How to approach Ryu SDN controller programming

Franco Callegati – Chiara Grasselli

LAB. OF NETWORK PROGRAMMABILITY AND AUTOMATION - PROGRAMMABLE NETWORKING

A.Y. 2024/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Goals

The goal is NOT to teach you how to program in Python

– Different SDN controller framework (might) leverage different programming languages (e.g.: Ryu – Python, ONOS & FloodLight - Java, etc.)

The goal is to provide basic guidelines about how to approach Ryu app controller logic programming whenever you have to write your own control application.

Ryu custom apps are based on *appManager.RyuApp*

RyuApp is the base class of Ryu applications

A custom app will be a Python class that:

- extends RyuApp
- defines its own variables/methods
- registers callback to handle events to which we are interested in

Controller logic is based on **event-handler** programming

Example

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller import dpset
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from threading import Timer
import os
import json
import time
import datetime

'''
This Controller (basic_openstack_l3_controller.py) is intended for OpenStack L3 scenario, assuming to have just 2 Users
'''

class BasicOpenStackL3Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'dpset': dpset.DPSet}

    def __init__(self, *args, **kwargs):
        super(BasicOpenStackL3Controller, self).__init__(*args, **kwargs)
        self.dpset = kwargs['dpset'] #NOTE. dpset (argument of kwargs) is the name specified in the contexts variable
        #VARIABLES
        self.switch_dpid_name = {} #Keep track of each switch by mapping dpid to name
        self.connections_name_dpid = {} #Keep track name and connection

    # FUNCTIONS
    def _get_ports_info(self, dpid): #Return information about all port on a switch
        return self.dpset.get_ports(dpid)

    def _get_port_name(self, dpid, port): #Return the name associated to the specified port number on the specified switch
            return self.dpset.get_port(dpid, port).name

    #Handle reception of StateChange message (NOTE. the message is sent whenever a switch performs the handshake with controller)
    @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def state_change_handler(self, ev):
        datapath = ev.datapath
```

Import required Ryu/Python modules

State your class extending RyuApp, and define your own variables/methods (if needed)

Register callback to handle events to which you are interested in

Event Handlers

- Each event is handled by an event-handler function
- Event handlers
  - Start with @
  - Are "decorated" by specifying the event to handle and the context (dispatcher) to which the handler applies

- *@set_ev_cls(OpenFlowEvent, DISPATCHER)*

event

context

Events

- By convention, OpenFlow events start with ofp_event.EventOFPxxx
  - xxx is the name of the corresponding OpenFlow message.
- Some examples
  - ofp_event.EventOFPStateChange: for state changing messages
  - ofp_event.EventOFPPortStateChange: switch port state chainging messages

- Event classes can be found at

https://ryu.readthedocs.io/en/latest/ryu_app_api.html#event-classes

Dispatchers

A dispatcher represents a state, a specific phase during interaction with switches.

Dispatchers:

- **HANDSHAKE_DISPATCHER**: waiting for/sending HELLO messages
- **CONFIG_DISPATCHER**: post handshake phase, waiting for/sending CONFIG messages
- **MAIN_DISPATCHER**: post config phase, waiting for/sending messages from/to switches
- **DEAD_DISPATCHER**: disconnection from switches (also in case of errors)

Start with a simple example (1)

Event on data path

Attributes:

- Data path number
- Whether it is connected
- Which are the available ports on the switch

```
@set_ev_cls(dpset.EventDP, MAIN_DISPATCHER)
def _dp_event_handler(self, ev):
    datapath = ev.dp
    isconnected = ev.enter
    portlist = ev.ports

    dpid = datapath.id
    if isconnected:
        self.logger.info("%s +++ New datapath connected to controller - dpid:%s", datetime.now(), dpid)
        self.logger.info("    Port list:")
        for p in portlist:
            self.logger.info("    Port no. %s (%s) - MAC address: %s", p.port_no, p.name, p. hw_addr)
    else:
        self.logger.info("%s --- Datapath disconnected from controller - dpid:%s", datetime.now(), dpid)
    self.logger.info("")
```

This is the event

Event attributes are copied to local variables

ryu.controller.dpset.EventDP

class `ryu.controller.dpset.EventDP`(*dp, enter_leave*)

An event class to notify connect/disconnect of a switch.

For OpenFlow switches, one can get the same notification by observing ryu.controller.ofp_event.EventOFPStateChange. An instance has at least the following attributes.

Attribute	Description
dp	A ryu.controller.controller.Datapath instance of the switch
enter	True when the switch connected to our controller. False for disconnect.
ports	A list of port instances.

Start with a simple example (2)

This event handler prints information about the data path:

- Whether it is conncted or disconnected
- The list of ports in the switch when it is connected

```
@set_ev_cls(dpset.EventDP, MAIN_DISPATCHER)
def _dp_event_handler(self, ev):
    datapath = ev.dp
    isconnected = ev.enter
    portlist = ev.ports

    dpid = datapath.id
    if isconnected:
        self.logger.info("%s +++ New datapath connected to controller - dpid:%s", datetime.now(), dpid)
        self.logger.info("    Port list:")
        for p in portlist:
            self.logger.info("    Port no. %s (%s) - MAC address: %s", p.port_no, p.name, p. hw_addr)
    else:
        self.logger.info("%s --- Datapath disconnected from controller - dpid:%s", datetime.now(), dpid)
    self.logger.info("")
```

Let’s go further

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
    # store source and destination MAC addresses in local variables
    dst = eth.dst
    src = eth.src
```

