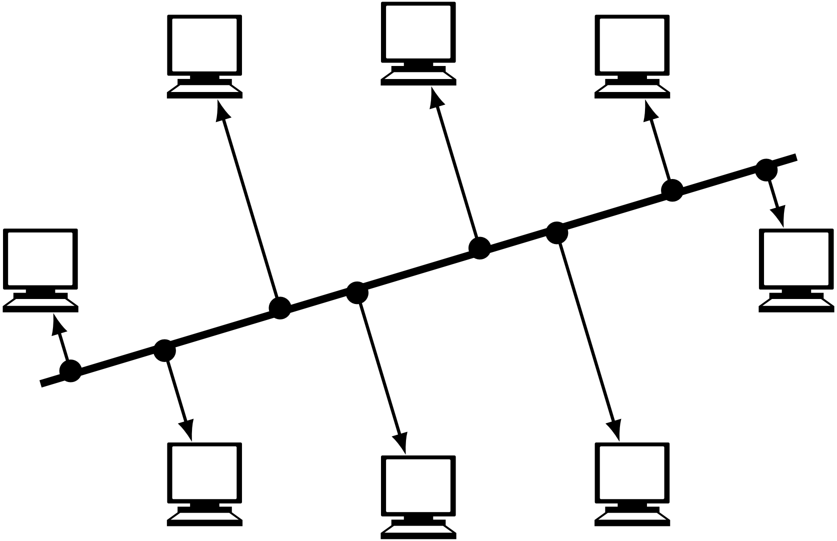
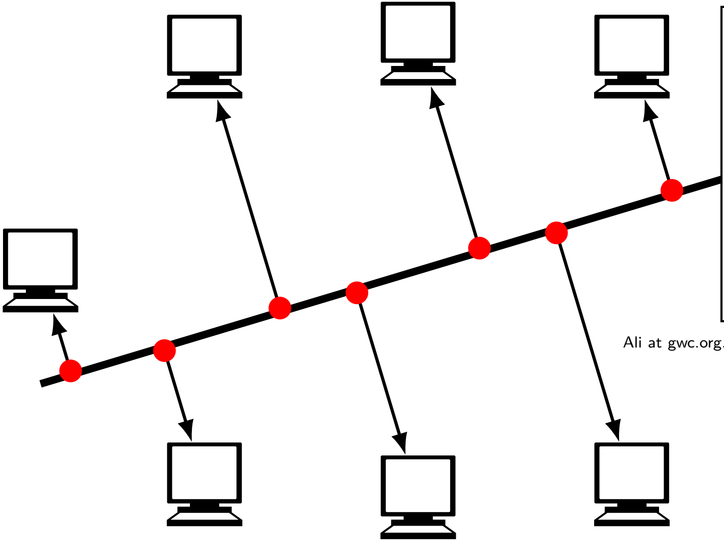
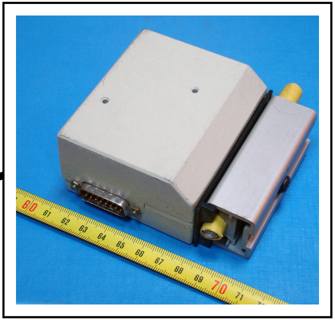
recall: multi-access media

