PORT'}
        state_str = {PORT_STATE_DISABLE: 'DISABLE',
```

```
                PORT_STATE_BLOCK: 'BLOCK',
                PORT_STATE_LISTEN: 'LISTEN',
                PORT_STATE_LEARN: 'LEARN',
                PORT_STATE_FORWARD: 'FORWARD'}

    if self.state is PORT_STATE_DISABLE:
        self.ofctl.set_port_status(self.ofport, self.state)

    while True:
        self.logger.info('[port=%d] %s / %s', self.ofport.port_no,
                         role_str[self.role], state_str[self.state],
                         extra=self.dpid_str)

        self.state_event = hub.Event()
        timer = self._get_timer()
        if timer:
            timeout = hub.Timeout(timer)
            try:
                self.state_event.wait()
            except hub.Timeout as t:
                if t is not timeout:
                    err_msg = 'Internal error. Not my timeout.'
                    raise RyuException(msg=err_msg)
                new_state = self._get_next_state()
                self._change_status(new_state, thread_switch=False)
            finally:
                timeout.cancel()
        else:
            self.state_event.wait()

        self.state_event = None

def _get_timer(self):
    timer = {PORT_STATE_DISABLE: None,
             PORT_STATE_BLOCK: None,
             PORT_STATE_LISTEN: self.port_times.forward_delay,
             PORT_STATE_LEARN: self.port_times.forward_delay,
             PORT_STATE_FORWARD: None}
```

```
        return timer[self.state]


    def _get_next_state(self):
        next_state = {PORT_STATE_DISABLE: None,
                      PORT_STATE_BLOCK: None,
                      PORT_STATE_LISTEN: PORT_STATE_LEARN,
                      PORT_STATE_LEARN: (PORT_STATE_FORWARD
                                         if (self.role is ROOT_PORT or
                                             self.role is DESIGNATED_PORT)
                                         else PORT_STATE_BLOCK),
                      PORT_STATE_FORWARD: None}
        return next_state[self.state]


    def _change_status(self, new_state, thread_switch=True):
        if new_state is not PORT_STATE_DISABLE:
            self.ofctl.set_port_status(self.ofport, new_state)


        if(new_state is PORT_STATE_FORWARD
           or (self.state is PORT_STATE_FORWARD
               and (new_state is PORT_STATE_DISABLE
                    or new_state is PORT_STATE_BLOCK))):
            self.topology_change_notify(new_state)


        if (new_state is PORT_STATE_DISABLE
                or new_state is PORT_STATE_BLOCK):
            self.send_tc_flg = False
            self.send_tc_timer = None
            self.send_tcn_flg = False
            self.send_bpdu_thread.stop()
        elif new_state is PORT_STATE_LISTEN:
            self.send_bpdu_thread.start()


        self.state = new_state
        self.send_event(EventPortStateChange(self.dp, self))


        if self.state_event is not None:
            self.state_event.set()
```

```
            self.state_event = None
        if thread_switch:
            hub.sleep(0)  # For thread switching.


    def _change_role(self, new_role):
        if self.role is new_role:
            return
        self.role = new_role
        if (new_role is ROOT_PORT
                or new_role is NON_DESIGNATED_PORT):
            self.wait_bpdu_thread.start()
        else:
            assert new_role is DESIGNATED_PORT
            self.wait_bpdu_thread.stop()


    def rcv_config_bpdu(self, bpdu_pkt):
        # Check received BPDU is superior to currently held BPDU.
        root_id = BridgeId(bpdu_pkt.root_priority,
                           bpdu_pkt.root_system_id_extension,
                           bpdu_pkt.root_mac_address)
        root_path_cost = bpdu_pkt.root_path_cost
        designated_bridge_id = BridgeId(bpdu_pkt.bridge_priority,
                                        bpdu_pkt.bridge_system_id_extension,
                                        bpdu_pkt.bridge_mac_address)
        designated_port_id = PortId(bpdu_pkt.port_priority,
                                    bpdu_pkt.port_number)


        msg_priority = Priority(root_id, root_path_cost,
                                designated_bridge_id,
                                designated_port_id)
        msg_times = Times(bpdu_pkt.message_age,
                          bpdu_pkt.max_age,
                          bpdu_pkt.hello_time,
                          bpdu_pkt.forward_delay)


        rcv_info = Stp.compare_bpdu_info(self.designated_priority,
                                         self.designated_times,
                                         msg_priority, msg_times)
        if rcv_info is SUPERIOR:
```