An Openflow PacketIn message arrived from the switch on the datapath

This is the packet attached to the PacketIn message

Let’s take the Ethernet header of that packet

And copy the source and destination mac address of the packet into two local variables

mac_to_port

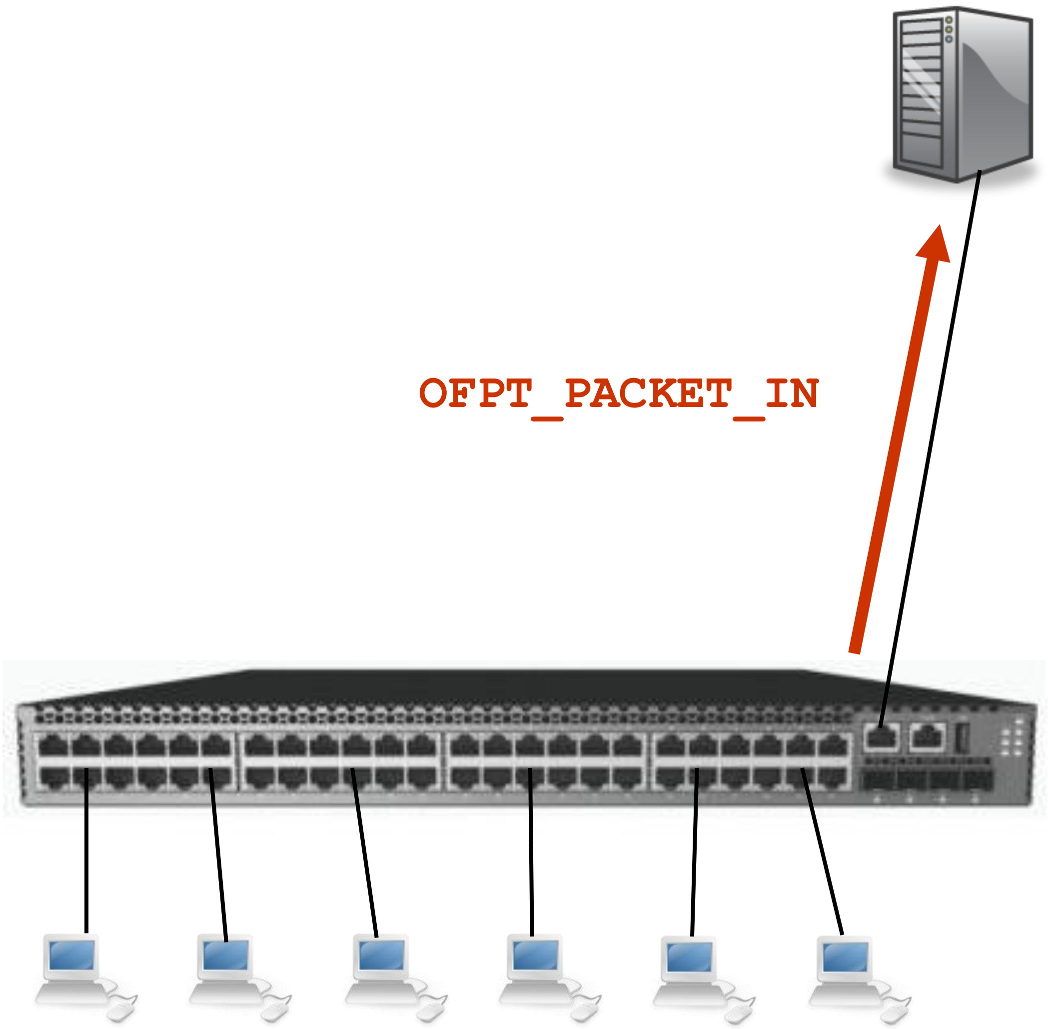
- A dictionary is created and stored in the controller
  - A dictionary is as a set of *key: value* pairs, with the requirement that keys are unique (within one dictionary)
- Create a dictionary : mac_to_port{}
  - Create a dictionary as value *self.mac_to_port.setdefault(dpid, {})*
- Inserting a key into a dictionary
- Inserting a value corresponding to the key
  - *self.mac_to_port[dpid][src] = msg.in_port*

