1 課題内容

- ルータをマルチプルテーブル化
  - テーブル構成とソースコードの説明
  - 動作確認

2 テーブル構成とソースコードの説明

2.1 テーブル構成

テーブル構成およびテーブル間の遷移を図 1 に示す．また，各テーブルの主な役割を下記に示す．

id 0: Classifier テーブル
Ether タイプ (IPv4 / ARP) によって，遷移するテーブルを決定する．

id 1: ARP Responder テーブル
ARP パケットの処理を行う．

id 2: Routing テーブル
宛先 IP アドレスを見て，インターフェースを決定する．

id 3: Interface Lookup テーブル
インターフェースによって，送信元 MAC アドレスを変更する．

id 4: ARP Lookup テーブル
宛先 IP アドレスによって，宛先 MAC アドレスを変更する．

id 5: Egress テーブル
指定されたポートにパケットを転送する．

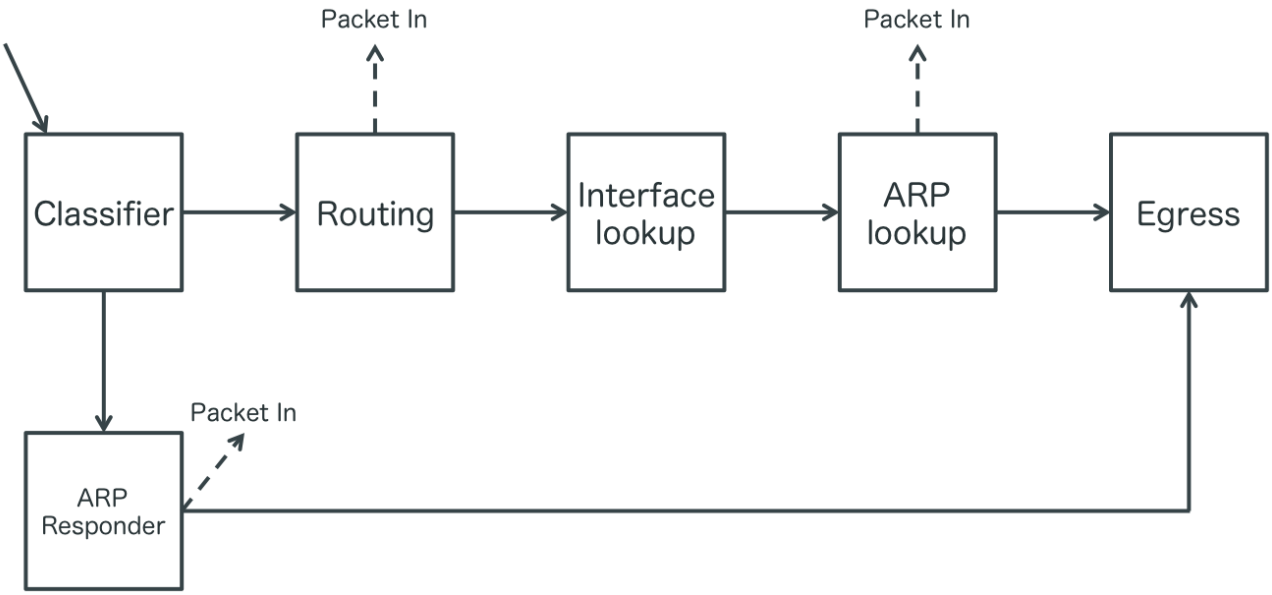

図 1: テーブル構成

2.2 動作概要

動作概要を説明するにあたり，図 2 に示すネットワークを用いる．図 2 のように host1 と host2 はルータを介して異なるネットワークに接続されており，それぞれの IP アドレスおよび MAC アドレスは図 2 に示したとおりである．

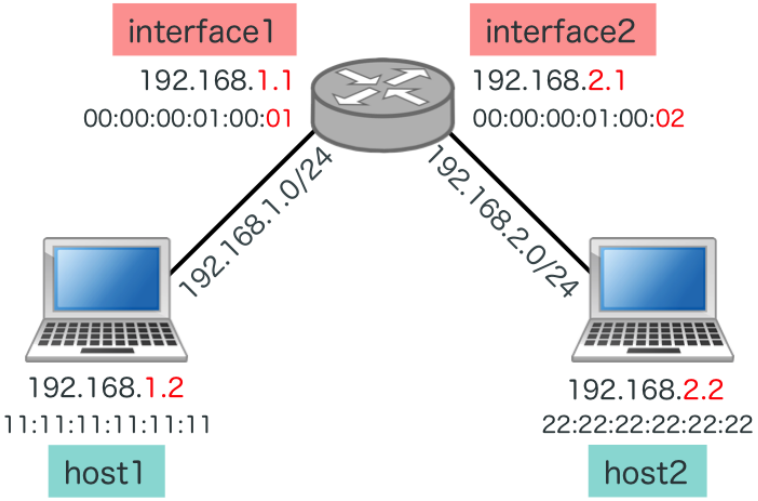

図 2: ネットワーク図

2.2.1 フローテーブルの初期状態

Trema 実行直後のフローテーブルの初期状態を以下に示す．

Priority	Match Fields	Instructions
0	Ether Type = IPv4 (0x0800)	Goto Routing Table
0	Ether Type = ARP (0x0806)	Goto ARP Responder Table

図 3: Classifier テーブル

Priority	Match Fields	Instructions
4545	IPv4 DST = 192.168.1.1	Packet In
4545	IPv4 DST = 192.168.2.1	Packet In
24	IPv4 DST = 192.168.1.0/24	reg0 := IPv4 DST Goto Interface Lookup Table
24	IPv4 DST = 192.168.2.0/24	reg0 := IPv4 DST Goto Interface Lookup Table
0	なし	reg0 := 192.168.1.2 Goto Interface Lookup Table

図 4: Routing テーブル

Priority	Match Fields	Instructions
24	reg0 = 192.168.1.0/24	reg1 := 1 (ポート番号) ETH SRC := 00:00:00:01:00:01 Goto ARP Lookup Table
24	reg0 = 192.168.2.0/24	reg2 := 2 (ポート番号) ETH SRC := 00:00:00:01:00:02 Goto ARP Lookup Table