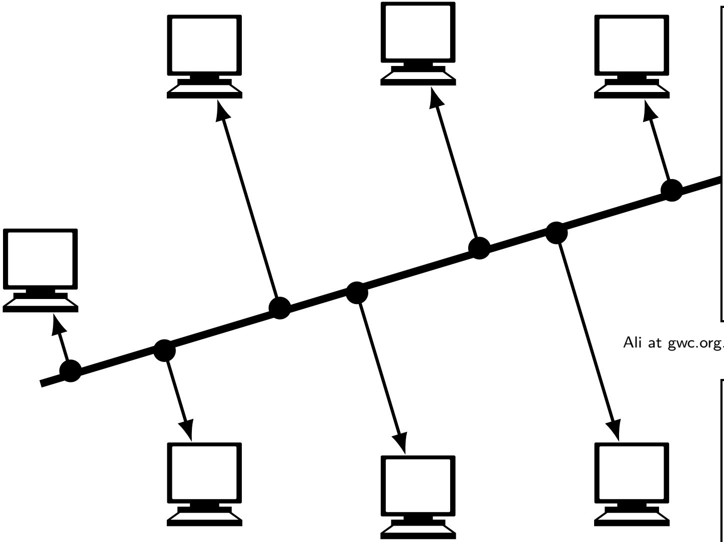


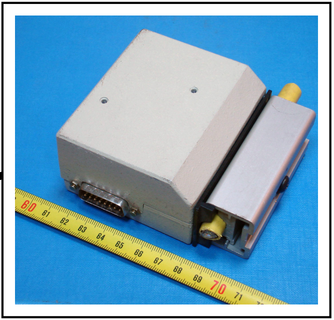
recall: multi-access media



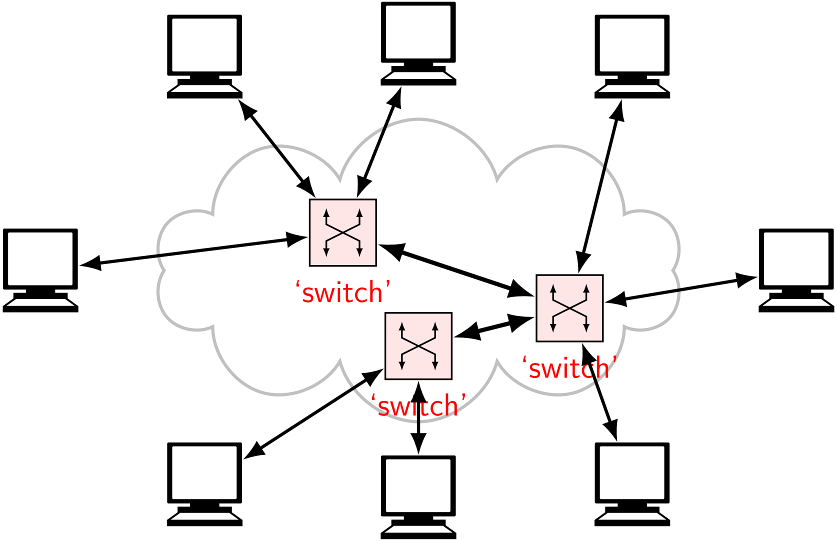


recall: multi-access media

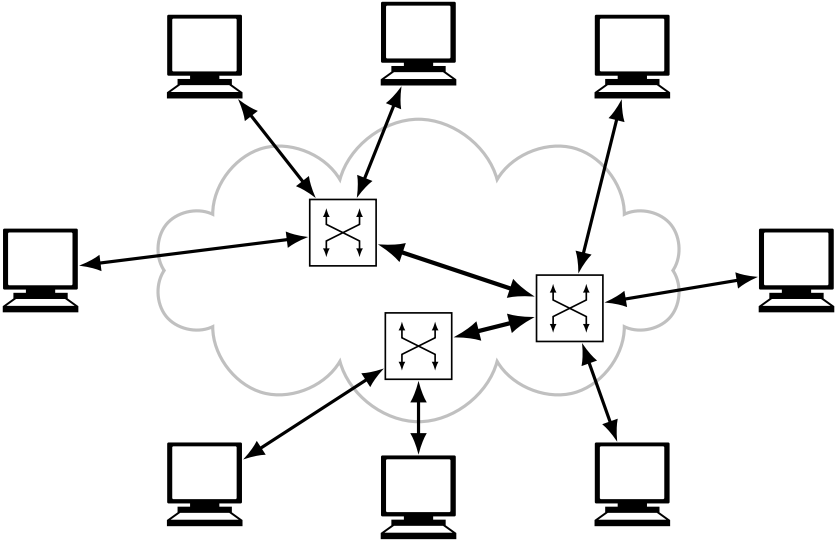




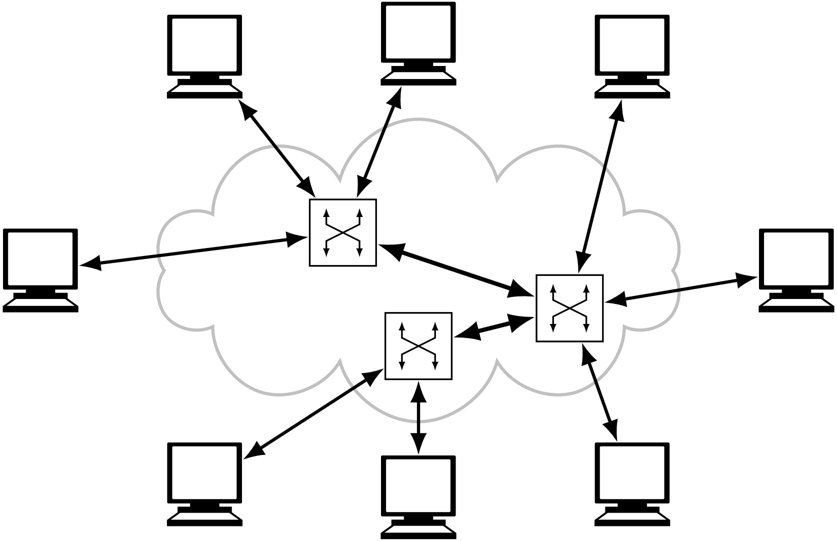
recall: switched network



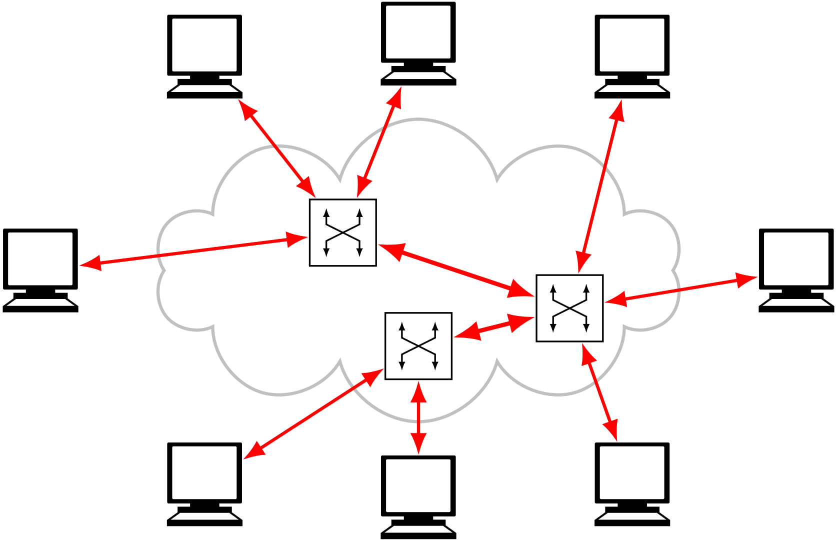
recall: switched network



recall: switched network

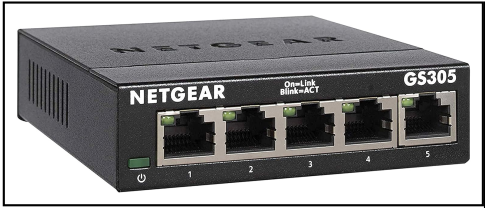


recall: switched network



hubs and switches

Switch



Hub

difference is hidden inside

hub: electrically connects hosts — as if shared wires

switch: decides what to send on each output

history: multi-access to switched

a lot of early networking technology was multi-access

wireless (wifi, cellular) and most home broadband still is

most wired networks are *switched*

- frames mostly directed to correct machine

switching versus routing

switches — forward frames for common network

routers — forward packets between networks

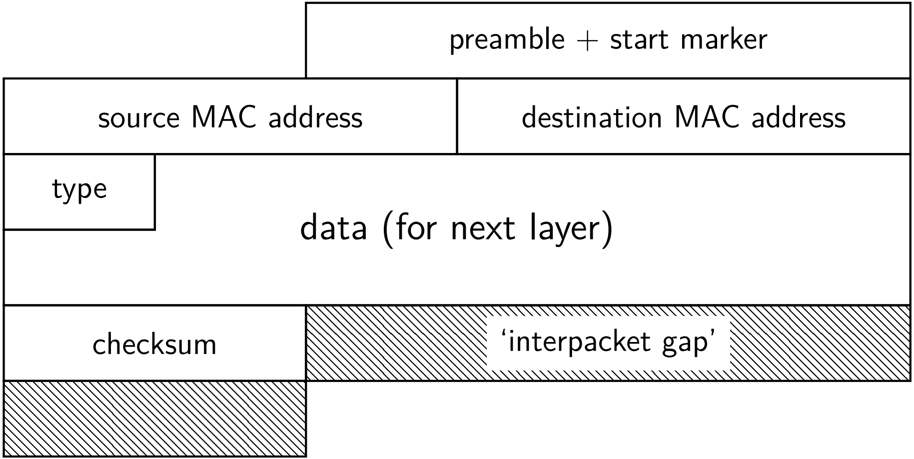
basically same functionality

differences:

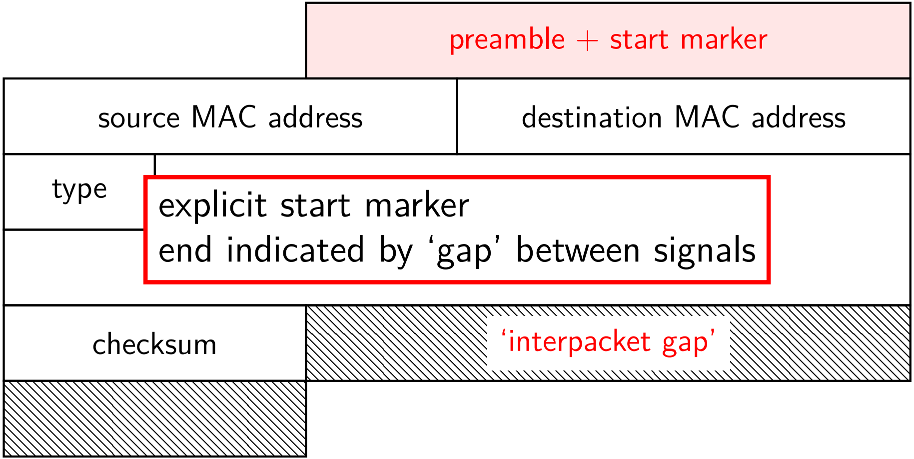
- extra layer for internetwork packets
- different mechanism to decide where to forward
- switch forwarding typically simpler

will start with simpler switching

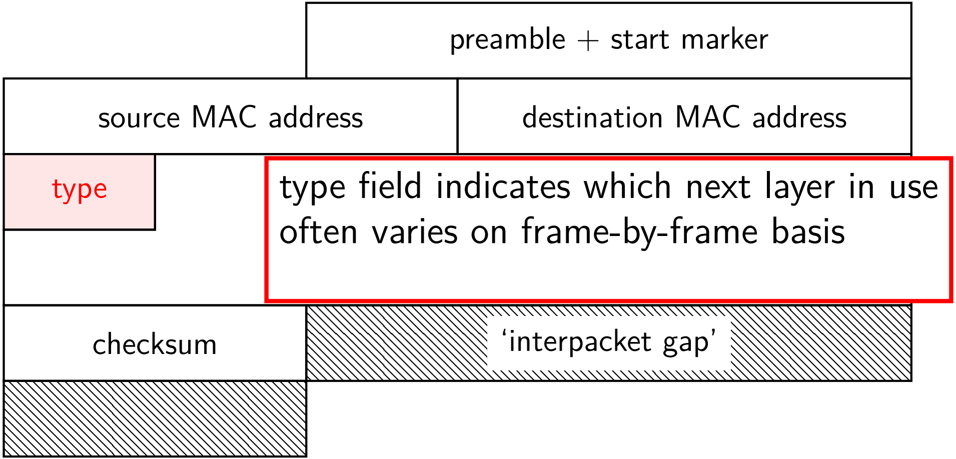
typically sent on Ethernet



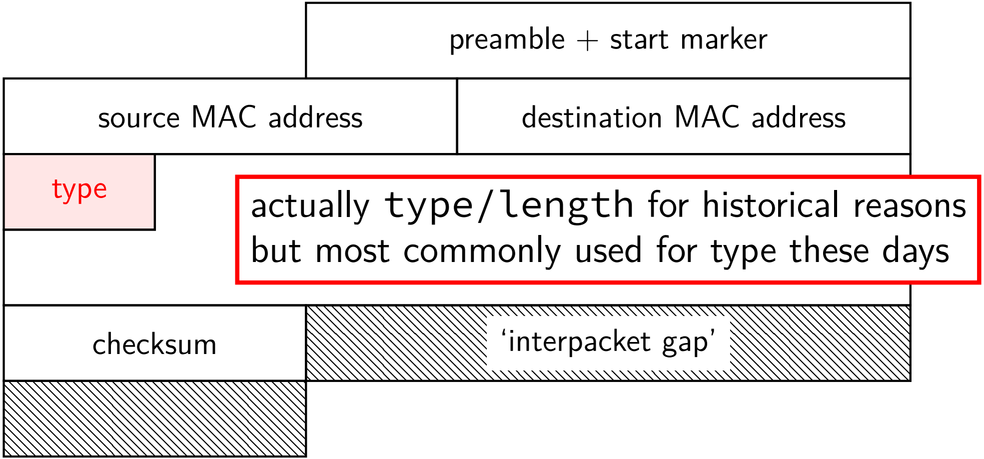
typically sent on Ethernet



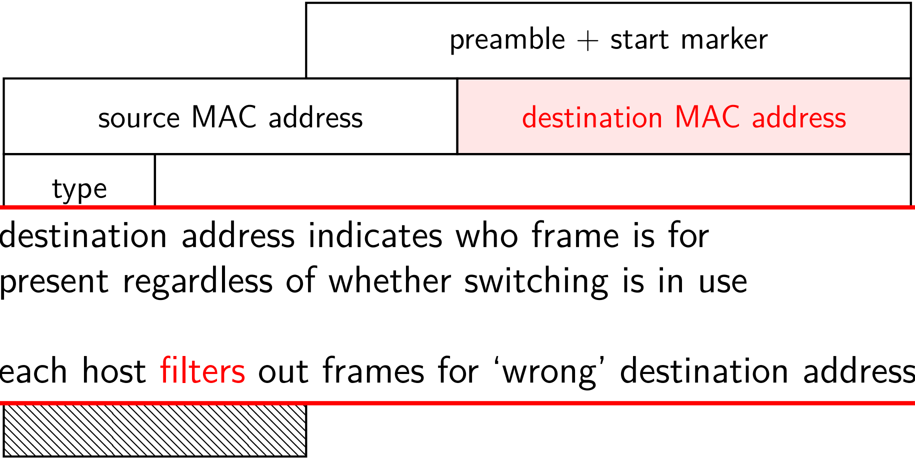
typically sent on Ethernet



typically sent on Ethernet



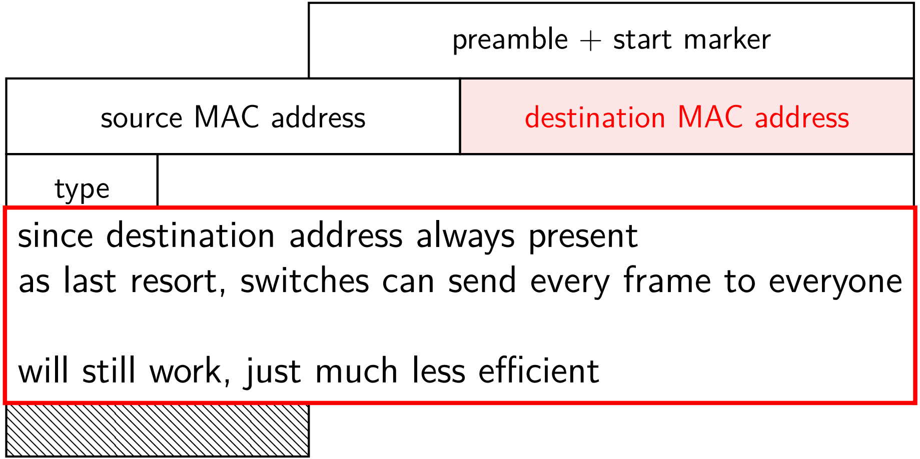
typically sent on Ethernet



destination address indicates who frame is for
present regardless of whether switching is in use