Key

Value

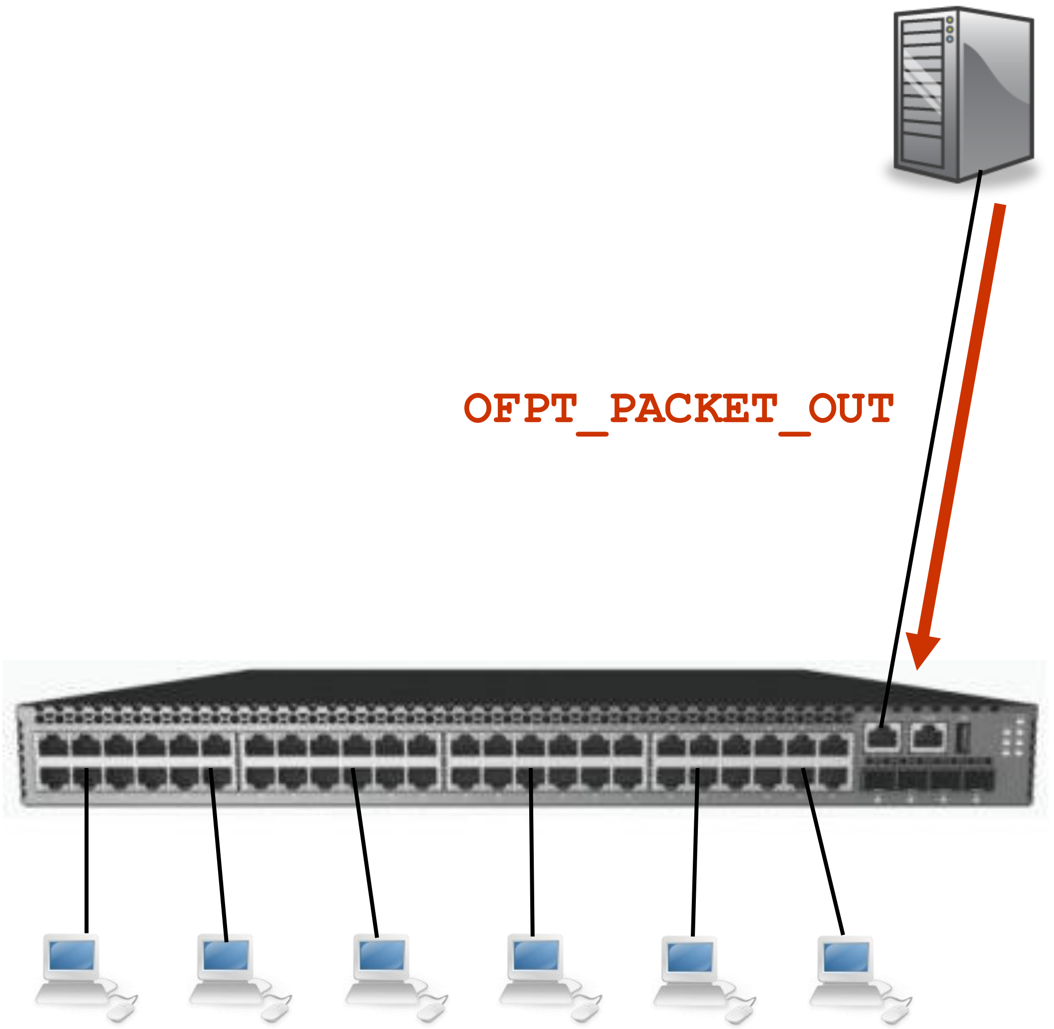
Switch operations

- OFPT_PACKET_IN
  - Asynchronous
- "Send to controller" action set in the action table
  - A **Packet-In** message is sent to the controller
  - The message encapsulates the original message for controller analysis
    - either whole packet
    - or a fraction + buffer ID



Controller response

- OFPT_PACKET_OUT
  - Reply to a Packet-In
- Command to send a packet out of a specified switch port
  - either encapsulates the packet to be sent
  - or specifies the buffer ID