図 5: Interface Lookup テーブル

Priority	Match Fields	Instructions
0	なし	Packet In

図 6: ARP Lookup テーブル

Priority	Match Fields	Instructions
0	なし	Output (reg1)

図 7: Egress テーブル

Priority	Match Fields	Instructions
1	ARP Request ポート = 1 ARP TPA = 192.168.1.1	Packet In ETH DST := ETH SRC ARP TPA := ARP SPA ARP THA := ARP SHA Operation := Reply ARP SHA := 00:00:00:01:00:01 ARP SPA := 192.168.1.1 ETH SRC := 192.168.1.1 IN PORT
1	ARP Reply ポート = 1 ARP TPA = 192.168.1.1	Packet In
1	ARP Request ポート = 2 ARP TPA = 192.168.2.1	Packet In ETH DST := ETH SRC ARP TPA := ARP SPA ARP THA := ARP SHA Operation := Reply ARP SHA := 00:00:00:01:00:02 ARP SPA := 192.168.2.1 ETH SRC := 192.168.2.1 IN PORT
1	ARP Reply ポート = 2 ARP TPA = 192.168.2.1	Packet In
0	ARP reg1 = 1	ARP SHA := 00:00:00:01:00:01 ARP SPA := 192.168.1.1 ETH SRC := 192.168.1.1 Goto Egress Table
0	ARP reg1 = 2	ARP SHA := 00:00:00:01:00:02 ARP SPA := 192.168.2.1 ETH SRC := 192.168.2.1 Goto Egress Table

図 8: ARP Responder テーブル

2.2.2 ARP Request に対する処理

図 9 のように host1 が interface1 に ARP Request を送信した際の処理について説明する．図 11 のように ARP Request パケットを受信したルータは，Classifier テーブル → ARP Responder テーブルと遷移する．そして，図 8 で示したように，ARP Request パケットを ARP Reply パケットに書き直し，元のポートへ Packet Out する．さらに，受信した ARP Request パケットをコントローラに Packet In をすることで，ARP Lookup テーブルに host1 の MAC アドレスの情報を新規フローエントリとして追加する (図 12)．最後に図 10 のように ARP Reply パケットを host1 が受信し，ARP Request に対する処理が終了する．

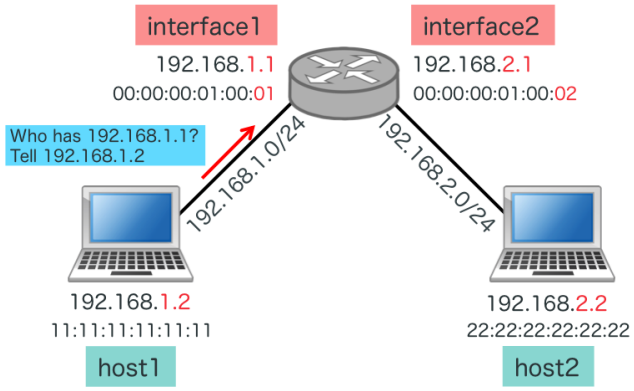

図 9: host1 が interface1 に ARP Request を送信

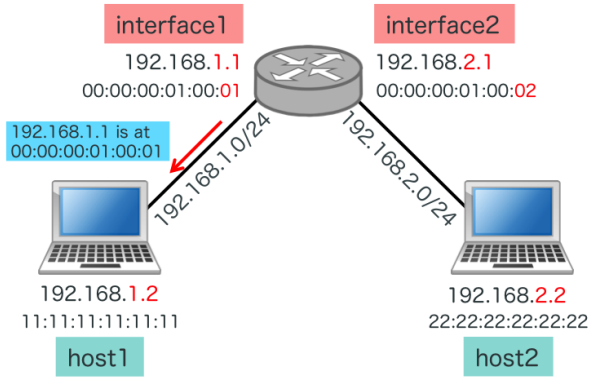

図 10: host1 が ARP Reply を受信

Ethernet Type Number	0x0806
Destination Mac Address	FF:FF:FF:FF:FF:FF
Source Mac Address	11:11:11:11:11:11
Operation	Request (1)
Sender Hardware Address	11:11:11:11:11:11
Sender Protocol Address	192.168.1.2
Target Hardware Address	00:00:00:00:00:00
Target Protocol Address	192.168.1.1

Ethernet Type Number	0x0806
Destination Mac Address	11:11:11:11:11:11
Source Mac Address	00:00:00:01:00:01
Operation	Reply (2)
Sender Hardware Address	00:00:00:01:00:01
Sender Protocol Address	192.168.1.1
Target Hardware Address	11:11:11:11:11:11
Target Protocol Address	192.168.1.2

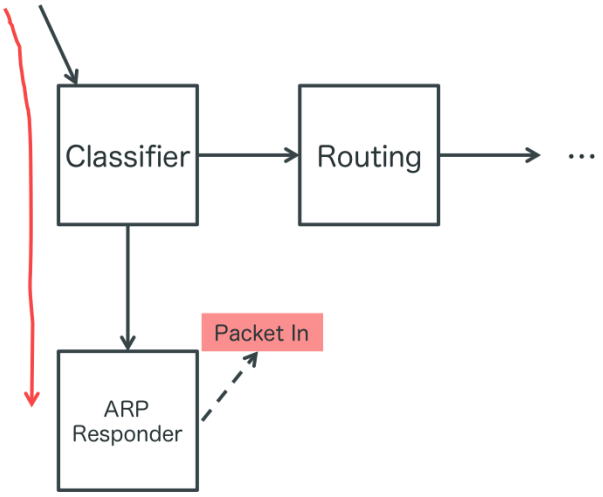

図 11: ARP Request パケットのフロー図

Priority	Match Fields	Instructions
4545	reg0 = 192.168.1.2	ETH DST := 11:11:11:11:11:11 Goto Egress Table
0	なし	Packet In