each host filters out frames for ‘wrong’ destination address

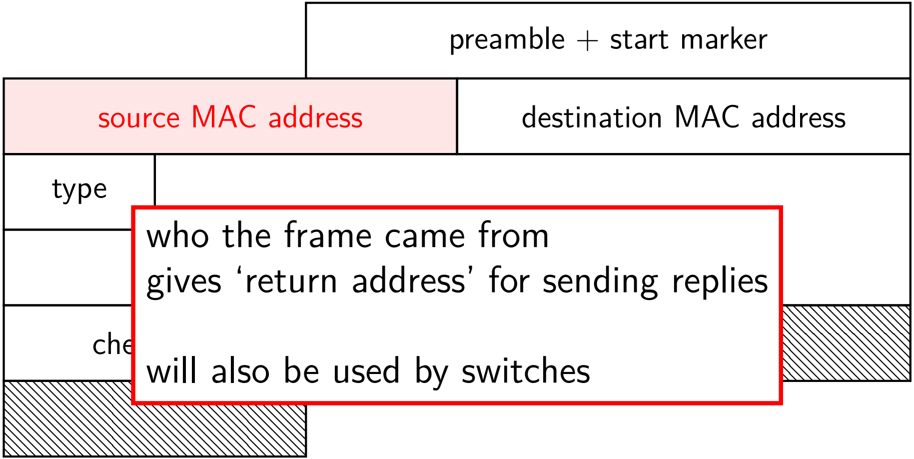
typically sent on Ethernet



since destination address always present
as last resort, switches can send every frame to everyone

will still work, just much less efficient

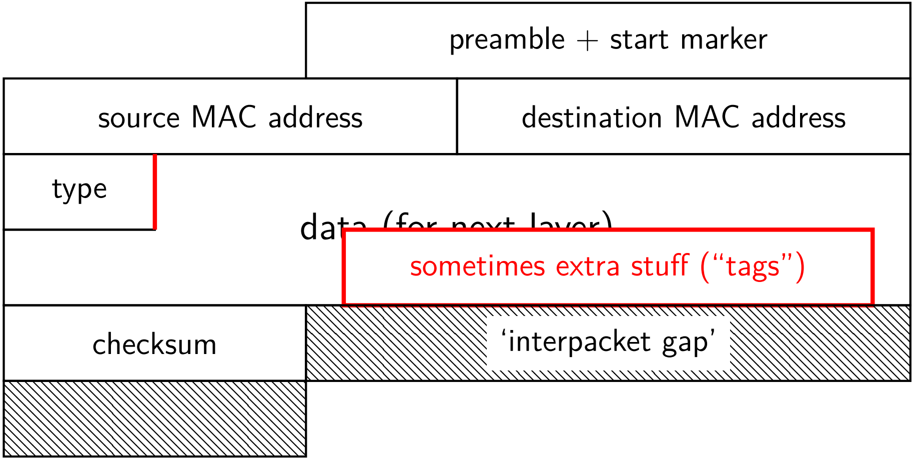
typically sent on Ethernet



who the frame came from
gives 'return address' for sending replies

will also be used by switches

typically sent on Ethernet



MAC addresses

MAC [media access control] addresses

used by Ethernet, Wifi, and lots of other protocols

48-bit number written in hex: `01:23:45:67:89:AB`

(sometimes seperated with `-` instead of `:`)

assigned by IEEE to networking manufacturers in blocks

Institution of Electrical and Electronics Engineers

example: `00:02:B3:`..., `00:03:47:`..., (and many more) for Intel

individual addresses hard-coded in networking hardware

uniquely identify port/device

special MAC addresses

`00:00:00:00:00:00` (all zeroes)

`FF:FF:FF:FF:FF:FF` (all ones), `FF:`...

- special destination meaning "send to everyone" (on this network)
- called *broadcast*

`01:80:C2:`..., `33:33:`..., (and some more)

- special destinations representing multiple receivers
- example: 'all IPv6 routers on this network' (`33:33:00:00:00:02`)
- called *multicast*

larger MAC addresses

IEEE now calls MAC address EUI-48 (48-bit Extended Unique Identifier)

also created EUI-64, with 64-bit addresses
- way of mapping EUI-48s to EUI-64s
- turns out 48 bits might have been low

I'm not sure what status is on switching to 64-bit addresses
- IEEE 802.15.4 (used in ZigBee, 6LoWPAN, some others) uses EUI-64
- I don't know other local network protocols that do

datagram idea

can always send something to anyone on network

- just put destination MAC in frame

no need for reservations/'connections'/etc.

- not like the interface you've seen with [TCP] sockets

not the only model for networks (and internetworks)

virtual circuit

other model: virtual circuit

two machines setup a 'circuit'

- some sort of 'special' messages to do this

switches/routers reserve resources for circuit

- "gaurenteed" bandwidth

transmitted data must be part of estabished circuit

example: ATM (Asynchronous Transfer Mode)

- used (?historically?) by some telephone networks

an annoyance

traditionally, switches/routers have been 'fixed function' specialized hardware

special hardware needed for multigigabit performance

limited configuration options

usually non-automated configuration

- login to each managed switch/router to change settings
- no standardization for configuration across vendors

little visibility into internal design

- even though switches/routers often running complicated programs

the historical situation

let's say I want to design a new extension to Ethernet

historical options if want to test/deploy it...

- implement it in slow/low-capacity software/FPGA switch
- convince switch HW company to implement it
- contort extension to fit with features not intended for use case
  - example: using VPN support to change path of frames on network

software defined networking (SDN)

movement toward *programmable* networks

“software-defined”

rules about how network works defined in “normal” software

control plane and data plane

control plane

decides *how* to handle traffic
"slow path", where complicated decisions are

data plane

actually implements the decisions made by the control plane
"fast path", implementing simple rules

probably what switches did internally before SDN was a thing

separate control/data plane

one SDN key idea: separate control and data plane

allow new vendor-neutral implementations of control plane

- requires standard interface for programming data plane
- most prominent example: OpenFlow

easily allows for central ‘control plane’ server

- instead of separate control plane running on each switch/router

separate control/data plane

one SDN key idea: separate control and data plane

allow new vendor-neutral implementations of control plane

- requires standard interface for programming data plane
- most prominent example: OpenFlow

easily allows for central ‘control plane’ server

- instead of separate control plane running on each switch/router

P4

P4 — programming langauge for data planes

intended to be compiled to run on fast switches

includes ‘runtime’ defining how control plane configures data plane

future P4 assignment

given:

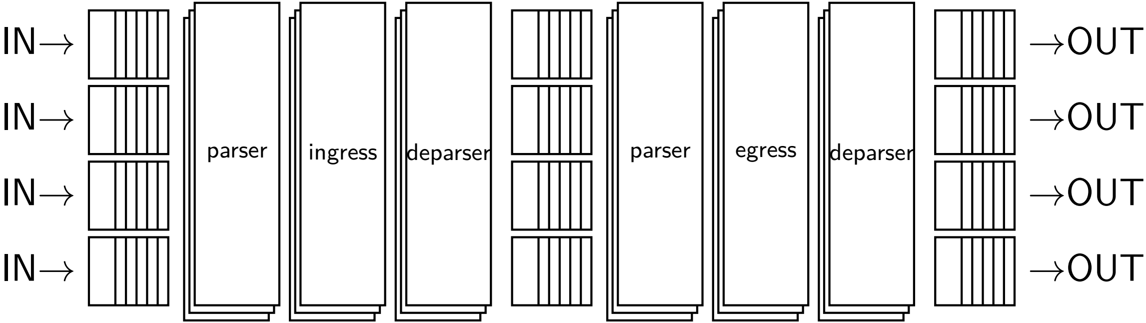
- simple P4 switch that doesn't know where to direct frames
- simpler controller (in Python) that configures switch
- simulated 4-machine network in VM

your task will be:

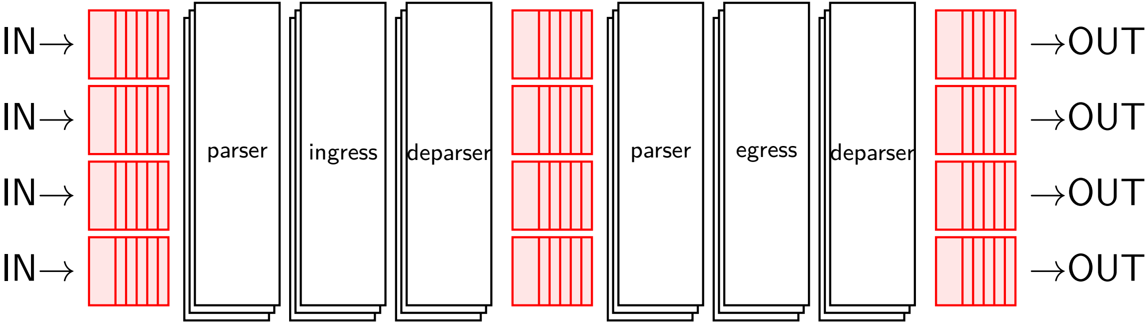
- have controller write static configuration to direct to right place
- modify data plane to send information to control plane
- have controller change configuration based on info from data plane

(we'll discuss more details later)

P4 switch architecture

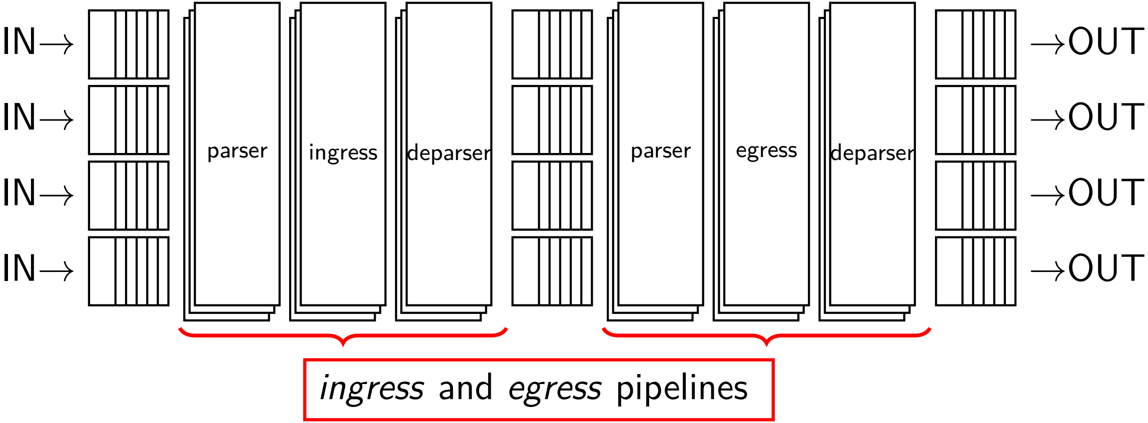


P4 switch architecture

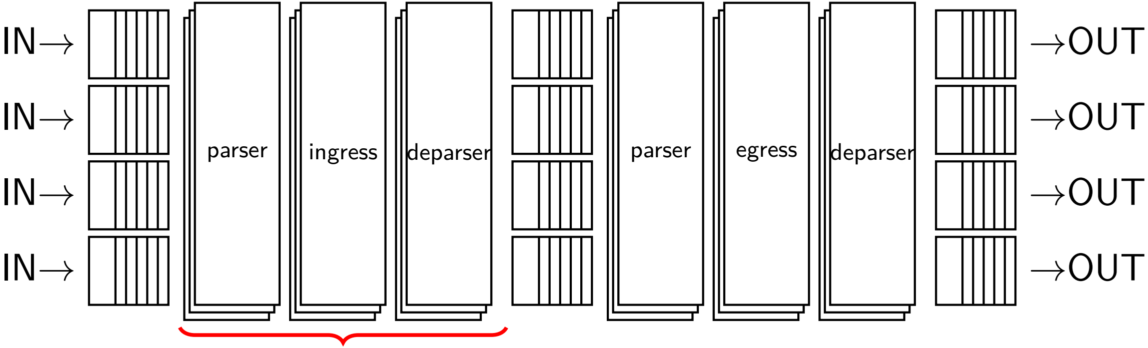


queues of frames to be processed
if needed, hold packets in between processing steps