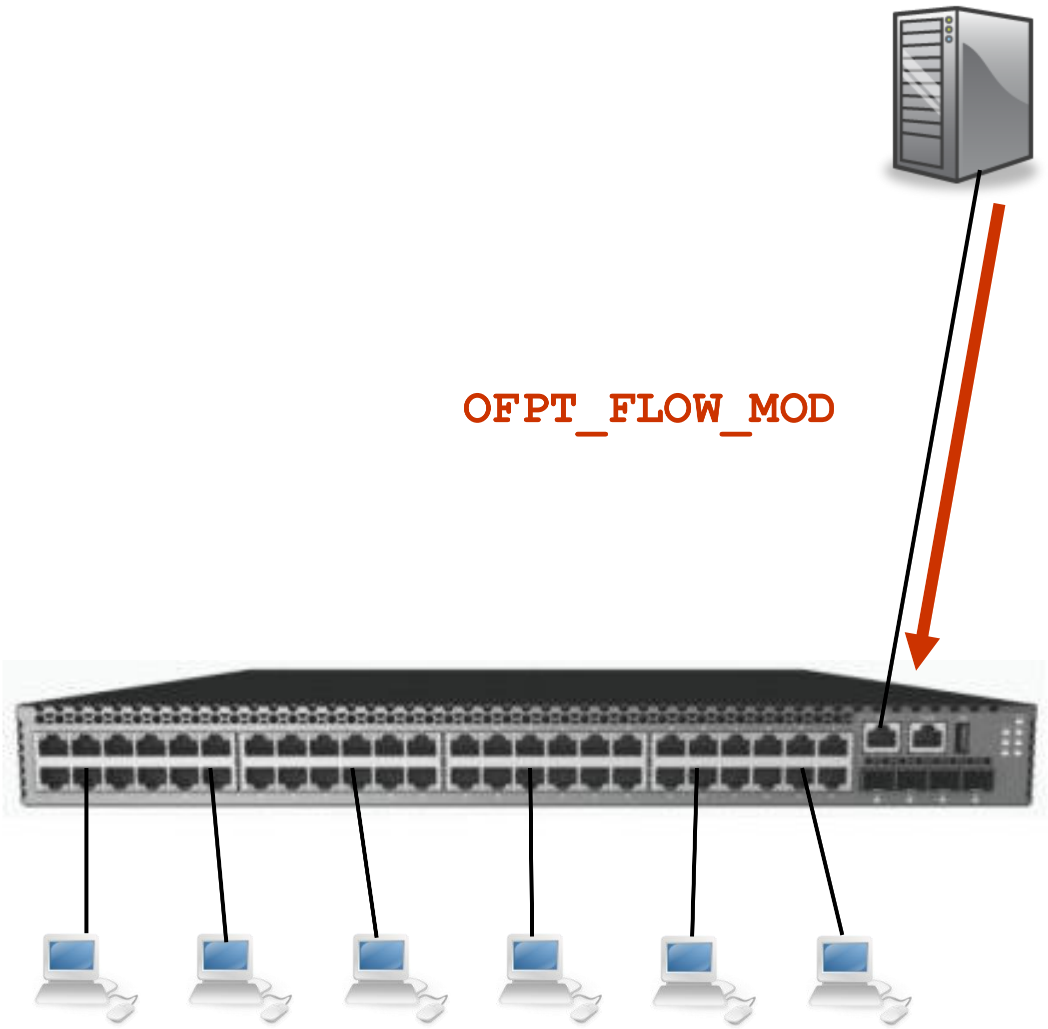
Netprog_no_flows.py

- A simple controller that does not install flow rules
- It is used to experiment the Packet-In – Packet-Out dynamics
    - The controller learns MAC addresses and related ports
    - Every time there is a new packet, the switch sends it to the controller
    - The controller tells the switch to send the packet out from a given port

- All traffic crossing the switch goes to the controller and back
- This raises a performance issue

Make action persistent

- OFPT_FLOW_MOD
  - controller-to-switch
- Command to add, delete, or modify an entry in the flow table
- Includes matching rule, actions, priority, soft and hard timeouts, …
- Makes the packet-out action **persistent** in the switch



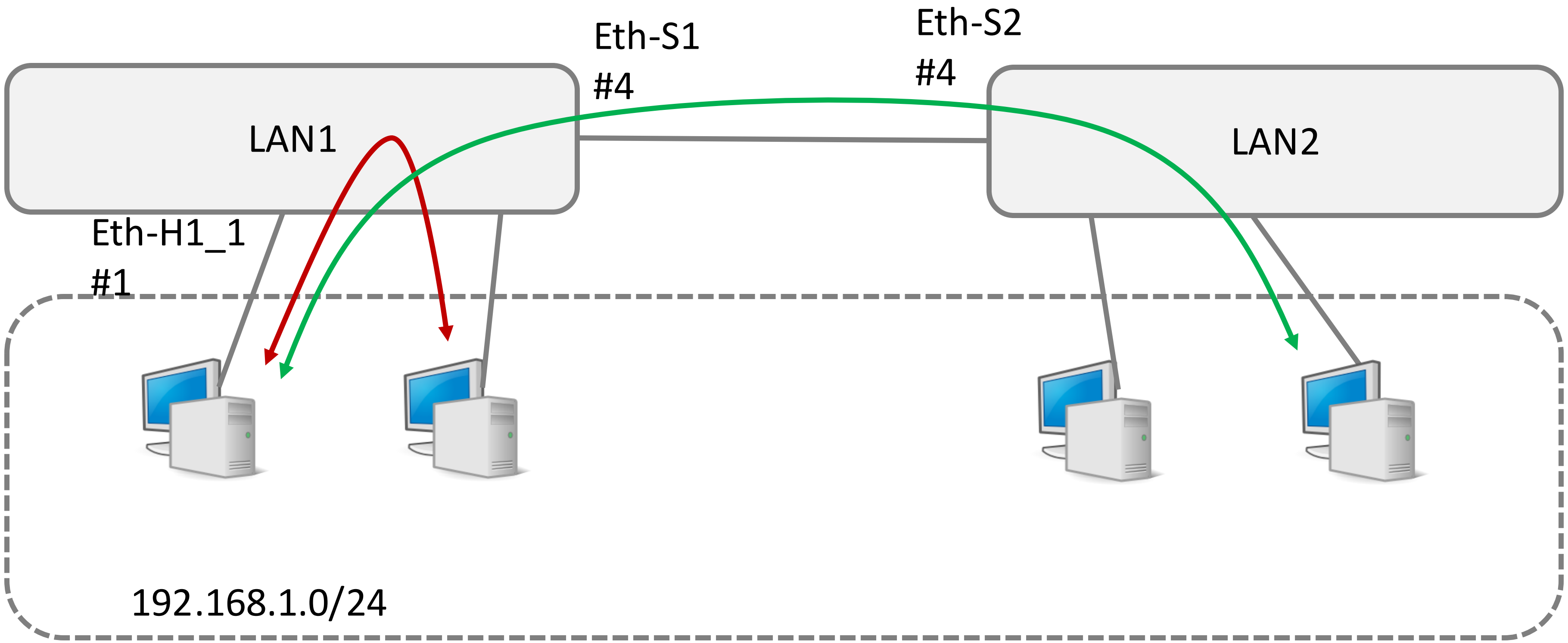
Netprog_src_dst.py

- Example of flow entry installation
- When the packet arrives the controller stores the «source port-mac address» pair
- If the «destination port – mac address pair» is known, it installs the corresponding rule
- Experiment
  - Network with 3 hosts
  - Host 1 pings host 2
  - Host 1 pings host 3
- Check the traffic between switch and controller, and the flow rules installed