図 12: アドレス解決後の ARP Lookup テーブル

2.2.3 ARP Reply に対する処理

ルータがパケットを host2 へ転送する場合を考える．転送パケットの宛先 MAC アドレスが不明なとき，ARP Lookup テーブルで Packet In が発生する．Packet In を受け取ったコントローラは，当該パケットをキューにエンキューし，後ほど処理をするパケットとして保存しておく．そして，図 13 のようにコントローラは host2 へ ARP Request を送信し，host2 の MAC アドレスを問い合わせる．すると，図 14 のように host2 から ARP Reply が返ってくる．ARP Reply パケットを受信したルータは Classifier テーブル → ARP Responder テーブルと遷移し，Packet In を発生させる (図 15)．ARP Reply パケットの Packet In を得たコントローラは，ARP Lookup テーブルに host2 の MAC アドレスの情報を新規フローエントリとして追加する (図 16)．さらに，host2 へ転送すべきパケットをキューから全てデキューし，再びフローテーブルの Classifier テーブルから流すことで，溜め込んでいたパケットの転送が可能となる．

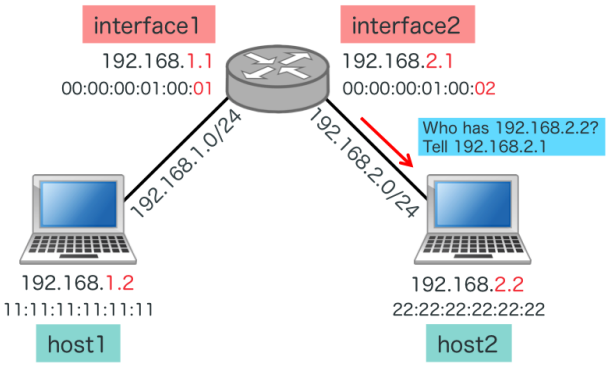

図 13: interface2 が host2 に ARP Request を送信

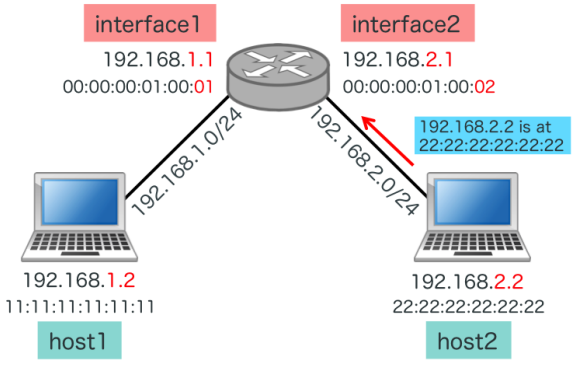

図 14: interface2 が ARP Reply を受信

Ethernet Type Number	0x0806
Destination Mac Address	00:00:00:01:00:02
Source Mac Address	22:22:22:22:22:22
Operation	Reply (2)
Sender Hardware Address	22:22:22:22:22:22
Sender Protocol Address	192.168.2.2
Target Hardware Address	00:00:00:01:00:02
Target Protocol Address	192.168.2.1

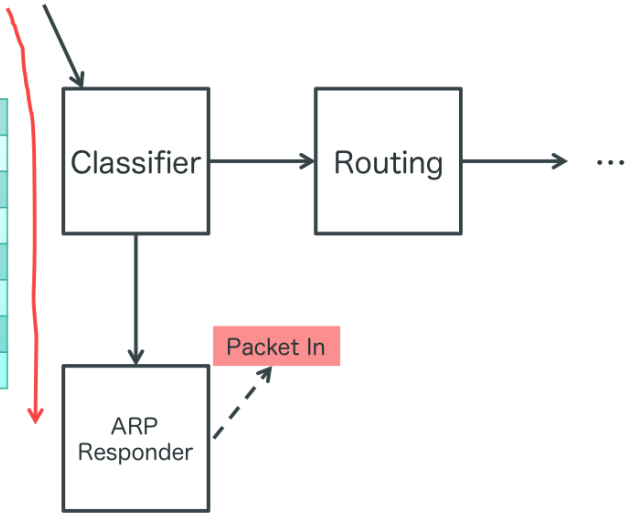

図 15: ARP Reply パケットのフロー図

Priority	Match Fields	Instructions
4545	reg0 = 192.168.1.2	ETH DST := 11:11:11:11:11:11 Goto Egress Table
4545	reg0 = 192.168.2.2	ETH DST := 22:22:22:22:22:22 Goto Egress Table
0	なし	Packet In

図 16: アドレス解決後の ARP Lookup テーブル その 2

2.2.4 パケットの転送

host1 がルータを介して host2 へパケットを転送する場合を考える．host1 が ARP Request をルータに送信する処理については 2.2.2 で述べたため省略する．例として host1 が ping を送信した場合を考える．host1 が送信した ping パケットを受け取ったルータは，順にフローテーブルを遷移していくが，ARP Lookup テーブルに host2 の MAC アドレスが登録されていないため，2.2.3 で述べたように host2 へ MAC アドレスを問い合わせる．そして，host2 のアドレス解決後，再度 ping パケットを Classifier テーブルから流す．すると，図 17 のように ping パケットが流れていき，最終的に書き換えられた ping パケットが host2 が属するポート 2 へ Packet Out される．図 17 のフローの上に示している表は左側のテーブルから右側のテーブルに遷移する際のパケットの情報を表しており，書き換えられたフィールドは赤色で示している．また，レジスタ 0 には宛先 IP アドレスが，レジスタ 1 には転送先のポート番号が格納されている．そして，ポート 2 へ Packet Out された ping パケットを host2 が受信し，パケットの転送は完了する．