P4 switch architecture

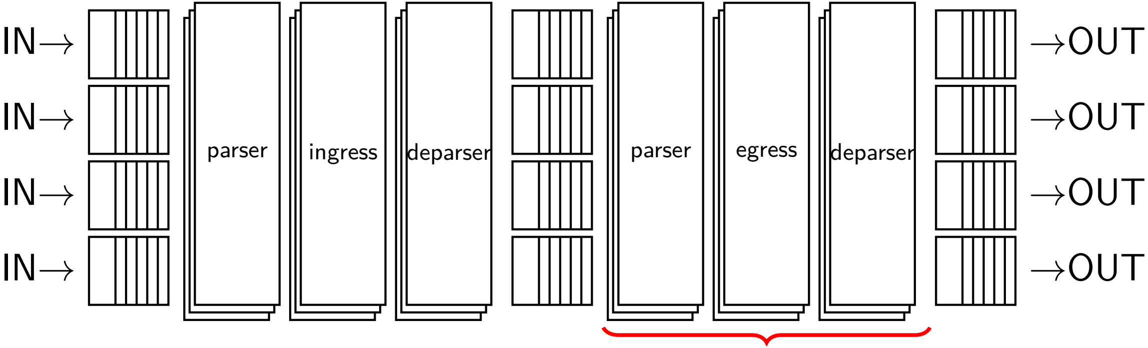


P4 switch architecture



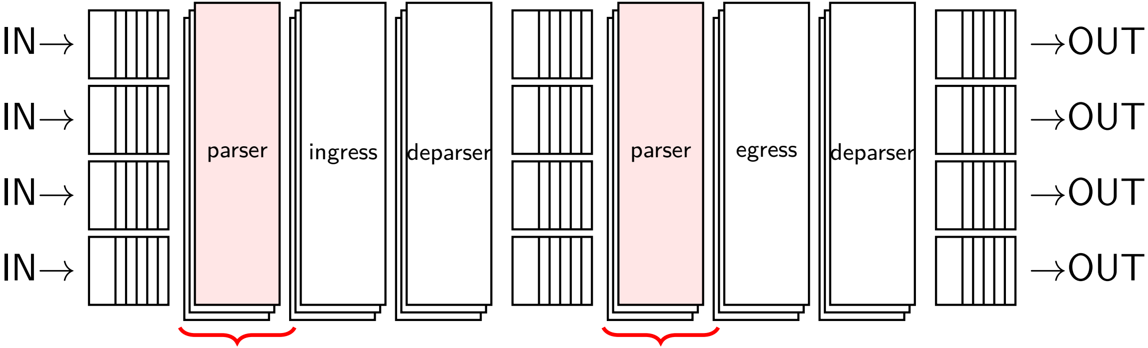
ingress pipeline: processes every input frame
primary job: decide where to forward frame (if anywhere)

P4 switch architecture



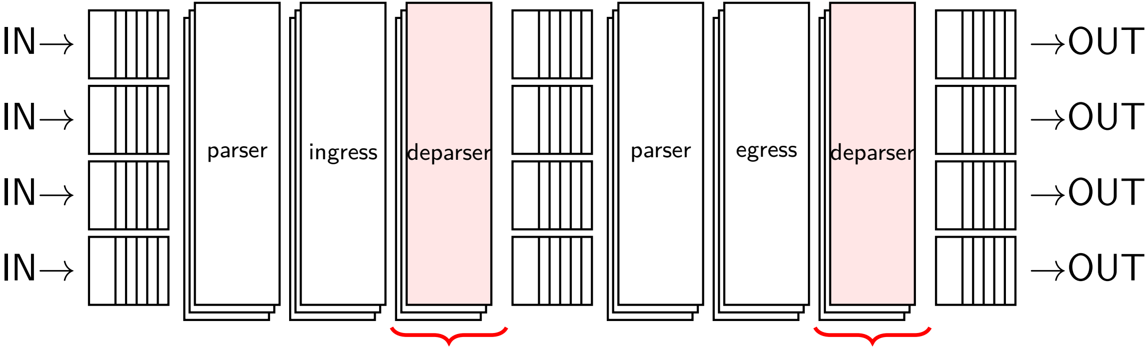
egress pipeline: process every output frame
run one time for each copy output

P4 switch architecture



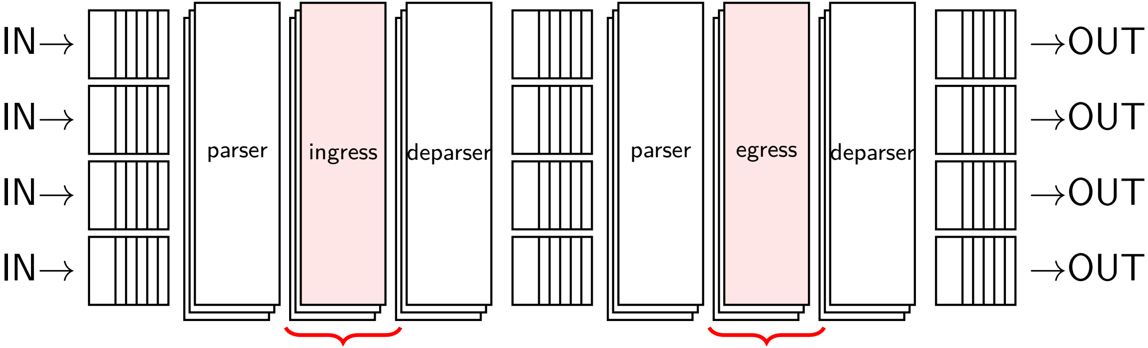
parser: decode frame fields into temporary storage

P4 switch architecture



deparser: converted decoded fields back into bytes for network

P4 switch architecture



"match/action pipelines"
do series of table lookups based on parsed fields
each table lookup specifies next action

P4: switch parts

```
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

code to declare instance of dataplane from functions for each part

will look at most of these components

won't have reason to change verify/compute checksum

P4: declaring headers (1)

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t srcAddr;
    macAddr_T dstAddr;
    bit<16>   etherType;
}
```

P4: declaring headers (1)

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t srcAddr;
    macAddr_T dstAddr;
    bit<16>   etherType;
}
```

kinda like `typedef struct { ... } ethernet_t;`

P4: declaring headers (1)

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t srcAddr;
    macAddr_T dstAddr;
    bit<16>   etherType;
}
```

kinda like `typedef struct { ... } ethernet_t;`

order needs to *exactly* match what's sent on network

- switch will do bit-by-bit copy into this struct

P4: declaring headers (2)

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    ipv6_t       ipv6;
}
```

P4: declaring headers (2)

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    ipv6_t       ipv6;
}
```

struct of all possible headers

- from all layers the switch/router handles
- not all frames will use all of them, some may be mutually exclusive
- order does not need to match frame storage (will write code to handle that)

P4: declaring headers (2)

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    ipv6_t       ipv6;
}
```

struct of all possible headers

- from all layers the switch/router handles
- not all frames will use all of them, some may be mutually exclusive
- order does not need to match frame storage (will write code to handle that)

references header types defined previously

P4: parsing

```
parser MyParser(
    packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {
```

function modifies 'out' parameters instead of returning

parser function — takes parameters:

- packet — the input frame
- headers — extracted headers (`struct headers`)
- meta — program's local variables about frame (`struct metadata`)
- standard_metadata — info for system about frame
  - where frame came from, where it should go, etc.

P4: parsing

```
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            TYPE_IPV6: parse_ipv6
            default: accept;
        }
    }
    ....
}
```

P4: parsing

```
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.
        transit                        Type) {
            TYP
            TYPE_IPv6: parse_ipv6
            default: accept;
        }
    }
    ....
}
```

functions written as state machine
exectuion starts in state `start`
follows instructions in each state

P4: parsing

```
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
```

transition statements say which state to go to next
can be conditional using switch-statement-like syntax

```
        }
    }
    ....
}
```

P4: parsing

```
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
```

transition statements say which state to go to next
can be conditional using switch-statement-like syntax

```
        }
    }
    ....
}
```

P4: parsing

```
parser MyParser(...) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            T
            d
        }
    }
    ....
}
```

extract operation copies from packet into specific header object

exercise: header representation (1)

let's say packet format is:

- 16 bit source address
- 16 bit destination address
- 1 bit flag "is timestamp present"
- 1 bit flag "is data present'
- 6 bits unused
- (optional) 32-bit timestamp
- (optional) 16-bit data size
- (optional) data

how to represent this?

- how many structs?
- what's in the `header {...}`?

exercise: header parsing

```
struct basic_hdr_t {
    bit<16> src; bit<16> dst;
    bit<1> has_timestamp; bit<1> has_data; bit<6> unused;
};
struct timestamp_t {
    bit<32> timestamp;
}
struct data_hdr_t {
    bit<16> size;
}
```

what states/transitions in parser?

states/transitions

parse_basic:

- has_timestamp: to parse_timestamp:
- has_data: to parse_data
- otherwise: accept

parse_timestamp:

- has_data: to parse_data:
- otherwise: accept

parse_data: then accept

P4: ingress processing

example: send *everything* to output port number 4

```
control MyIngress(...) {
    apply {
        standard_metadata.egress_spec = 4;
    }
}
```

P4: ingress processing

example: send *everything* to output port number 4

```
control MyIngress(...) {
    apply {
        standard_metadata.egress_spec = 4;
    }
}
```

standard_metadata — 'hard-coded' phase reads `egress_spec`

special `egress_spec` value for "discard frame"

switch decisions

most decisions switches/routers make are table lookups

take field from packet (e.g. destination)

lookup in table what to do with that (e.g. send to port X, throw error, etc.)

P4 has special syntax for tables and table lookups

P4: tables