Port mirroring

- Purpose
  - Traffic monitoring
  - Specific traffic flows are copied to a port bringing them to a traffic monitoring system
- Local port mirroring
  - Single device
- Remote port mirroring
  - Remote source
  - Remote destination

Port Mirroring: local vs remote



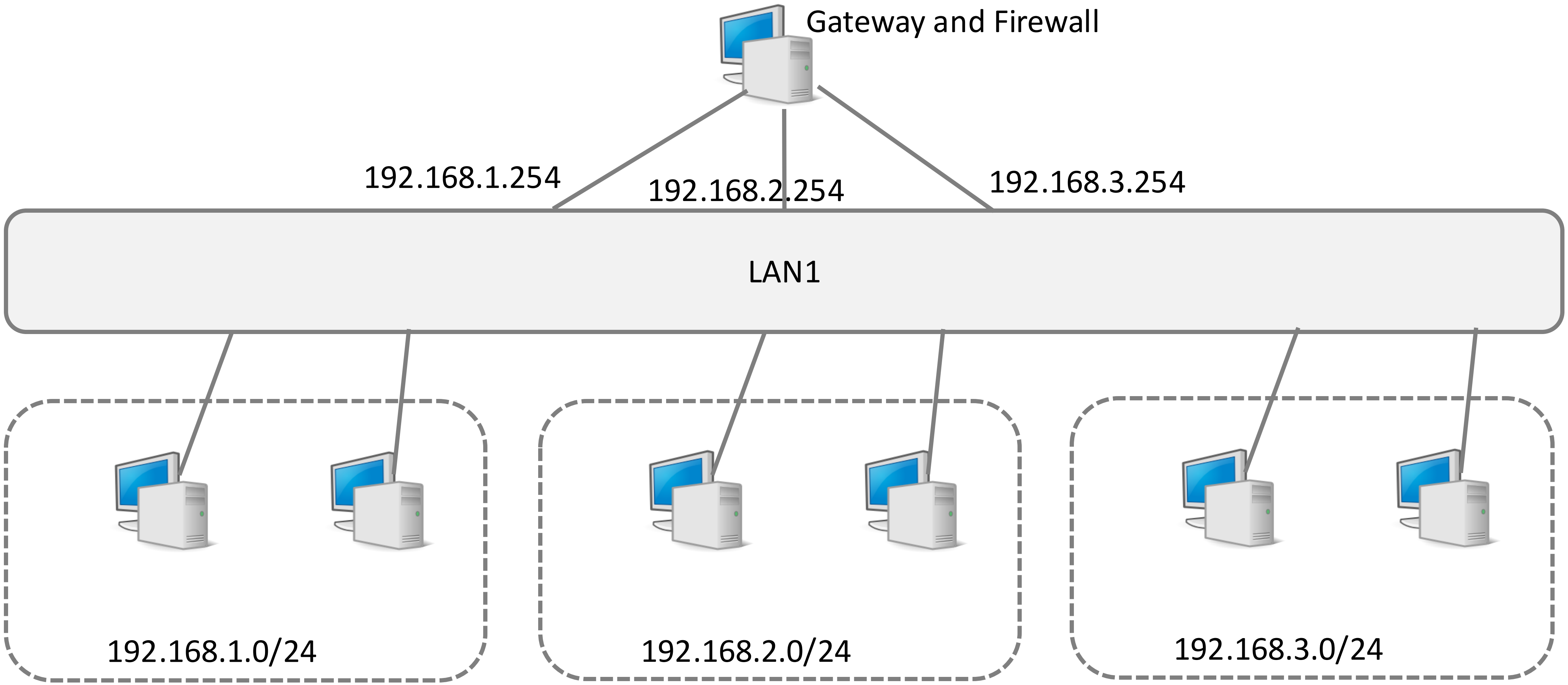
Firewall

- A gateway with special feature
  - In normal gateways:
    - Packets are forwarded depending on the IP forwarding table
    - If a packet has a destination that is not in the forwading table, it is dropped
  - In a firewall
    - Packets are forwarded based on specific rules that depend on some specific characteristic of the packets
- Packet filter
  - Allow or deny forwarding of packets based on IP addresses
- Application level gateway
  - Allow or deny forwarding of packets based on
    - Transport layer port number
    - Application protocol

Typical firewall policies

- Default DENY
  - All traffic is blocked
  - Requires specific rules to ACCEPT (allow) a traffic flow
- Default ACCEPT
  - All traffic can be forwarded
  - Requires specific rules to DENY (block) a traffic flow
- What is a traffic flow?
  - The question is similar to what discussed with the introduction of OpenFlow
  - The answer is the same

Example


Gateway and Firewall
192.168.1.254
192.168.2.254
192.168.3.254
LAN1
192.168.1.0/24
192.168.2.0/24
192.168.3.0/24

Legacy

- The most typical tool to implement a firewall in the Unix/Linux environment is IPTABLES
  - IPTABLES allows us to set specific forwarding rules