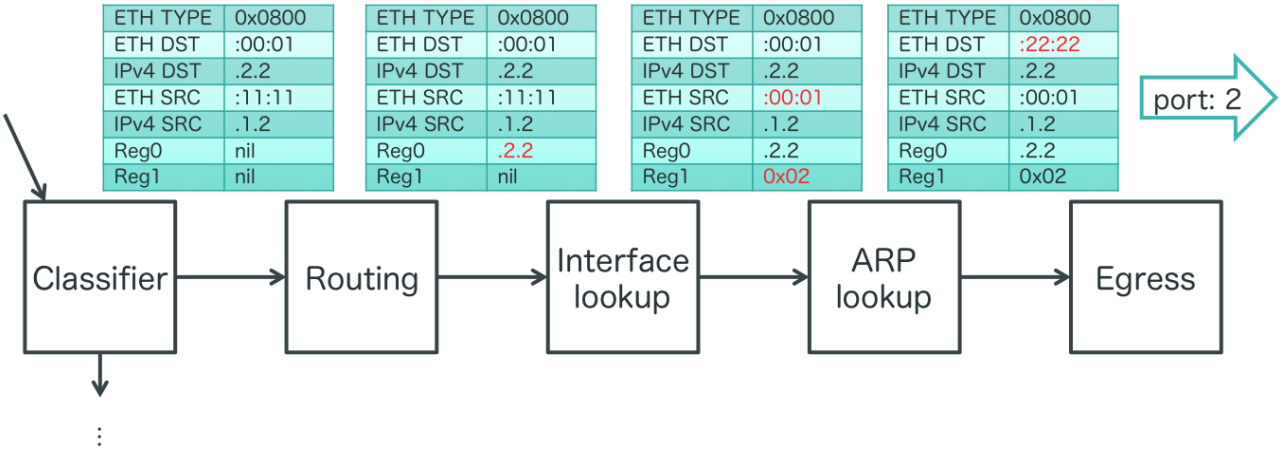

図 17: ping パケットのフロー図

2.2.5 ping に対する応答処理

host1 が interface1 に ping パケットを送信する場合を考える．図 18 のように，ルータ宛のパケットは Routing テーブルで Packet In を発生させている．これは Interface Lookup テーブルで送信元 MAC アドレスを interface の MAC アドレスに書き換える処理をしており，これ以降で Packet In を発生させてしまうと，送信元 MAC アドレスが書き換わった状態のパケットをコントローラが受信してしまい，本来の送信元に ping 応答を送れなくなる．そのため，Interface Lookup テーブルを通る前に Packet In を発生させる必要があり，ルータ宛のパケットは Routing テーブルで Packet In を発生させることにした．また，Open vSwitch は ICMP の書き換えに対応していないため，ping 応答はコントローラで直接行うことにした．ping 応答については，2.3 にて後述する．

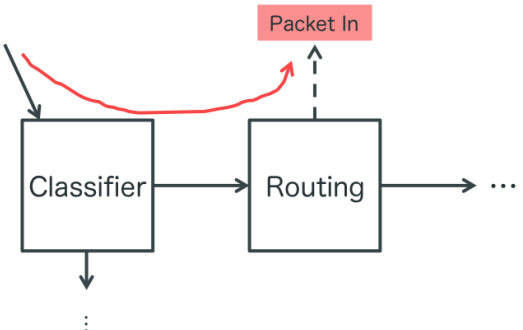

図 18: ping 応答のフロー図

2.3 ソースコードの説明

2.3.1 定数の定義

後述するソースコードの説明をするため，まず定義している定数を以下に示す．

```
5   CLASSIFIER_TABLE_ID        = 0
6   ARP_RESPONDER_TABLE_ID     = 1
7   ROUTING_TABLE_ID           = 2
8   INTERFACE_LOOKUP_TABLE_ID  = 3
9   ARP_LOOKUP_TABLE_ID        = 4
10  EGRESS_TABLE_ID            = 5
11
12  ETH_IPv4 = 0x0800
13  ETH_ARP  = 0x0806
14
15  MASK32 = 0xFFFFFFFF
```

2.3.2 テーブルの初期化

#switch_ready ハンドラが呼び出されると，以下のようにそれぞれのフローテーブルを初期化するメソッドを呼び出している．それぞれのフローテーブルの初期化については後述する．

```
25  def switch_ready(dpid)
26    init_classifier_flow_entry(dpid)
27    init_arp_responder_flow_entry(dpid)
28    init_routing_flow_entry(dpid)
29    init_interface_lookup_flow_entry(dpid)
30    init_arp_lookup_flow_entry(dpid)
31    init_packet_out_flow_entry(dpid)
32  end
```

2.3.3 Classifier テーブルの初期化

#init_classifier_flow_entry メソッドでは，Ethernet タイプ番号を見て，IPv4 であれば Routing テーブルへ，ARP であれば ARP Responder テーブルへ遷移させるフローエントリを Classifier テーブルに追加する処理を行っている．なお，#add_goto_table_flow_entry メソッドはテーブル A からテーブル B へ転送するフローエントリを追加するメソッドである．可読性向上のため，単純なテーブルの遷移処理は全て#add_goto_table_flow_entry メソッドに任せている．

```
156 def init_classifier_flow_entry(dpid)
157   add_goto_table_flow_entry(
158     dpid,
159     CLASSIFIER_TABLE_ID,
160     ROUTING_TABLE_ID,
161     Match.new(ether_type: ETH_IPv4),
162   )
163
164   add_goto_table_flow_entry(
165     dpid,
166     CLASSIFIER_TABLE_ID,
167     ARP_RESPONDER_TABLE_ID,
168     Match.new(ether_type: ETH_ARP),
169   )
170 end
```

2.3.4 ARP Responder テーブルの初期化

#init_arp_responder_flow_entry メソッドでは，それぞれの interface に対して，ARP Request および ARP Reply に関するフローエントリを ARP Responder テーブルに追加する．ここで呼びだされている #add_arp_request_flow_entry メソッドは ARP Request に関するメソッド，#add_arp_reply_flow_entry メソッドは ARP Reply に関するメソッドである．2.2.1 で示した通りに ARP Responder テーブルを初期化するだけで，特に重要な処理はしていないため，これら 2 つのメソッドについての説明は割愛する．