```
            self.designated_priority = msg_priority
            self.designated_times = msg_times


        chk_flg = False
        if ((rcv_info is SUPERIOR or rcv_info is REPEATED)
                and (self.role is ROOT_PORT
                     or self.role is NON_DESIGNATED_PORT)):
            self._update_wait_bpdu_timer()
            chk_flg = True
        elif rcv_info is INFERIOR and self.role is DESIGNATED_PORT:
            chk_flg = True


        # Check TopologyChange flag.
        rcv_tc = False
        if chk_flg:
            tc_flag_mask = 0b00000001
            tcack_flag_mask = 0b10000000
            if bpdu_pkt.flags & tc_flag_mask:
                self.logger.debug('[port=%d] receive TopologyChange BPDU.',
                                  self.ofport.port_no, extra=self.dpid_str)
                rcv_tc = True
            if bpdu_pkt.flags & tcack_flag_mask:
                self.logger.debug('[port=%d] receive TopologyChangeAck BPDU.',
                                  self.ofport.port_no, extra=self.dpid_str)
                if self.send_tcn_flg:
                    self.send_tcn_flg = False


        return rcv_info, rcv_tc


    def _update_wait_bpdu_timer(self):
        if self.wait_timer_event is not None:
            self.wait_timer_event.set()
            self.wait_timer_event = None
            self.logger.debug('[port=%d] Wait BPDU timer is updated.',
                              self.ofport.port_no, extra=self.dpid_str)
        hub.sleep(0)  # For thread switching.


    def _wait_bpdu_timer(self):
```

```
        time_exceed = False

        while True:
            self.wait_timer_event = hub.Event()
            message_age = (self.designated_times.message_age
                           if self.designated_times else 0)
            timer = self.port_times.max_age - message_age
            timeout = hub.Timeout(timer)
            try:
                self.wait_timer_event.wait()
            except hub.Timeout as t:
                if t is not timeout:
                    err_msg = 'Internal error. Not my timeout.'
                    raise RyuException(msg=err_msg)
                self.logger.info('[port=%d] Wait BPDU timer is exceeded.',
                                 self.ofport.port_no, extra=self.dpid_str)
                time_exceed = True
            finally:
                timeout.cancel()
                self.wait_timer_event = None

            if time_exceed:
                break

        if time_exceed:  # Bridge.recalculate_spanning_tree
            hub.spawn(self.wait_bpdu_timeout)

    def _transmit_bpdu(self):
        while True:
            # Send config BPDU packet if port role is DESIGNATED_PORT.
            if self.role == DESIGNATED_PORT:
                now = datetime.datetime.today()
                if self.send_tc_timer and self.send_tc_timer < now:
                    self.send_tc_timer = None
                    self.send_tc_flg = False

                if not self.send_tc_flg:
                    flags = 0b00000000
```

```
                log_msg = '[port=%d] Send Config BPDU.'
            else:
                flags = 0b00000001
                log_msg = '[port=%d] Send TopologyChange BPDU.'
            bpdu_data = self._generate_config_bpdu(flags)
            self.ofctl.send_packet_out(self.ofport.port_no, bpdu_data)
            self.logger.debug(log_msg, self.ofport.port_no,
                              extra=self.dpid_str)