- A simple example is applied
  - Block traffic between network 192.168.1.0/24 and network 192.168.3.0/24
    - ip tables –A FORWARD –s 192.168.1.0/24 –d 192.168.3.0/24 –j DROP
    - ip tables –A FORWARD –s 192.168.3.0/24 –d 192.168.1.0/24 –j DROP

SDN Firewall implementation

- Example for default accept policy
  - Implement traffic flow blocking policies
- Stateless
  - Switch port number-based
    - No packets from port i to port j
  - MAC-based
    - Block packets depending on MAC source and MAC destination
  - IP-based
    - Block packets depending on IP source, IP destination, or IP protocol
  - Layer4 port-based
    - Block packets depending on source PORT and destination PORT

Packet serialization

- Create a new variable as a serialized packet
  - The sequence of the bits in the current packet
  - Serialized packets must be parsed to get the header values
- Serialize the incoming packet in variable `pkt`
  - `pkt = packet.Packet(msg.data)`
- Get the ethernet header of the new packet (if the interface is ethernet an ethernet header is mandatory)
  - eth = pkt.get_protocols(ethernet.ethernet)[0]
- Get the IP header
  - The inner protocol after ethernet is not necessarily IP
  - The `ethertype` field says which is the inner protocol

Some ethernet types

Ethernet Protocol Type field (exadecimal) 2 bytes	Protocol
0x8000	IPv4
0x8006	Address Resolution Protocol (ARP)
0x8035	Reverse Address Resolution Protocol (RARP)
0x8100	IEEE 802.1Q VLAN tagged
0x86DD	IPv6
0x88F7	Precision Time Protocol (PTP) over IEEE 802.3 Ethernet

Get IP and transport protocol

- Get the IP packet

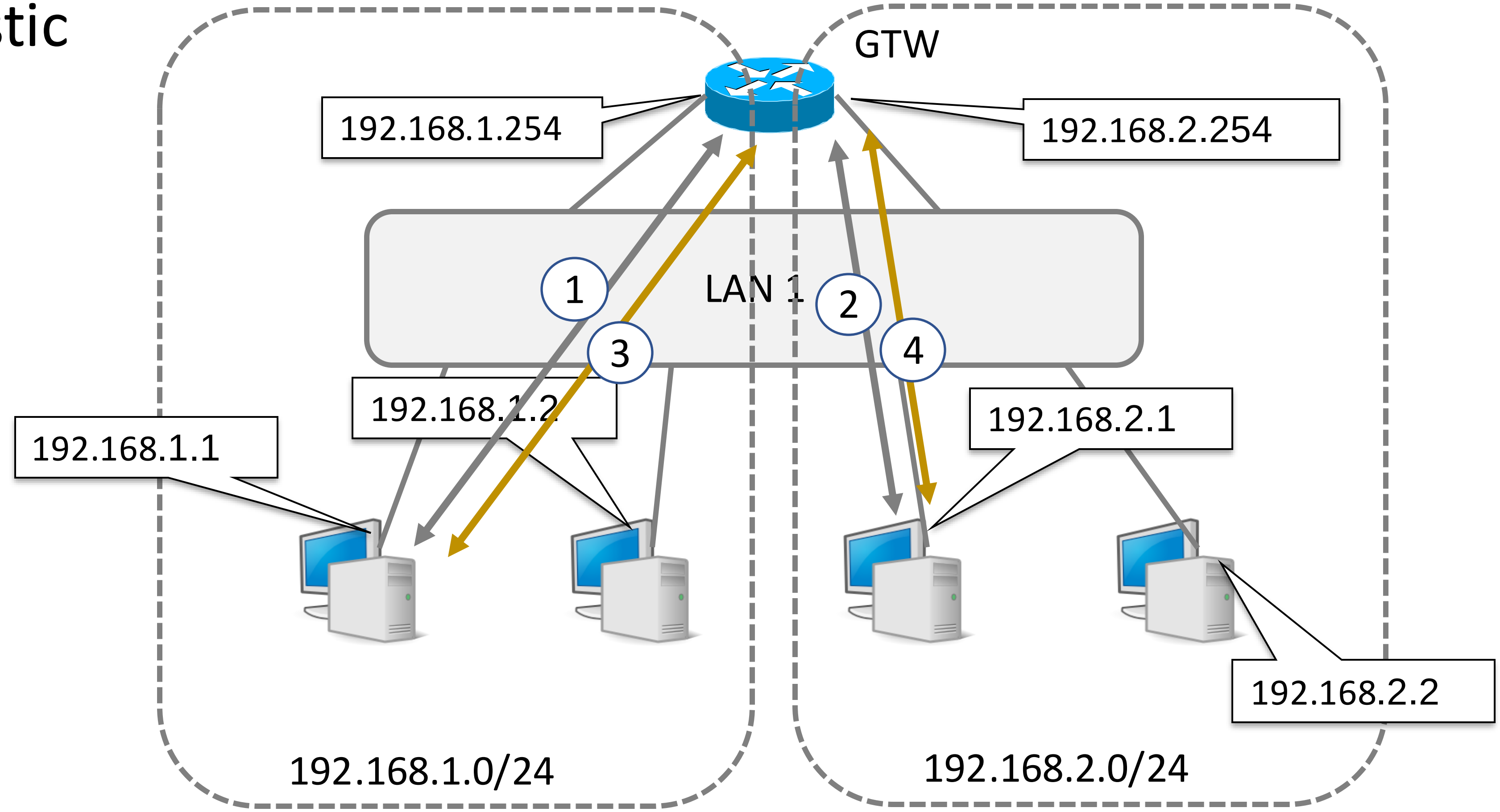
```
if eth.ethertype == ether_types.ETH_TYPE_IP:
    ip_pkt = pkt.get_protocol(ipv4.ipv4)
    ip_dst = ip_pkt.dst # GET IP destination
    ip_src = ip_pkt.src # GET IP source
    ip_proto = ip_pkt.proto # GET inner protocol
```