また，ルータから ARP Request を送信する場合も，このフローテーブルを介して送信されるためそのフローエントリを#init_arp_responder_flow_entry メソッドで追加している．レジスタ 1(出力ポート番号) を参照して，それに応じて送信元 MAC/IP アドレスフィールドを書き換えて，Egress テーブルに遷移するフローエントリである．

```
172   def init_arp_responder_flow_entry(dpid)
173     @interfaces.get_list.each do |each|
174       add_arp_request_flow_entry(dpid, each)
175       add_arp_reply_flow_entry(dpid, each)
176       add_sending_arp_request_flow_entry(dpid, each)
177     end
178   end
```

```
237   def add_sending_arp_request_flow_entry(dpid, interface)
238     send_flow_mod_add(
239       dpid,
240       table_id: ARP_RESPONDER_TABLE_ID,
241       priority: 0,
242       match: Match.new(
243         ether_type: ETH_ARP,
244         reg1: interface.port_number,
245       ),
246       instructions: [
247         Apply.new([
248           SetArpSenderHardwareAddress.new(interface.mac_address),
249           SetArpSenderProtocolAddress.new(interface.ip_address),
250           SetSourceMacAddress.new(interface.mac_address),
251         ]),
252         GotoTable.new(EGRESS_TABLE_ID),
253       ],
254     )
255   end
```

2.3.5 Routing テーブルの初期化

#init_routing_flow_entry メソッドでは，interface ごとに routing 処理を行うフローエントリを Routing テーブルに追加する．#add_transfer_routing_flow_entry メソッドは，宛先 IP アドレスが interface のネットワークアドレスであれば，レジスタ 0 に宛先 IP アドレスを格納し，Interfac Lookup テーブルに遷移させるフローエントリを追加している．**このとき，288 行目で優先度を指定しているが，優先度をプレフィックス長にすることで，ロンゲストマッチを実現している．**また，2.2.5 で述べたようにルータ宛の ping パケットを Routing テーブルで Packet In する必要があるため，#add_interface_routing_flow_entry メソッドでルータ宛のパケットに関するフローエントリを追加している．

さらに，#init_arp_responder_flow_entry メソッドでは，#add_default_routing_flow_entry メソッドを呼び出すことで，デフォルトルーティングに関するフローエントリも追加している (ソースコードは省略)．

```
257   def init_routing_flow_entry(dpid)
258     add_default_routing_flow_entry(dpid)
259     @interfaces.get_list.each do |interface|
```

```
260       add_transfer_routing_flow_entry(dpid, interface)
261       add_interface_routing_flow_entry(dpid, interface)
262     end
263   end
```

```
284   def add_transfer_routing_flow_entry(dpid, interface)
285     send_flow_mod_add(
286       dpid,
287       table_id: ROUTING_TABLE_ID,
288       priority: interface.netmask_length,
289       match: Match.new(
290         ether_type: ETH_IPv4,
291         ipv4_destination_address: interface.ip_address.mask(interface.netmask_length),
292         ipv4_destination_address_mask: gen_mask(interface.netmask_length),
293       ),
294       instructions: [
295         Apply.new([
296           NiciraRegMove.new(
297             from: :ipv4_destination_address,
298             to: :reg0,
299           ),
300         ]),
301         GotoTable.new(INTERFACE_LOOKUP_TABLE_ID),
302       ],
303     )
304   end
```

```
306   def add_interface_routing_flow_entry(dpid, interface)
307     send_flow_mod_add(
308       dpid,
309       table_id: ROUTING_TABLE_ID,
310       priority: 4545,
311       match: Match.new(
312         ether_type: ETH_IPv4,
313         ipv4_destination_address: interface.ip_address,
314       ),
315       instructions: [
316         Apply.new([
317           SendOutPort.new(:controller)
318         ]),
319       ],
320     )
321   end
```

2.3.6 Interface Lookup テーブル

`#init_interface_lookup_flow_entry` メソッドでは，interface ごとにレジスタ 0 の値が当該 interface のネットワークアドレスに含まれていれば，レジスタ 1 に当該 interface のポート番号を格納し，送信元 MAC アドレスを当該 interface の MAC アドレスに書き換え，ARP Lookup テーブルに遷移するフローエントリを追加している．前述のとおり，レジスタ 0 には宛先 IP アドレスが格納されているが，宛先 IP アドレスを直接参照せずにレジスタ 0 に格納しておくことで，デフォルトルーティングにも対応ができている．

```
323   def init_interface_lookup_flow_entry(dpid)
324     @interfaces.get_list.each do |each|
325       send_flow_mod_add(
326         dpid,
327         table_id: INTERFACE_LOOKUP_TABLE_ID,
328         priority: each.netmask_length,
329         match: Match.new(
```

```
330           reg0: each.ip_address.mask(each.netmask_length).to_i,
331           reg0_mask: gen_mask(each.netmask_length).to_i,
332         ),
333         instructions: [
334           Apply.new([
335             NiciraRegLoad.new(
336               each.port_number,
337               :reg1,
338             ),
339             SetSourceMacAddress.new(each.mac_address),
340           ]),
341           GotoTable.new(ARP_LOOKUP_TABLE_ID),
342         ],
343       )
344     end
345   end
```

2.3.7 ARP Lookup テーブル

#init_arp_lookup_flow_entry メソッドでは，宛先 IP アドレスが ARP Lookup テーブルに登録されていなかったときに，Packet In を発生させるフローエントリを追加している．ARP Lookup テーブルには ARP Request や ARP Reply を取得する度に優先度が高いフローエントリが追加されていくが，初期状態はこのフローエントリのみである．

```
347   def init_arp_lookup_flow_entry(dpid)
348     send_flow_mod_add(
349       dpid,
350       table_id: ARP_LOOKUP_TABLE_ID,
351       priority: 0,
352       match: Match.new,
353       instructions: [
354         Apply.new([
355           SendOutPort.new(:controller),
356         ]),
357       ],
358     )
359   end
```

2.3.8 Egress テーブル

init_egress_flow_entry メソッドでは，宛先 MAC アドレスや送信元 MAC アドレスが書き換えられたパケットをレジスタ 1 に指定されたポートに Packet Out をするフローエントリを Egress テーブルに追加するメソッドである．この Egress テーブルは終始このフローエントリのみである．

```
361   def init_egress_flow_entry(dpid)
362     send_flow_mod_add(
363       dpid,
364       table_id: EGRESS_TABLE_ID,
365       priority: 0,
366       match: Match.new,
367       instructions: [
368         Apply.new([
369           NiciraSendOutPort.new(:reg1),
370         ]),
371       ],
372     )
373   end
```