        # Send Topology Change Notification BPDU until receive Ack.
        if self.send_tcn_flg:
            bpdu_data = self._generate_tcn_bpdu()
            self.ofctl.send_packet_out(self.ofport.port_no, bpdu_data)
            self.logger.debug('[port=%d] Send TopologyChangeNotify BPDU.',
                              self.ofport.port_no, extra=self.dpid_str)

        hub.sleep(self.port_times.hello_time)

def transmit_tc_bpdu(self):
    """ Set send_tc_flg to send Topology Change BPDU. """
    if not self.send_tc_flg:
        timer = datetime.timedelta(seconds=self.port_times.max_age
                                   + self.port_times.forward_delay)
        self.send_tc_timer = datetime.datetime.today() + timer
        self.send_tc_flg = True

def transmit_ack_bpdu(self):
    """ Send Topology Change Ack BPDU. """
    ack_flags = 0b10000001
    bpdu_data = self._generate_config_bpdu(ack_flags)
    self.ofctl.send_packet_out(self.ofport.port_no, bpdu_data)

def transmit_tcn_bpdu(self):
    self.send_tcn_flg = True

def _generate_config_bpdu(self, flags):
    src_mac = self.ofport.hw_addr
```

```
        dst_mac = bpdu.BRIDGE_GROUP_ADDRESS
        length = (bpdu.bpdu._PACK_LEN + bpdu.ConfigurationBPDUs.PACK_LEN
                  + llc.llc._PACK_LEN + llc.ControlFormatU._PACK_LEN)

        e = ethernet.ethernet(dst_mac, src_mac, length)
        l = llc.llc(llc.SAP_BPDU, llc.SAP_BPDU, llc.ControlFormatU())
        b = bpdu.ConfigurationBPDUs(
            flags=flags,
            root_priority=self.port_priority.root_id.priority,
            root_mac_address=self.port_priority.root_id.mac_addr,
            root_path_cost=self.port_priority.root_path_cost + self.path_cost,
            bridge_priority=self.bridge_id.priority,
            bridge_mac_address=self.bridge_id.mac_addr,
            port_priority=self.port_id.priority,
            port_number=self.ofport.port_no,
            message_age=self.port_times.message_age + 1,
            max_age=self.port_times.max_age,
            hello_time=self.port_times.hello_time,
            forward_delay=self.port_times.forward_delay)

        pkt = packet.Packet()
        pkt.add_protocol(e)
        pkt.add_protocol(l)
        pkt.add_protocol(b)
        pkt.serialize()

        return pkt.data

    def _generate_tcn_bpdu(self):
        src_mac = self.ofport.hw_addr
        dst_mac = bpdu.BRIDGE_GROUP_ADDRESS
        length = (bpdu.bpdu._PACK_LEN
                  + bpdu.TopologyChangeNotificationBPDUs.PACK_LEN
                  + llc.llc._PACK_LEN + llc.ControlFormatU._PACK_LEN)

        e = ethernet.ethernet(dst_mac, src_mac, length)
        l = llc.llc(llc.SAP_BPDU, llc.SAP_BPDU, llc.ControlFormatU())
        b = bpdu.TopologyChangeNotificationBPDUs()
```

```
        pkt = packet.Packet()
        pkt.add_protocol(e)
        pkt.add_protocol(l)
        pkt.add_protocol(b)
        pkt.serialize()

        return pkt.data


class PortThread(object):
    def __init__(self, function):
        super(PortThread, self).__init__()
        self.function = function
        self.thread = None

    def start(self):
        self.stop()
        self.thread = hub.spawn(self.function)

    def stop(self):
        if self.thread is not None:
            hub.kill(self.thread)
            hub.joinall([self.thread])
            self.thread = None


class BridgeId(object):
    def __init__(self, priority, system_id_extension, mac_addr):
        super(BridgeId, self).__init__()
        self.priority = priority
        self.system_id_extension = system_id_extension
        self.mac_addr = mac_addr
        self.value = bpdu.ConfigurationBPDUs.encode_bridge_id(
            priority, system_id_extension, mac_addr)
```