Some IP inner protocol codes

IP Protocol field (exadecimal) 1 byte	Protocol
0x01	ICMP
0x06	TCP
0x06	TCP
0x11	UDP
0x29	IPv6
0x2F	GRE Generic Routing Encapsulation

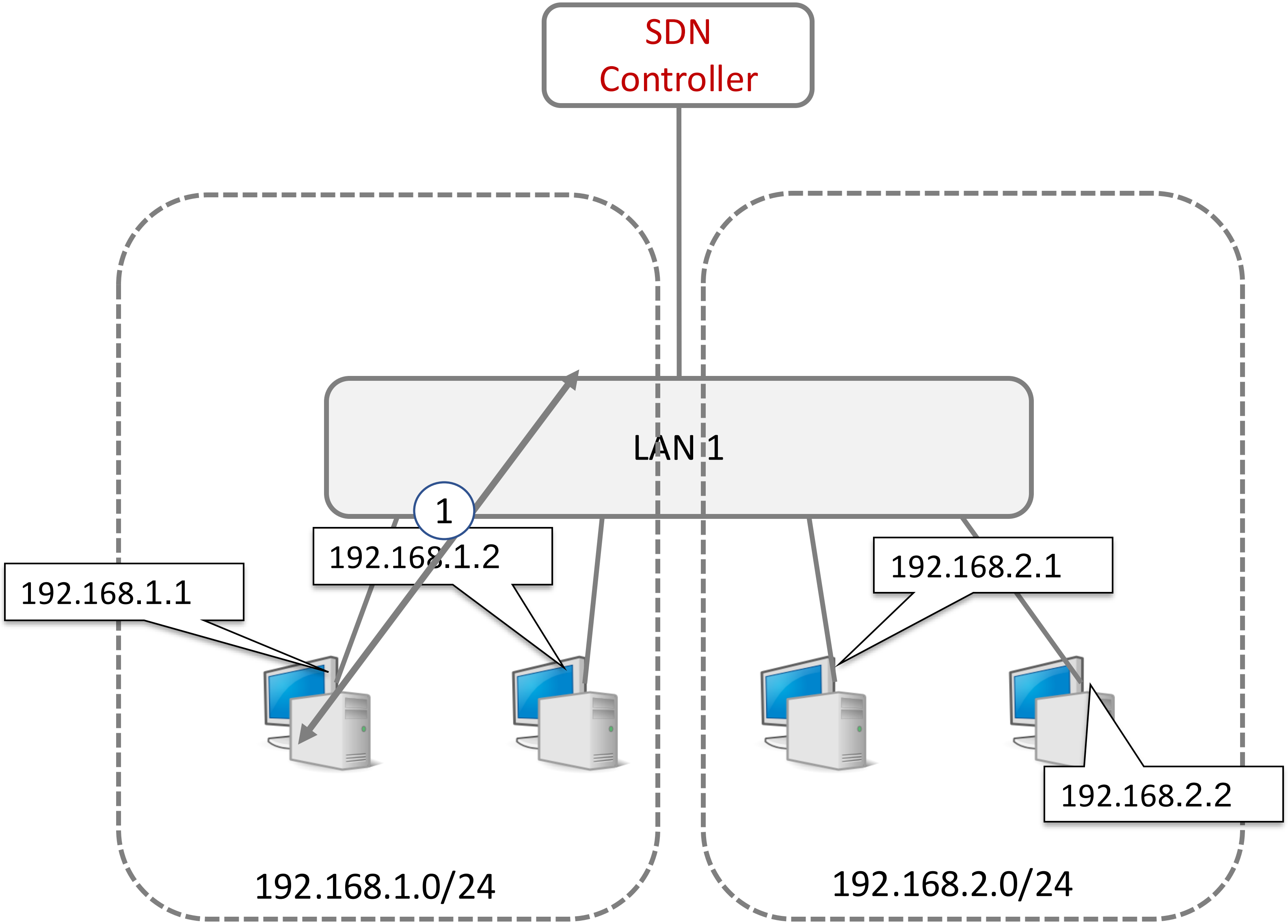
SDN controller as a router

- Let's start from a known topology
- Hosts are SDN agnostic



SDN evolution

- Can we implement the IP network interconnections simply by means of the SDN controller
  - All network functionalities at the edge (in the hosts) must be preserved



Activate the transport layer

- Netcat is a simple application that enables the use of the transport layer on top of IP
  - `nc -l -n 192.168.1.1 7090`
    - Opens trasport port 7090 using the TCP protocol in LISTEN mode (waiting for a call)
  - `nc -n 192.168.1.1 7090`
    - Calls host 192.168.1.1 on transport port 7090 using the TCP protocol
  - Other option
    - `-u` will force the use of the UDP protocol instead of TCP
    - `-p` allow to specify the source port to be used when conecting
    - `-s` allow to use a specific IP address in the messages