```
table mac_dst_exact {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        drop,
        forward_to,
    }
    size = 1024;
    default_action = drop();
}
```

P4: tables

```
table mac_dst_exact {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        drop,
        forward_to,
    }
    size = 1024;
    default_action = drop();
}
```

declare a table with a name
done has part of ingress/outgress

P4: tables

```
table mac_dst_exact {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        drop,
        forward_to,
    }
    size = 1024;
    default_action = drop();
}
```

key/value structure
keys usually from headers
values are actions to run

P4: tables

```
table mac_dst_exact {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        drop,
        forward_to,
    }
    size = 1024;
    default_action = drop();
}
```

here, each table entry contains whole address
P4 also supports partially matched keys

P4 table key types

exact — full entry

lpm — longest prefix match

- table entries contain 'prefixes' of header field(s)
- we'll see motivation for this when we talk about routing
- if multiple entries match, take longest one

ternary — 0/1/don't care

- table entries contain key and mask
- entry matches if bits included in mask match
- other view: keys represented with 0/1/don't care 'trits'

content-addressable memory

high-end routers/switches have *content addressable memory* (CAM)

...including *ternary content addressable memory* (TCAM)

= hardware that implements a table lookup

- 'content' ≈ key being looked up
- looks kinda like highly associative cache
- probably lots of comparators

allows multigigabit frame processing speeds

P4 goal: P4 programs compile to use CAM/TCAM when available

aside: TCAM cost

2.7x more transistors than SRAM (common cache technology)

probably much more power than SRAM

hard to find recent quotes for price/power

but probably 10s of dollars per megabyte

- e.g., secondary market price of Broadcom chip with 40Mbit of this is $\sim$\$180
- chip needs 80W heatsink, max 900MHz clock rate
- but chip has a bunch of other functionality, of course

P4: using a table

```
control MyIngress(...) {
    ...
    table mac_dst_exact = {
        /* seen earlier */ ...
    }
    apply {
        mac_dst_exact.apply();
    }
}
```

P4: using a table

```
control MyIngress(...) {
    ...
    table mac_dst_exact = {
        /* seen earlier */ ...
    }
    apply {
        mac_dst_exact.apply();
    }
}
```

apply operation invokes the actio

P4: actions

```
control MyIngress(...) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action forward_to(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }
    action mark_and_forward(egressSpec_t port) {
        hdr.some_proto.value = 1;
        standard_metadata.egress_spec = port;
    }
    ...
}
```

P4: actions

```
control MyIngress(...) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action forward_to(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }
    action mark_and_forward(egressSpec_t port) {
        hd
        st
    }
    ...
}
```

actions basically function calls for packet
can include parameters that are stored in table

P4: actions

```
control MyIngress(...) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action forward_to(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }
    action mark_and_forward(egressSpec_t port) {
        hdr
        sta
    }
    ...
}
```

typically, actions set standard metadata
to indicate where to send frame next

actual sending/not sending frame done later

P4: actions

```
control MyIngress(...) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action forward_to(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }
    action mark_and_forward(egressSpec_t port) {
        hdr.so
        standa
    }
    ...
}
```

actions can also edit packet headers
or do other table lookups
(which we will need in the future)

P4: special actions

some ways switch can direct packet (incomplete list)

send to the control plane

- special output port that goes to 'general purpose' CPU

multicast/broadcast to multiple ports

- control plane can set *multicast groups*
- makes multiple copies of packets

exercise

suppose we want to implement the following policy:

- by default, packets sent to servers A, B, C, and D are dropped
- specific machines are given permission to contact server A
- the same is true for servers B and C and D
- some specific machines are give access to contact all servers

what tables would be useful to have?

- what keys?
- what match strategy?

egress processing

same as ingress processing but different function

usage in upcoming assignment:

- ingress step duplicates packet to all output ports
- egress step step runs on each duplicate, drops excess one

P4: control plane

P4 control plane is a program

sends commands to one or more switches:

- load P4 program into data plane
- set table entries
- configure multicast groups
- receive frames to process them

aside: wrappers

I'll show code from a P4 controller for upcoming assignment

written in Python, using custom library to make things convenient

works with softawre based reference switch

no requirement to use Python or other specific language

controller sends commands over network/IPC to data plane

real raw code has more boilerplate/etc.

probably several things different for hardware-based switches

P4: control plane

P4 control plane is a program

sends commands to one or more switches:

- load P4 program into data plane
- set table entries
- configure multicast groups
- receive frames to process them

P4: loading P4 program

```
p4info_helper =  ....
switch = ....
switch.MasterArbitrationUpdate()
switch.SetForwardingPipelineConfig(
    p4info=p4info_helper.p4info,
    bmv2_json_file_path=bmv2_file_path
)
```

P4: loading P4 program

```
p4info_helper =  ....
switch = ....
switch.MasterArbitrationUpdate()
switch.SetForwardingPipelineConfig(
    p4info=p4info_helper.p4info,
    bmv2_json_file_path=bmv2_file_path
)
```

swtich supports having primary + backup controller,
so need to indicate this is primary controller now

P4: loading P4 program

```
p4info_helper =  ....
switch = ....
switch.MasterArbitrationUpdate()
switch.SetForwardingPipelineConfig(
    p4info=p4info_helper.p4info,
    bmv2_json_file_path=bmv2_file_path
)
```

"forwarding pipeline" = dataplane

P4: loading P4 program

```
p4info_helper =  ....
switch = ....
switch.MasterArbitrationUpdate()
switch.SetForwardingPipelineConfig(
    p4info=p4info_helper.p4info,
    bmv2_json_file_path=bmv2_file_path
)
```

P4 code compiled to file to load, specified here

P4: control plane

P4 control plane is a program

sends commands to one or more switches:

- load P4 program into data plane
- set table entries
- configure multicast groups
- receive frames to process them

P4: setting table entries

```
write_or_overwrite_table_entry(
    p4info_helper=p4info_helper, switch=switch,
    table_name='MyIngress.mac_dst_exact',
    match_fields={
        'hdr.ethernet.dstAddr': some_address,
    },
    action_name='forward_to',
    action_params={'port': port},
)
```

P4: setting table entries

```
write_or_overwrite_table_entry(
    p4info_helper=p4info_helper, switch=switch,
    table_name='MyIngress.mac_dst_exact',
    match_fields={
        'hdr.ethernet.dstAddr': some_address,
    },
    action_name='forward_to',
    action_params={'port': port},
)
```

p4info_helper, switch objects created by setup code

P4: setting table entries

```
write_or_overwrite_table_entry(
    p4info_helper=p4info_helper, switch=switch,
    table_name='MyIngress.mac_dst_exact',
    match_fields={
        'hdr.ethernet.dstAddr': some_address,
    },
    action_name='forward_to',
    action_params={'port': port},
)
```

full name of table, including stage it is defined in

P4: setting table entries

```
write_or_overwrite_table_entry(
    p4info_helper=p4info_helper, switch=switch,
    table_name='MyIngress.mac_dst_exact',
    match_fields={
        'hdr.ethernet.dstAddr': some_address,
    },
    action_name='forward_to',
    action_params={'port': port},
)
```

match value — format would be different if key was `lpm` or `ternary` instead of exact match

P4: setting table entries

```
write_or_overwrite_table_entry(
    p4info_helper=p4info_helper, switch=switch,
    table_name='MyIngress.mac_dst_exact',
    match_fields={
        'hdr.ethernet.dstAddr': some_address,
    },
    action_name='forward_to',
    action_params={'port': port},
)
```

`write_or_overwrite_table_entry` not the 'raw' function
(one I wrote based on one P4 tutorial authors wrote)
uses gRPC (remote procedure call) library, which adds some extra steps

P4: control plane

P4 control plane is a program

sends commands to one or more switches:

- load P4 program into data plane
- set table entries
- configure multicast groups
- receive frames to process them

P4: multicast groups

```
switch.WritePREEntry(
    p4info_helper.buildMulticastGroupEntry(
        multicast_group_id=1,
        replicas=[
            {'egress_port': 1, 'instance': 0},
            {'egress_port': 2, 'instance': 0},
            {'egress_port': 3, 'instance': 0},
            {'egress_port': 4, 'instance': 0},
        ]
    )
)
```

P4: multicast groups

```
switch.WritePREEntry(
    p4info_helper.buildMulticastGroupEntry(
        multicast_group_id=1,
        replicas=[
            {'egress_port': 1, 'instance': 0},
            {'egress_port': 2, 'instance': 0},
            {'egress_port': 3, 'instance': 0},
            {'egress_port': 4, 'instance': 0},
        ]
    )
)
```

in P4 dataplane code can write `standard_metadata.mcast_grp = 1` to use this

P4: multicast groups

```
switch.WritePREEntry(
    p4info_helper.buildMulticastGroupEntry(
        multicast_group_id=1,
        replicas=[
            {'egress_port': 1, 'instance': 0},
            {'egress_port': 2, 'instance': 0},
            {'egress_port': 3, 'instance': 0},
            {'egress_port': 4, 'instance': 0},
        ]
    )
)
```

list of ports to output to when group selected

`instance` can be inspected by dataplane code for the egress step

P4: multicast groups

```
switch.WritePREEntry(
    p4info_helper.buildMulticastGroupEntry(
        multicast_group_id=1,
        replicas=[
            {'egress_port': 1, 'instance': 0},
            {'egress_port': 2, 'instance': 0},
            {'egress_port': 3, 'instance': 0},
            {'egress_port': 4, 'instance': 0},
        ]
)
```

PRE = packet replication engine

supports multicast groups (shown) and “clone sessions” (not shown)
(clone sessions make extra copy of packet,
but process original normally)

P4: control plane

P4 control plane is a program

sends commands to one or more switches:

- load P4 program into data plane
- set table entries
- configure multicast groups
- receive frames to process them

P4: receiving frame

```
for item in switch.stream_msg_resp:
    if item.HasField('packet'):
        do_something_with(item.packet.payload,
                          item.packet.metadata)
```

P4: receiving frame