2.3.9 ARP Request パケットの Packet In

ARP Request パケットの Packet In が発生すると，#packet_in ハンドラ内で#add_arp_lookup_flow_entry メソッドが呼び出され，ARP Lookup テーブルに送信元 MAC アドレスの情報がフローエントリに追加され，アドレス解決がされる．このとき，2.3.7 で説明したアドレス解決のために Packet In を発生させるフローエントリより優先度を高くしておく必要がある．

```
375   def add_arp_lookup_flow_entry(dpid, message)
376     send_flow_mod_add(
377       dpid,
378       table_id: ARP_LOOKUP_TABLE_ID,
379       priority: 4545,
380       match: Match.new(
381         reg0: message.sender_protocol_address.to_i,
382       ),
383       instructions: [
384         Apply.new([
385           SetDestinationMacAddress.new(message.sender_hardware_address),
386         ]),
387         GotoTable.new(EGRESS_TABLE_ID),
388       ],
389     )
390   end
```

2.3.10 ARP Reply パケットの Packet In

ARP Reply パケットの Packet In が発生すると，#packet_in ハンドラ内で#add_arp_lookup_flow_entry メソッドが呼び出され，アドレス解決がされる (2.3.9 と同様なので詳細は 2.3.9 を参照)．

また，キューに溜め込んだアドレス解決待ちのパケットの処理をする必要があるため，#packet_in_arp_reply メソッドを呼び出しパケットの処理をしている．このとき，**75 行目の** `SendOutPort.new(:table)` **を actions に指定することで，再度 Classifier テーブルからパケットを流すようにしている．**

```
64   def packet_in_arp_reply(dpid, message)
65     flush_unsent_packets(
66       dpid,
67       message.data.sender_protocol_address,
68     )
69   end
70
71   def flush_unsent_packets(dpid, destination_ip)
72     @unresolved_packet_queue[destination_ip].each do |each|
73       send_packet_out(dpid,
74         raw_data: each.to_binary_s,
75         actions: SendOutPort.new(:table),
76       )
77     end
78     @unresolved_packet_queue[destination_ip] = []
79   end
```

2.3.11 IPv4 パケットの Packet In (他端末に転送するパケット)

他端末宛の IPv4 パケットの PacketIn が発生した場合，#packet_in_ipv4 メソッドを介して，#send_later メソッドが呼び出される．104 行目でパケットを一時キューに溜め込んだ後に，#send_arp_request メソッドを呼び出し，アドレス解決のために ARP Request を宛先端末に送信する．与えられた Openflow1.0 版のコードではここで ARP Request を直接 Packet Out をしていたが，本プログラムでは 2.3.10 と同様に `SendOutPort.new(:table)` を actions に指定することで，フローテーブルを介して ARP Request を送信するようにしている．

```
98   def send_later(dpid, message)
99     destination_ip = message.destination_ip_address
100
101    interface = @interfaces.find_by_prefix(destination_ip)
102    return if !interface || (interface.port_number == message.in_port)
103
104    @unresolved_packet_queue[destination_ip] += [message.data]
105    send_arp_request(dpid, destination_ip, interface)
106  end
107
108  def send_arp_request(dpid, destination_ip, interface)
109    arp_request =
110      Arp::Request.new(
111        source_mac: interface.mac_address,
112        sender_protocol_address: interface.ip_address,
113        target_protocol_address: destination_ip,
114      )
115    send_packet_out(
116      dpid,
117      raw_data: arp_request.to_binary,
118      actions: [
119        NiciraRegLoad.new(interface.port_number, :reg1),
120        SendOutPort.new(:table),
121      ]
122    )
123  end
```

2.3.12 IPv4 パケットの Packet In (ルータ宛の ping パケット)

ルータ宛の IPv4 パケット (ping) の PacketIn が発生した場合，`#packet_in_ipv4` メソッドを介して，`#packet_in_icmpv4_echo_request` メソッドが呼び出される．open vSwitch は ICMP の書き換えに対応していないため，ping 応答のみ直接ポート番号を指定して，`#send_packet_out` メソッドを呼び出している．

```
125  def packet_in_icmpv4_echo_request(dpid, message)
126    icmp_request = Icmp.read(message.raw_data)
127    send_packet_out(
128      dpid,
129      raw_data: create_icmp_reply(icmp_request).to_binary,
130      actions: SendOutPort.new(message.in_port),
131    )
132  end
```

3 動作確認

simple_router.conf および trema.conf を以下の内容にして，動作確認を行った．

```
# Simple router configuration
module Configuration
  INTERFACES = [
    {
      port: 1,
      mac_address: '00:00:00:01:00:01',
      ip_address: '192.168.1.1',
      netmask_length: 24
    },
    {
      port: 2,
      mac_address: '00:00:00:01:00:02',
      ip_address: '192.168.2.1',
      netmask_length: 24
    }
  ]

  ROUTES = [
    {
      destination: '0.0.0.0',
      netmask_length: 0,
      next_hop: '192.168.1.2'
    }
  ]
end
```

```
# Trema configuration
vswitch('0x1') { dpid 0x1 }
netns('host1') {
  ip '192.168.1.2'
  netmask '255.255.255.0'
  route net: '0.0.0.0', gateway: '192.168.1.1'
}
netns('host2') {
  ip '192.168.2.2'
  netmask '255.255.255.0'
  route net: '0.0.0.0', gateway: '192.168.2.1'
}
link '0x1', 'host1'
link '0x1', 'host2'
```

3.1 起動直後のフローテーブル

起動直後のフローテーブルのダンプを以下に示す (一部，改行などを加えて見やすく編集している)．2.2.1 で示した通りにフローテーブルが初期化されており，正常に動作していることがこのダンプから分かる．

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=3.439s, table=0, n_packets=0, n_bytes=0, priority=0,
    ip
    actions=goto_table:2
 cookie=0x0, duration=3.402s, table=0, n_packets=0, n_bytes=0, priority=0,
    arp
    actions=goto_table:1

 cookie=0x0, duration=3.402s, table=1, n_packets=0, n_bytes=0, priority=1,
    arp,in_port=1,arp_tpa=192.168.1.1,arp_op=1
    actions=
```