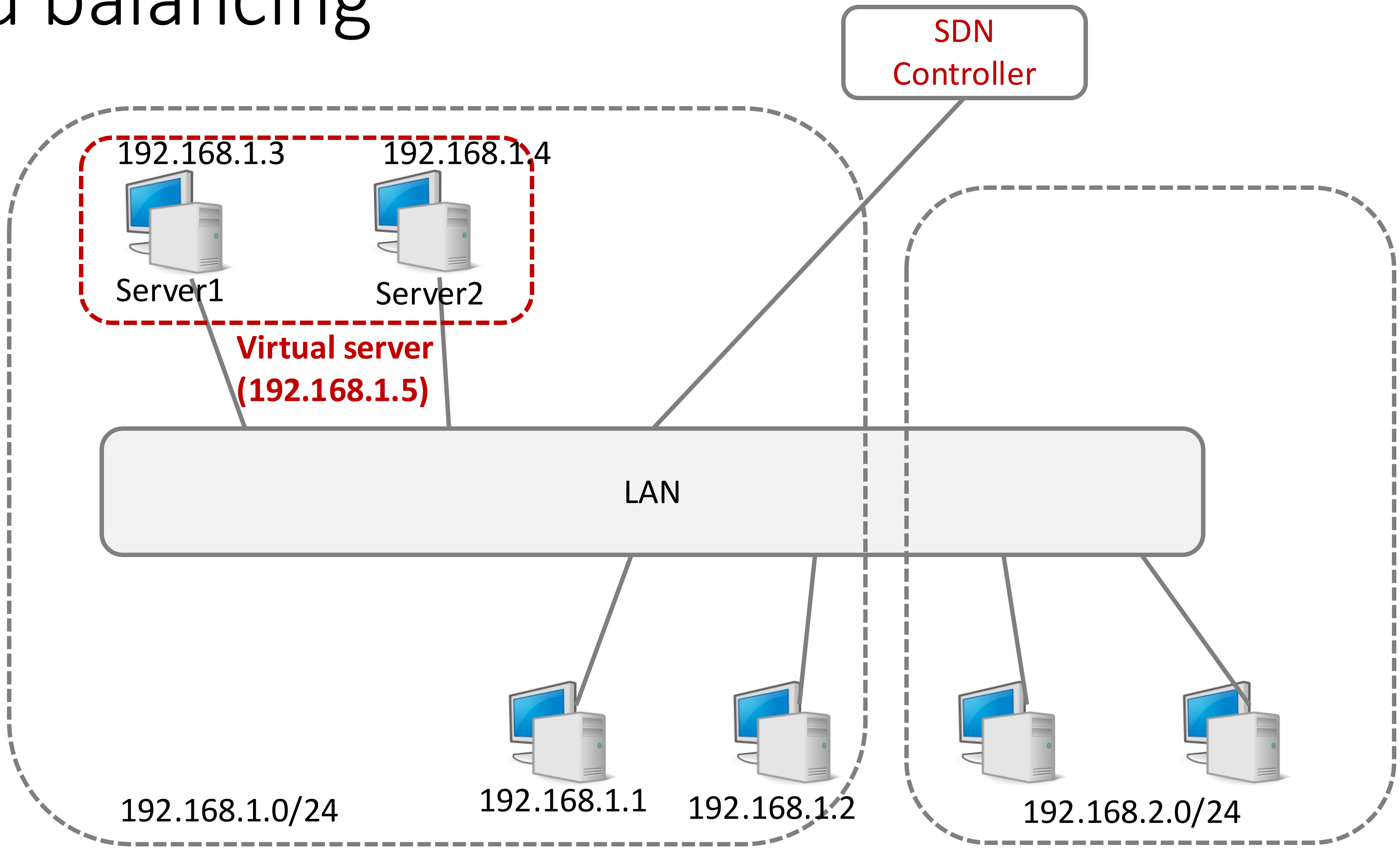
Server Load Balancing

- Goal
  - Distribute high traffic among several servers using a network-based hardware or software-defined appliance
- Implementation
  - Virtual server
    - An IP address is reserved for the logical service end-point
    - The IP address is given to the load balancer that behaves as a gateway towards the physical servers
  - Physical servers
    - Servers running the same service
    - Traffic is routed by the load balancer to one of the physical servers

Load balancing algorithms

- Round robin
  - Requests are distributed to servers in rotation
  - Does not take into account the characteristics and load of the servers
  - Works well when all requests are similar
- Weighted round robin
  - Servers are given a weight and are chosen with a probability that is proportinal to the weight
  - More powerful servers may have a larger weight and are chosen more often
- Least Connection
  - Servers are chosen based on the open connections
  - Servers with less connections open are chosen first

Load balancing



References

To deep dive into Ryu app programming, useful references are:

- https://ryu.readthedocs.io/en/latest/ryu_app_api.html : Ryu programming model
- https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html : reference OpenFlow v1.3 Messages and Structures (other versions are also documented)

Be aware that:

- Other useful references can be found online
- Reverse engineering is pretty time consuming, but can be of help sometimes