```
for item in switch.stream_msg_resp:
    if item.HasField('packet'):
        do_something_with(item.packet.payload,
                          item.packet.metadata)
```

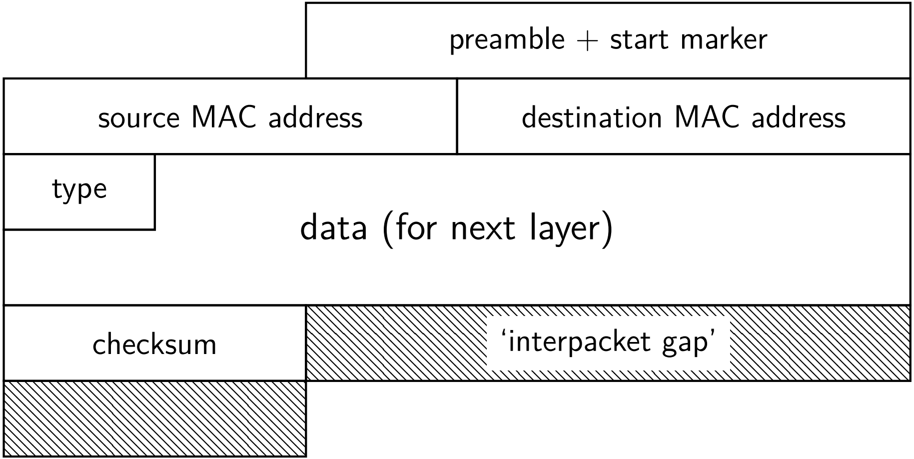
payload = bytes of frame
(what would normally be sent on network)

P4: receiving frame

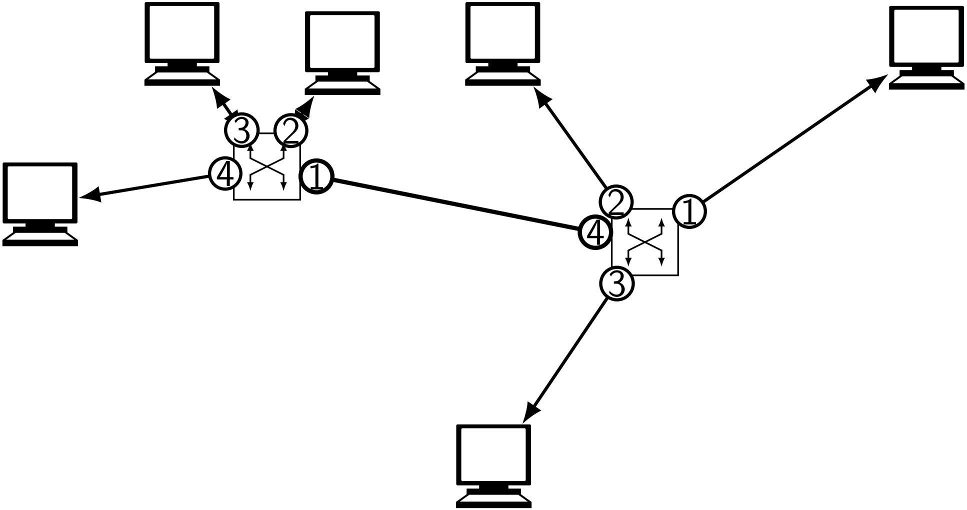
```
for item in switch.stream_msg_resp:
    if item.HasField('packet'):
        do_something_with(item.packet.payload,
                          item.packet.metadata)
```

extra metadata can be set by dataplane
example: which port packet came from

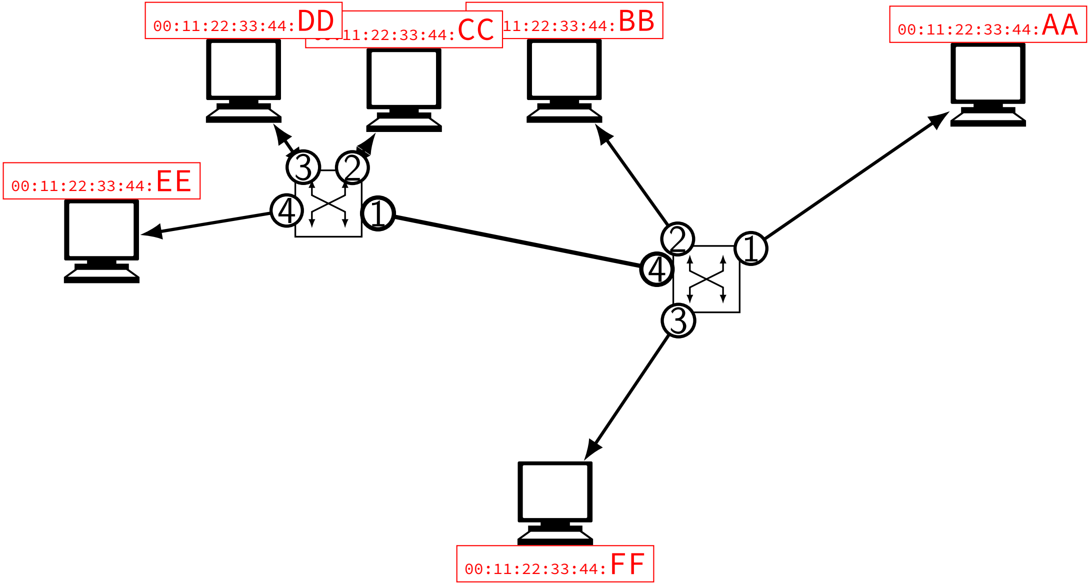
typically sent on Ethernet



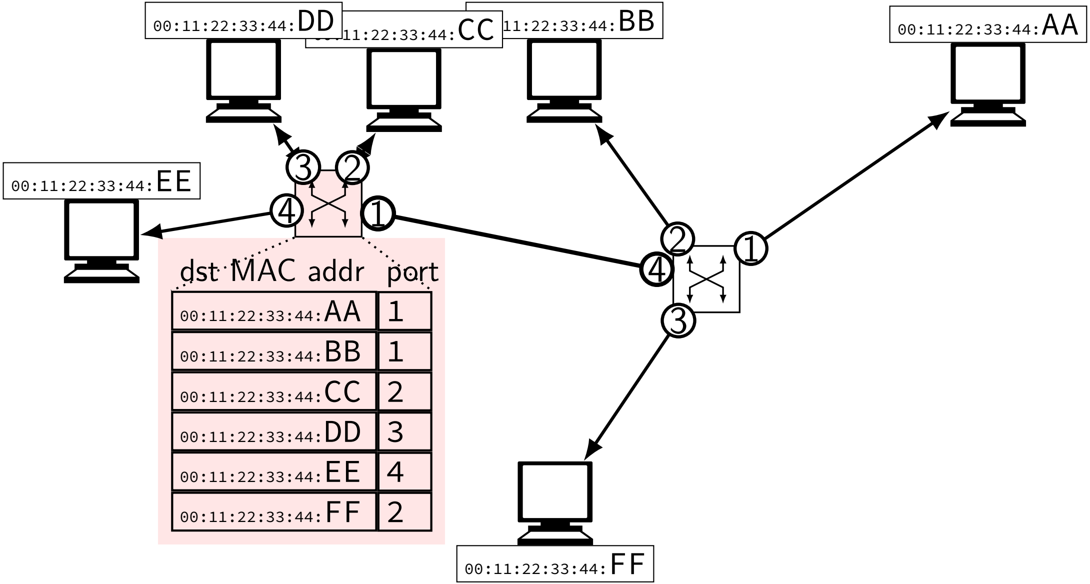
network and switch tables



network and switch tables

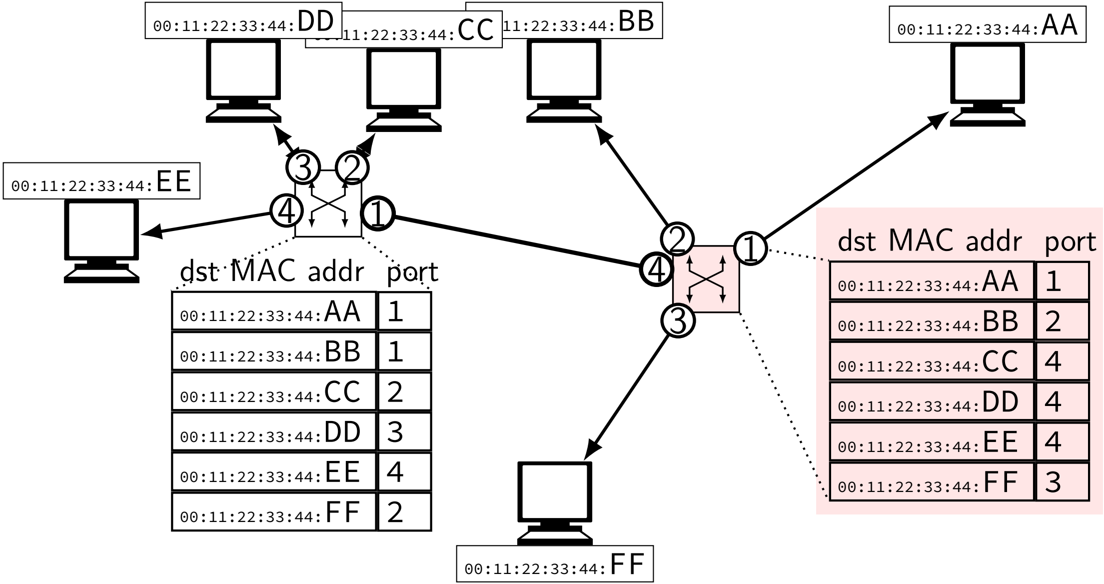


network and switch tables



dst MAC addr	port
00:11:22:33:44:AA	1
00:11:22:33:44:BB	1
00:11:22:33:44:CC	2
00:11:22:33:44:DD	3
00:11:22:33:44:EE	4
00:11:22:33:44:FF	2

network and switch tables



dst MAC addr	port
00:11:22:33:44:AA	1
00:11:22:33:44:BB	1
00:11:22:33:44:CC	2
00:11:22:33:44:DD	3
00:11:22:33:44:EE	4
00:11:22:33:44:FF	2

dst MAC addr	port
00:11:22:33:44:AA	1
00:11:22:33:44:BB	2
00:11:22:33:44:CC	4
00:11:22:33:44:DD	4
00:11:22:33:44:EE	4
00:11:22:33:44:FF	3

constructing switch tables

could have system administrator input these by hand

- through an SSH-like interface, probably

works, but error-prone, hard to change, etc.

constructing switch tables

could have system administrator input these by hand

- through an SSH-like interface, probably

works, but error-prone, hard to change, etc.

alternative: switch should figure it out

MAC learning

forwarding table

dst MAC addr	port

MAC learning

forwarding table

dst MAC addr	port
(default)	ALL*

MAC learning

forwarding table