```
class PortId(object):
    def __init__(self, priority, port_no):
        super(PortId, self).__init__()
        self.priority = priority
        self.port_no = port_no
        self.value = bpdu.ConfigurationBPDUs.encode_port_id(priority, port_no)
```

```
class Priority(object):
    def __init__(self, root_id, root_path_cost,
                 designated_bridge_id, designated_port_id):
        super(Priority, self).__init__()
        self.root_id = root_id
        self.root_path_cost = root_path_cost
        self.designated_bridge_id = designated_bridge_id
        self.designated_port_id = designated_port_id
```

```
class Times(object):
    def __init__(self, message_age, max_age, hello_time, forward_delay):
        super(Times, self).__init__()
        self.message_age = message_age
        self.max_age = max_age
        self.hello_time = hello_time
        self.forward_delay = forward_delay
```

```
class OfCtl_v1_0(object):
    def __init__(self, dp):
        super(OfCtl_v1_0, self).__init__()
        self.dp = dp
```

```
    def send_packet_out(self, out_port, data):
        actions = [self.dp.ofproto_parser.OFPActionOutput(out_port, 0)]
        self.dp.send_packet_out(buffer_id=self.dp.ofproto.OFP_NO_BUFFER,
                                in_port=self.dp.ofproto.OFPP_CONTROLLER,
                                actions=actions, data=data)


    def set_port_status(self, port, state):
        ofproto_parser = self.dp.ofproto_parser
        mask = 0b1111111
        msg = ofproto_parser.OFPPortMod(self.dp, port.port_no, port.hw_addr,
                                        PORT_CONFIG_V1_0[state], mask,
                                        port.advertised)
        self.dp.send_msg(msg)



class OfCtl_v1_2later(OfCtl_v1_0):
    def __init__(self, dp):
        super(OfCtl_v1_2later, self).__init__(dp)


    def set_port_status(self, port, state):
        ofp = self.dp.ofproto
        parser = self.dp.ofproto_parser
        config = {ofproto_v1_2: PORT_CONFIG_V1_2,
                  ofproto_v1_3: PORT_CONFIG_V1_3}


        # Only turn on the relevant bits defined on OpenFlow 1.2+, otherwise
        # some switch that follows the specification strictly will report
        # OFPPMFC_BAD_CONFIG error.
        mask = 0b1100101
        msg = parser.OFPPortMod(self.dp, port.port_no, port.hw_addr,
                                config[ofp][state], mask, port.advertised)
        self.dp.send_msg(msg)


        if config[ofp][state] & ofp.OFPPC_NO_PACKET_IN:
            self.add_no_pkt_in_flow(port.port_no)
        else:
            self.del_no_pkt_in_flow(port.port_no)
```

```
def add_bpdu_pkt_in_flow(self):
    ofp = self.dp.ofproto
    parser = self.dp.ofproto_parser


    match = parser.OFPMatch(eth_dst=bpdu.BRIDGE_GROUP_ADDRESS)
    actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER,
                                      ofp.OFPCML_NO_BUFFER)]
    inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                         actions)]
    mod = parser.OFPFlowMod(self.dp, priority=BPDU_PKT_IN_PRIORITY,
                            match=match, instructions=inst)
    self.dp.send_msg(mod)


def add_no_pkt_in_flow(self, in_port):
    parser = self.dp.ofproto_parser


    match = parser.OFPMatch(in_port=in_port)
    mod = parser.OFPFlowMod(self.dp, priority=NO_PKT_IN_PRIORITY,
                            match=match)
    self.dp.send_msg(mod)


def del_no_pkt_in_flow(self, in_port):
    ofp = self.dp.ofproto
    parser = self.dp.ofproto_parser


    match = parser.OFPMatch(in_port=in_port)
    mod = parser.OFPFlowMod(self.dp, command=ofp.OFPFC_DELETE_STRICT,
                            out_port=ofp.OFPP_ANY, out_group=ofp.OFPG_ANY,
                            priority=NO_PKT_IN_PRIORITY, match=match)
    self.dp.send_msg(mod)
```