```
12      CONTROLLER:65535,move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],
13      move:NXM_OF_ARP_SPA[]->NXM_OF_ARP_TPA[],move:NXM_NX_ARP_SHA[]->NXM_NX_ARP_THA[],
14      set_field:2->arp_op,set_field:00:00:00:01:00:01->arp_sha,
15      set_field:192.168.1.1->arp_spa,set_field:00:00:00:01:00:01->eth_src,IN_PORT
16  cookie=0x0, duration=3.402s, table=1, n_packets=0, n_bytes=0, priority=1,
17    arp,in_port=1,arp_tpa=192.168.1.1,arp_op=2
18    actions=CONTROLLER:65535
19  cookie=0x0, duration=3.396s, table=1, n_packets=0, n_bytes=0, priority=1,
20    arp,in_port=2,arp_tpa=192.168.2.1,arp_op=1
21    actions=
22      CONTROLLER:65535,move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],
23      move:NXM_OF_ARP_SPA[]->NXM_OF_ARP_TPA[],move:NXM_NX_ARP_SHA[]->NXM_NX_ARP_THA[],
24      set_field:2->arp_op,set_field:00:00:00:01:00:02->arp_sha,
25      set_field:192.168.2.1->arp_spa,set_field:00:00:00:01:00:02->eth_src,IN_PORT
26  cookie=0x0, duration=3.393s, table=1, n_packets=0, n_bytes=0, priority=1,
27    arp,in_port=2,arp_tpa=192.168.2.1,arp_op=2
28    actions=CONTROLLER:65535
29  cookie=0x0, duration=3.402s, table=1, n_packets=0, n_bytes=0, priority=0,
30    arp,reg1=0x1
31    actions=
32      set_field:00:00:00:01:00:01->arp_sha,set_field:192.168.1.1->arp_spa,
33      set_field:00:00:00:01:00:01->eth_src,goto_table:5
34  cookie=0x0, duration=3.388s, table=1, n_packets=0, n_bytes=0, priority=0,
35    arp,reg1=0x2
36    actions=
37      set_field:00:00:00:01:00:02->arp_sha,set_field:192.168.2.1->arp_spa,
38      set_field:00:00:00:01:00:02->eth_src,goto_table:5
39
40  cookie=0x0, duration=3.379s, table=2, n_packets=0, n_bytes=0, priority=4545,
41    ip,nw_dst=192.168.1.1
42    actions=CONTROLLER:65535
43  cookie=0x0, duration=3.373s, table=2, n_packets=0, n_bytes=0, priority=4545,
44    ip,nw_dst=192.168.2.1
45    actions=CONTROLLER:65535
46  cookie=0x0, duration=3.382s, table=2, n_packets=0, n_bytes=0, priority=24,
47    ip,nw_dst=192.168.1.0/24
48    actions=move:NXM_OF_IP_DST[]->NXM_NX_REG0[],goto_table:3
49  cookie=0x0, duration=3.375s, table=2, n_packets=0, n_bytes=0, priority=24,
50    ip,nw_dst=192.168.2.0/24
51    actions=move:NXM_OF_IP_DST[]->NXM_NX_REG0[],goto_table:3
52  cookie=0x0, duration=3.386s, table=2, n_packets=0, n_bytes=0, priority=0
53    actions=load:0xc0a80102->NXM_NX_REG0[],goto_table:3
54
55  cookie=0x0, duration=3.368s, table=3, n_packets=0, n_bytes=0, priority=24,
56    reg0=0xc0a80100/0xffffff00
57    actions=load:0x1->NXM_NX_REG1[],set_field:00:00:00:01:00:01->eth_src,goto_table:4
58  cookie=0x0, duration=3.361s, table=3, n_packets=0, n_bytes=0, priority=24,
59    reg0=0xc0a80200/0xffffff00
60    actions=load:0x2->NXM_NX_REG1[],set_field:00:00:00:01:00:02->eth_src,goto_table:4
61
62  cookie=0x0, duration=3.359s, table=4, n_packets=0, n_bytes=0, priority=0
63    actions=CONTROLLER:65535
64
65  cookie=0x0, duration=3.356s, table=5, n_packets=0, n_bytes=0, priority=0
66    actions=output:NXM_NX_REG1[]
```

3.2 端末間のパケット転送 (ping)

host1 から host2 へ ping を 10 回送信し，パケット転送の動作確認を行った．その結果を以下に示す．1 パケット目は応答に 61.7ms 要しているが，2 パケット目は以降は約 0.06ms となっており 1 パケット目と比較して約 1/1000 となっている．これは，1 パケット目では，ルータが host2 のアドレス解決を行っているため転送に時間を要したからだと考えられる．

また，ping 送信後のフローテーブルのダンプを一部抜粋して以下に示す．n_packets の値が変化したフローエントリと新たに追加されたフローエントリ (先頭に*を記述) を抜粋した．4–14 行目より，ルータが受信した ARP Request/Reply 共に正常にフローテーブルを通過していることが分かる．16–18 行目より，ルータから host2 へ送信する ARP Request が正常にフローテーブルを通過していることが分かる．38-39 行目より，1 パケット目のみきちんとアドレス解決のために Packet In を発生させていることが分かる．23,30 行目のフローエントリを通過したパケットが 11 パケットとなっているが，これは 1 パケット目が 2 度通過していることを表しており，アドレス解決後に再度 Classifier テーブルからパケットが正常に流れていることが分かる．

```
$ bin/trema netns host1
# ping  -c 10 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=61.7 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.292 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.053 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=64 time=0.063 ms
64 bytes from 192.168.2.2: icmp_seq=5 ttl=64 time=0.060 ms
64 bytes from 192.168.2.2: icmp_seq=6 ttl=64 time=0.054 ms
64 bytes from 192.168.2.2: icmp_seq=7 ttl=64 time=0.057 ms
64 bytes from 192.168.2.2: icmp_seq=8 ttl=64 time=0.058 ms
64 bytes from 192.168.2.2: icmp_seq=9 ttl=64 time=0.059 ms
64 bytes from 192.168.2.2: icmp_seq=10 ttl=64 time=0.061 ms

--- 192.168.2.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9002ms
rtt min/avg/max/mdev = 0.053/6.250/61.744/18.498 ms
```