dst MAC addr	port
(default)	ALL*

incoming frame 1

input port=2, output port = ???	
data =	src=00:11:22:33:44:AA dst=00:11:22:33:44:FF type = IPV4 data = 33 45 43 42 ...

MAC learning

forwarding table

dst MAC addr	port
(default)	ALL*

incoming frame 1

input port=2, output port = all but 2

data =
src=00:11:22:33:44:AA
dst=00:11:22:33:44:FF
type = IPV4
data = 33 45 43 42 …

MAC learning

forwarding table

dst MAC addr	port
00:11:22:33:44:AA	2
(default)	ALL*

incoming frame 1

input port=2, output port = all but 2

data =

src=00:11:22:33:44:AA
dst=00:11:22:33:44:FF
type = IPV4
data = 33 45 43 42 …

MAC learning

forwarding table

dst MAC addr	port
00:11:22:33:44:AA	2
(default)	ALL*

incoming frame 1

input port=2, output port = all but 2
data = src=00:11:22:33:44:AA dst=00:11:22:33:44:FF type = IPV4 data = 33 45 43 42 ...

incoming frame 2

input port=3, output port = ???
data = src=00:11:22:33:44:FF dst=00:11:22:33:44:AA type = IPV4 data = 34 45 43 42 ...

MAC learning

forwarding table

dst MAC addr	port
00:11:22:33:44:AA	2
(default)	ALL*

incoming frame 1

input port=2, output port = all but 2

data = src=00:11:22:33:44:AA
dst=00:11:22:33:44:FF
type = IPV4
data = 33 45 43 42 …

incoming frame 2

input port=3, output port = 2

data = src=00:11:22:33:44:FF
dst=00:11:22:33:44:AA
type = IPV4
data = 34 45 43 42 …

MAC learning

forwarding table

dst MAC addr	port
00:11:22:33:44:AA	2
00:11:22:33:44:FF	3
(default)	ALL*

incoming frame 1

input port=2, output port = all but 2
data = src=00:11:22:33:44:AA dst=00:11:22:33:44:FF type = IPV4 data = 33 45 43 42 ...

incoming frame 2

input port=3, output port = 2
data = src=00:11:22:33:44:FF dst=00:11:22:33:44:AA type = IPV4 data = 34 45 43 42 ...

aside: no backwards broadcast

recall: broadcast sent to all but incoming port

