Alumno:

Calva Hernández José Manuel 2017630201

Redes de Computadora

Profesor: Axel Ernesto Moreno Cervantes  
Grupo: 2CM10  
Fecha: 11 / Junio /2018

Tarea No. 3

Spanning Tree



Instituto Politécnico Nacional

Escuela Superior de Cómputo

Índice

[Introducción 2](#_Toc516414852)

[Protocolo STP 2](#_Toc516414853)

[Funcionamiento 3](#_Toc516414854)

[Elección del puente raíz 3](#_Toc516414855)

[Elección de los puertos raíz 3](#_Toc516414856)

[Elección de los puertos designados 3](#_Toc516414857)

[Puertos bloqueados 4](#_Toc516414858)

[Mantenimiento del Spanning Tree 4](#_Toc516414859)

[Unidades de datos del protocolo puente 4](#_Toc516414860)

[Estado de los puertos 5](#_Toc516414861)

[Ejemplo Implementación (Python) 6](#_Toc516414862)

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) |