```
OFPST_FLOW reply (OF1.3) (xid=0x2):

...
 cookie=0x0, duration=759.388s, table=1, n_packets=1, n_bytes=42, priority=1,
   arp,in_port=1,arp_tpa=192.168.1.1,arp_op=1
   actions=
     CONTROLLER:65535, ...
...
 cookie=0x0, duration=759.383s, table=1, n_packets=1, n_bytes=42, priority=1,
   arp,in_port=2,arp_tpa=192.168.2.1,arp_op=1
   actions=CONTROLLER:65535, ...
 cookie=0x0, duration=759.380s, table=1, n_packets=1, n_bytes=42, priority=1,
   arp,in_port=2,arp_tpa=192.168.2.1,arp_op=2
   actions=CONTROLLER:65535
...
 cookie=0x0, duration=759.375s, table=1, n_packets=1, n_bytes=64, priority=0,
   arp,reg1=0x2
   actions=set_field:00:00:00:01:00:02->arp_sha,...
...

 cookie=0x0, duration=759.369s, table=2, n_packets=10, n_bytes=980, priority=24,
   ip,nw_dst=192.168.1.0/24 actions=move:NXM_OF_IP_DST[]->NXM_NX_REG0[],goto_table:3
 cookie=0x0, duration=759.362s, table=2, n_packets=11, n_bytes=1078, priority=24,
   ip,nw_dst=192.168.2.0/24 actions=move:NXM_OF_IP_DST[]->NXM_NX_REG0[],goto_table:3
...

 cookie=0x0, duration=759.355s, table=3, n_packets=10, n_bytes=980, priority=24,
```

```
28     reg0=0xc0a80100/0xffffff00
29     actions=load:0x1->NXM_NX_REG1[],...
30  cookie=0x0, duration=759.348s, table=3, n_packets=11, n_bytes=1078, priority=24,
31     reg0=0xc0a80200/0xffffff00
32     actions=load:0x2->NXM_NX_REG1[],...
33
34 *cookie=0x0, duration=261.220s, table=4, n_packets=10, n_bytes=980, priority=4545,
35     reg0=0xc0a80102 actions=set_field:26:aa:67:24:0b:e2->eth_dst,goto_table:5
36 *cookie=0x0, duration=256.146s, table=4, n_packets=10, n_bytes=980, priority=4545,
37     reg0=0xc0a80202 actions=set_field:ae:b8:17:c3:38:b8->eth_dst,goto_table:5
38  cookie=0x0, duration=759.346s, table=4, n_packets=1, n_bytes=98, priority=0
39     actions=CONTROLLER:65535
40
41  cookie=0x0, duration=759.343s, table=5, n_packets=21, n_bytes=2024, priority=0
42     actions=output:NXM_NX_REG1[]
```

続けて，host2 から host1 へ ping を 10 回送信し，動作確認を行った．その結果を以下に示す．前述の host1 から host2 への ping 送信とは異なり，1 パケット目から応答時間が 1ms 以下になっている．これは，ルータの ARP Lookup テーブルで host1, host2 共にアドレス解決済みであるため，ルータから host1 へ ARP Request を送信する必要がなく，応答が早くなったと考えられる．

```
1  $ bin/trema netns host2
2  # ping -c 10 192.168.1.2
3  PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
4  64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.402 ms
5  64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.067 ms
6  64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.059 ms
7  64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.047 ms
8  64 bytes from 192.168.1.2: icmp_seq=5 ttl=64 time=0.064 ms
9  64 bytes from 192.168.1.2: icmp_seq=6 ttl=64 time=0.058 ms
10 64 bytes from 192.168.1.2: icmp_seq=7 ttl=64 time=0.044 ms
11 64 bytes from 192.168.1.2: icmp_seq=8 ttl=64 time=0.044 ms
12 64 bytes from 192.168.1.2: icmp_seq=9 ttl=64 time=0.082 ms
13 64 bytes from 192.168.1.2: icmp_seq=10 ttl=64 time=0.166 ms
14
15 --- 192.168.1.2 ping statistics ---
16 10 packets transmitted, 10 received, 0% packet loss, time 8999ms
17 rtt min/avg/max/mdev = 0.044/0.103/0.402/0.105 ms
```

3.2.1 ルータの ping 応答

host1 から interface1 へ向けて ping を 10 回送信し，ルータの ping 応答の動作確認を行った (ルータは再起動している)．結果，ping 応答時間が平均 9.8ms となり，端末間の ping 送信と比較して 100 倍以上の時間を要している．これは，ルータ宛の ping パケットは全て Packet In を発生させ，コントローラから ping 応答を行っているためであると考えられる．この結果より，高速化をするためにはいかに Packet In を発生させないようにフローテーブルを設計するかが重要であることが分かる．

```
1  $ bin/trema netns host1
2  # ping -c 10 192.168.1.1
3  PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
4  64 bytes from 192.168.1.1: icmp_seq=1 ttl=128 time=14.4 ms
5  64 bytes from 192.168.1.1: icmp_seq=2 ttl=128 time=10.0 ms
6  64 bytes from 192.168.1.1: icmp_seq=3 ttl=128 time=7.25 ms
7  64 bytes from 192.168.1.1: icmp_seq=4 ttl=128 time=17.3 ms
8  64 bytes from 192.168.1.1: icmp_seq=5 ttl=128 time=8.83 ms
9  64 bytes from 192.168.1.1: icmp_seq=6 ttl=128 time=7.42 ms
10 64 bytes from 192.168.1.1: icmp_seq=7 ttl=128 time=8.06 ms
11 64 bytes from 192.168.1.1: icmp_seq=8 ttl=128 time=7.87 ms
12 64 bytes from 192.168.1.1: icmp_seq=9 ttl=128 time=8.40 ms
13 64 bytes from 192.168.1.1: icmp_seq=10 ttl=128 time=8.06 ms
```

```
14
15 --- 192.168.1.1 ping statistics ---
16 10 packets transmitted, 10 received, 0% packet loss, time 9029ms
17 rtt min/avg/max/mdev = 7.251/9.784/17.397/3.227 ms
```