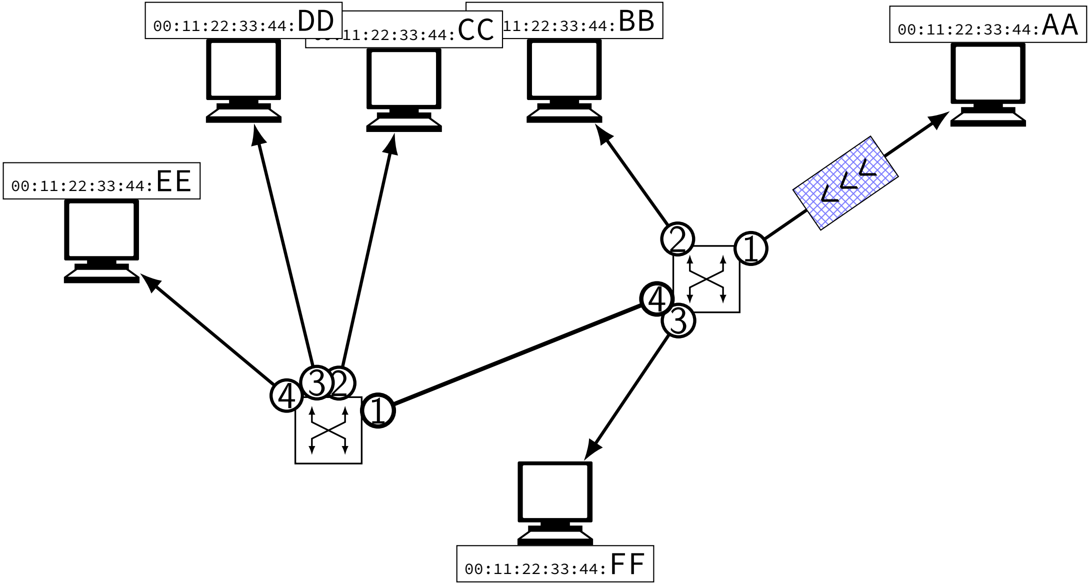
question: what would happen if we didn't do this?

multiple may apply

A. might cause host to receive duplicates
B. might cause copies sent to non-incoming port to be dropped
C. might cause frames sent at same time to be dropped
D. might cause frames sent much later to be dropped

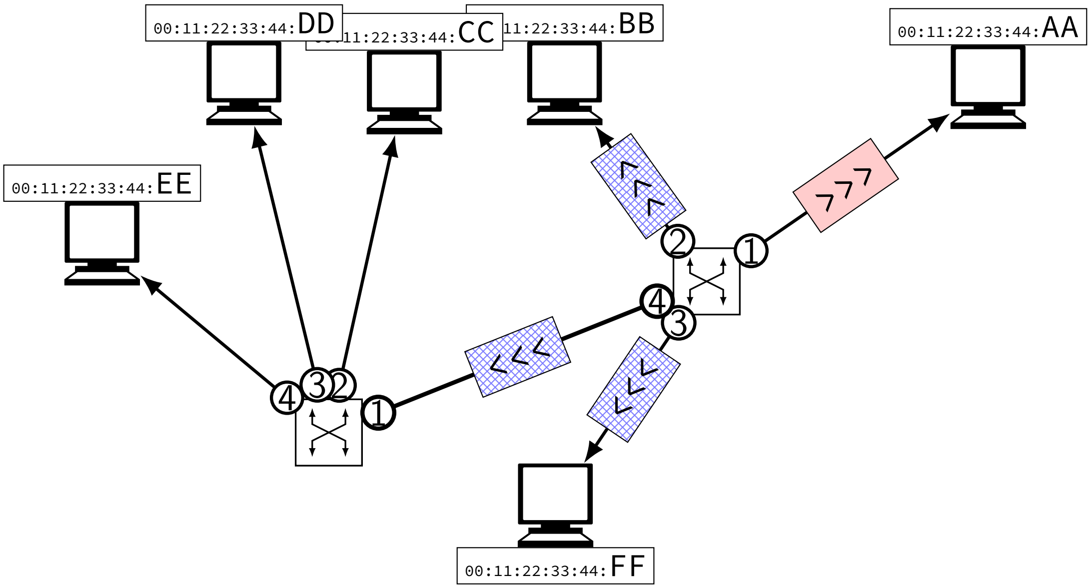
loop from backwards broadcast

time step 1



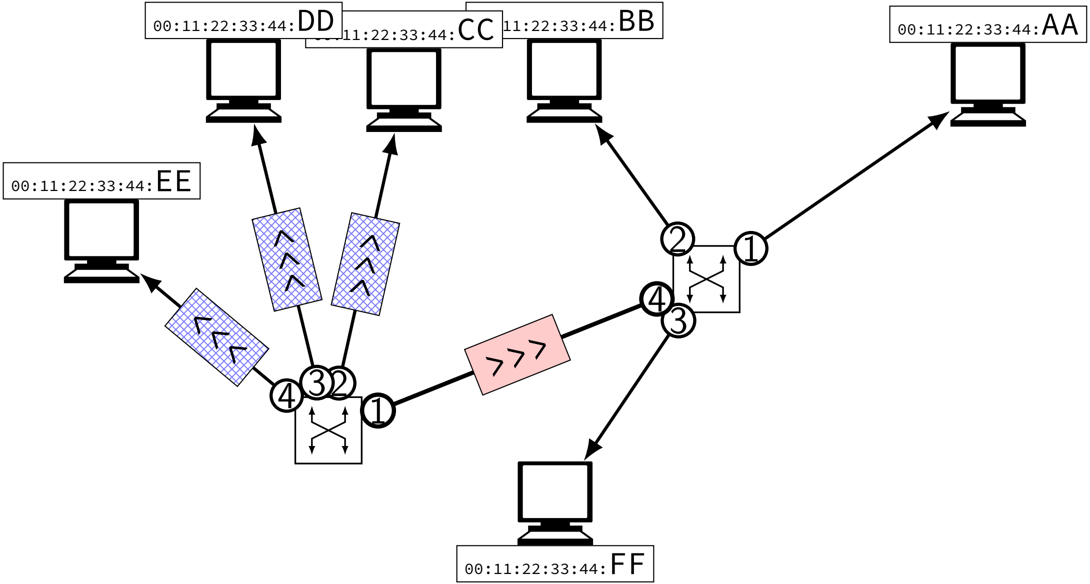
loop from backwards broadcast

time step 2



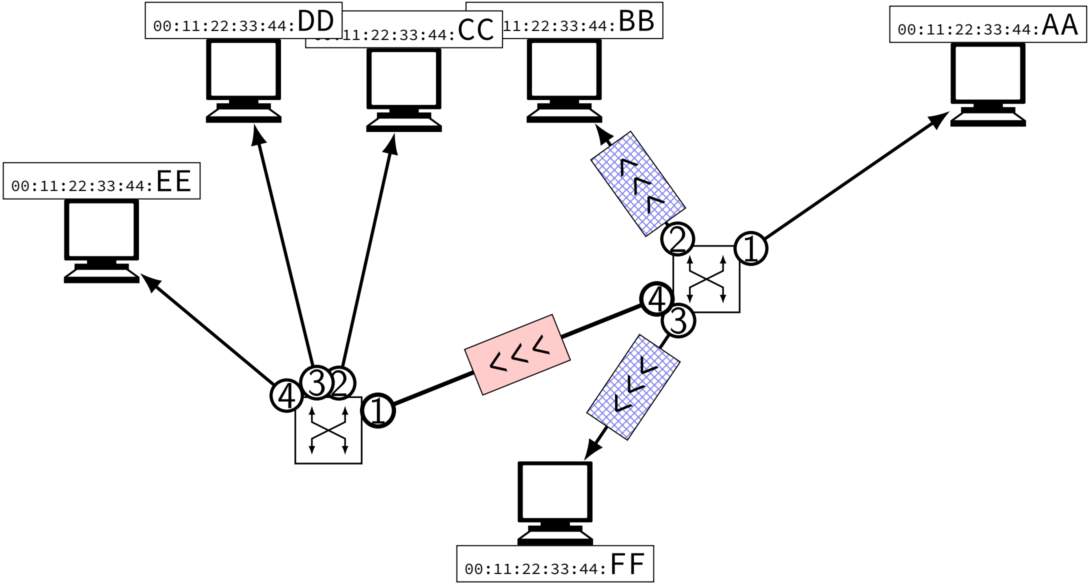
loop from backwards broadcast

time step 3



loop from backwards broadcast

time step 4



loops

each packet keeps getting sent indefinitely

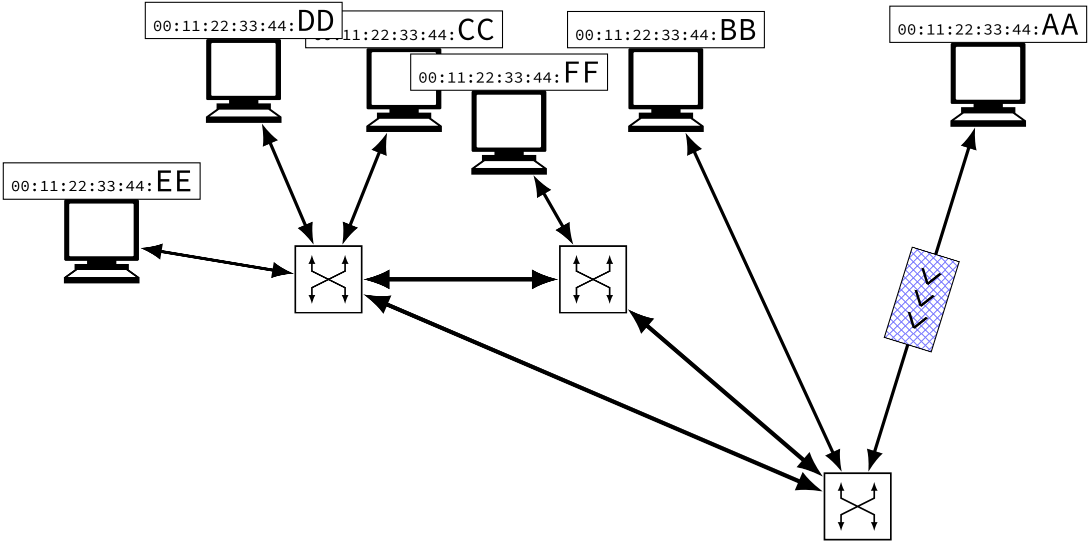
remember: happens for *every packet sent*

quickly overwhelms link between switches

but can just avoid by not sending back?

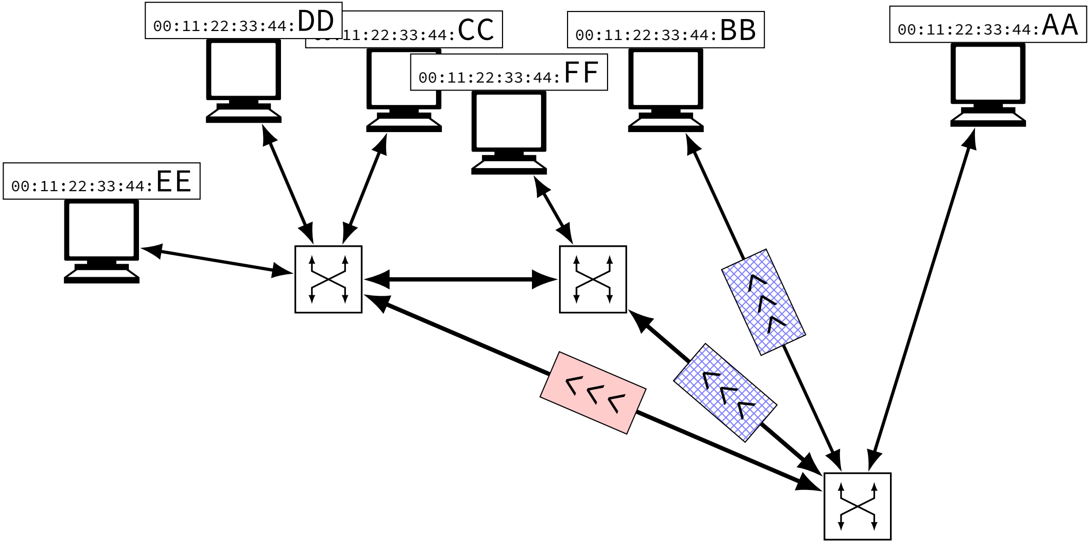
loops with only-to-other

time step 1



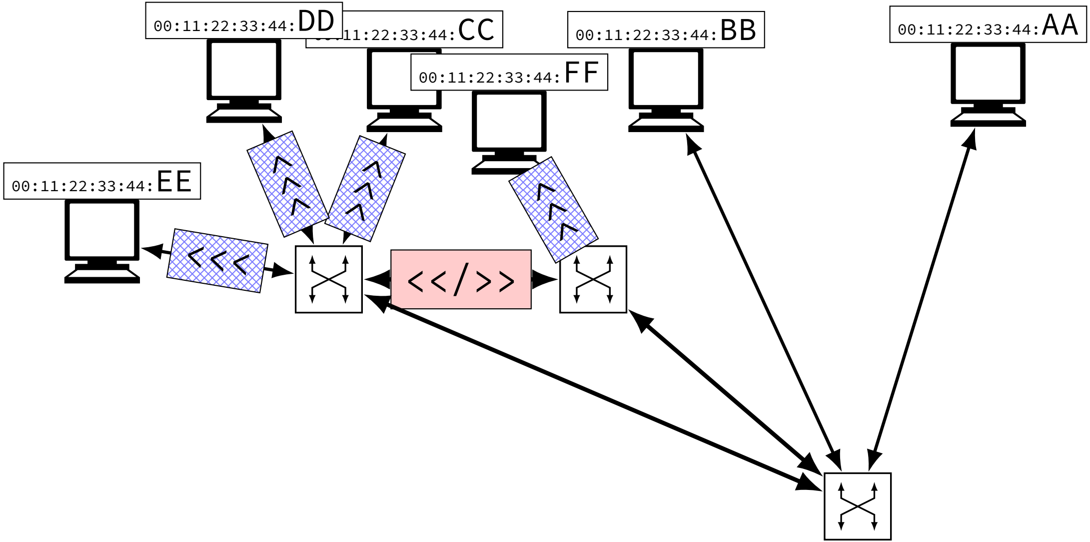
loops with only-to-other

time step 2



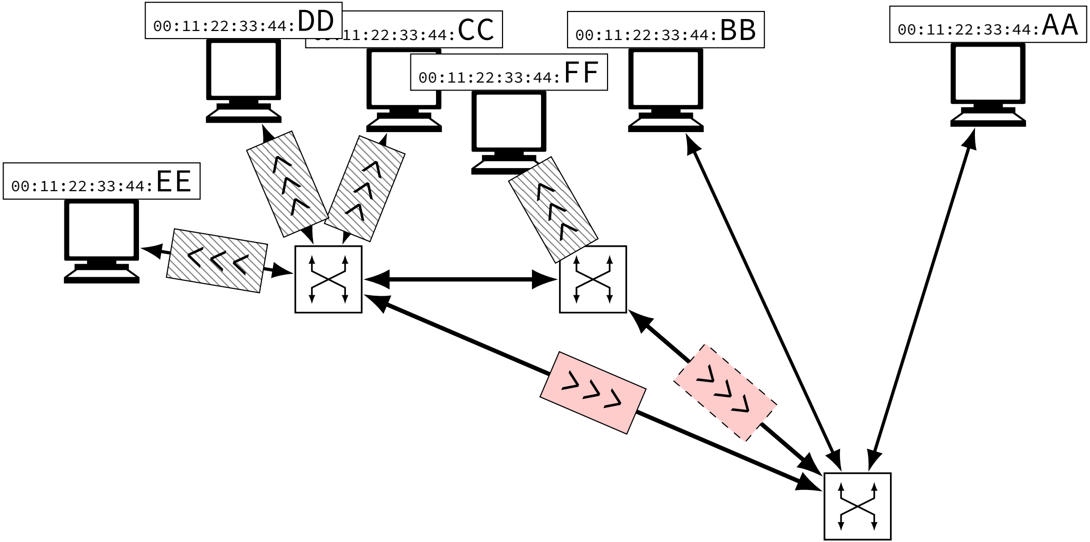
loops with only-to-other

time step 3



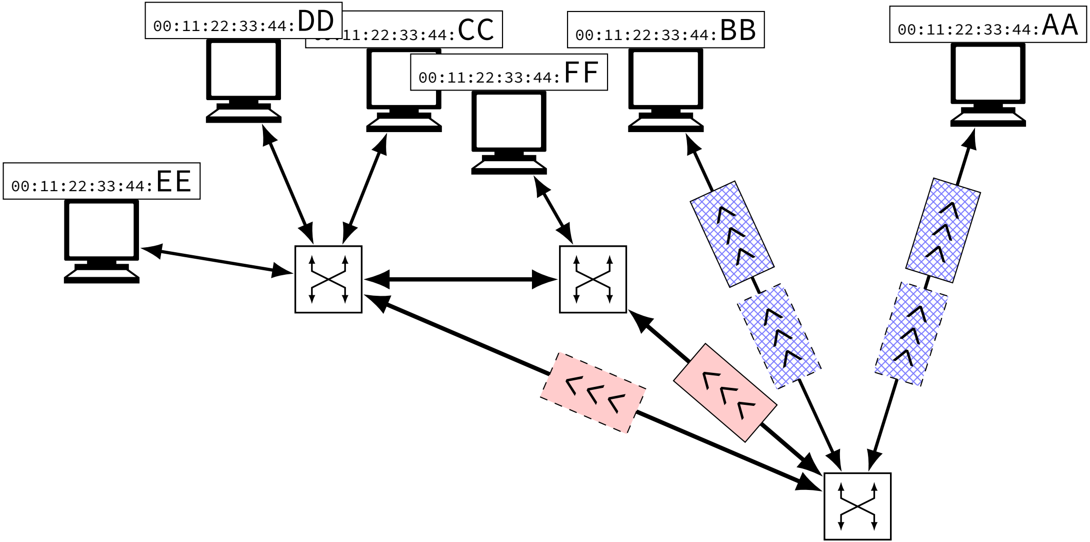
loops with only-to-other

time step 4



loops with only-to-other

time step 5



explosion from loops

avoiding sending back is not enough

will get catastrophic failure of network!

simple fix: only have one path from A to B

BUT means network is more fragile

we'll have better solutions when we talk about routing later

preview: routing

better ways to decide where to send packets

...but require coordinating between switches

- avoid loops
- choose between multiple paths
- avoid 'flooding' for each new machine

problem also very important for large networks...

...like the Internet

we will revisit it when we talk about IP routing

backup slides