Explanation, Pseudocode, Data Structures

High Level Explanation:
The algorithm is centered around how the controller will determine where the given switch should forward the packet to. The controller uses a data structure to maintain a state and to incrementally learn about the topology. My approach was to ensure the controller could have a table for each switch. When a switch sent a packet to the controller, it would first update/add the source information from the packet (via a source to port mapping. Then it would attempt to deal with the destination information the packet specified (finding an entry that has the destination address its corresponding port can be sent back to the switch). However, it was essential this search for the dest-port mapping entry was only done in the switches own table and not in **all** the switch tables the controller has, which is why each element in the controllers data structure is representing a different switches table. This left only two possible cases to handle: either the switch table had the matching dest-port mapping entry, or it didn't have the entry. If a match was found then we simply send the corresponding port value back to the switch and if no match was found then we tell the switch to flood all of its other ports.

Thought Process Through the Algorithm:
Steps one and two are about making sure the controller has a table for the switch that sent the packet and adding the source-related mapping to the table. This needs to be done before any destination-related logic is dealt with.

STEP 1
Check if the controller already has a switch-table for the switch which sent the packet: if not, then create empty table for this switch

STEP 2
Use the packets source address and 'packetInPort' to add the address-port mapping of the source as an entry in the switches table (unless the table already has an entry for this source address. (the source address will be the 'value' of the 'mac' key, so we would be iterating through the elements in the switch_table, and comparing the 'value' of the dictionaries first elem, to packet.src (since the first elem in the dictionary is the 'mac':'address' key-value pair, and we want to check 'address', which is the associated value)

STEP 3
Now the controller will take the packets destination address and look for a matching entry in the switch_table (using the same iteration that was done to check for a source address entry in step 2).
CASE 1: MATCH FOUND --> get 'port' from match and put it in the message back to the switch
CASE 2: NOT FOUND: --> flood to all other of the switches ports.

More detailed explanations of the two cases:
CASE 1: If a matching entry was found, then we can get the corresponding 'port' which is the important piece of information the switch was asking for when it sent the packet to the controller in the first place. We can then send this port value in the message being sent back to the switch and it will then know where to forward the packet to.
CASE 2: If a match was not found in the switches table, then we send a message back to the switch which tells it to flood all of its 'other' ports. Because of the initial steps of the algorithm, when this flood eventually returns the needed host-port information, it will be entered into the controllers topology right away instead of needing to wait for the destination-related address to be dealt with. This is what allows for the topology to be iteratively mapped out, even when starting out with no knowledge of the host/switch relationships.

PSEUDO-CODE (Cormen):

```
if switchID not in all_switch_tables:
     add empty table for this switchID to all_switch_tables

switch_table = all_switch_tables[switchID] // table of the switch that sent the packet
if packet.src not in switch_table: // add the source-port mapping to the switch table
     switch_table.append(packet.src, inputPort)

// search the table for entry matching packet.dst…
if packet.dst in switch_table: // entry found
     foundPort = switch_table[packet.dst]
     msg = (foundPort)
     sendBackToSwitch(msg)
else:  // entry not found…
     msg = (flood)
     sendBackToSwitch(msg)
```

Data structure explanation: (also in ethernet-learning.py)

```
36  # ********************************************************************
37  # DATA STRUCTURE OVERVIEW:
38  # ********************************************************************
39
40  # CONTROLLER:
41  # the controller learns/maps the network with a DICT called all_switch_tables. The key-value pairs of this dict are the switchID and the corresponding switch table
42  # (key == switchID, val == switch_table)
43  # This allows the controller to quickly check if it already has a switch_table for the switch that just sent it the packet in question, while also ensuring it doesn't have
44  # multiple tables for the same switch.
45
46  # SWITCH-TABLE:
47  # a switch table is a LIST, called switch_table, where each elem is itself a DICT (and this DICT will have TWO elements, each of which is a key-val pair)
48  # The 2-elem DICT's will be as follows: (one element is for the mac address and the other is for the port)
49  #  left elem: key-mac:addr-val
50  #  right elem: key-port:port-val
51
52  # The primary task the controller needs to be able to do is search through a particular switches table to match the packets address (either source or dest) with the address in the table.
53  # Because each element of the table is a dictionary, the address the controller needs in the table is the 'value' of the first element.
54  # So for each entry in the table, it will evaluate that entries first element by referencing the "mac" key to get the associated 'value', which is the actual address for the entry, and this
55  # is the value the controller will be trying to match the packets addresses with.
56
```

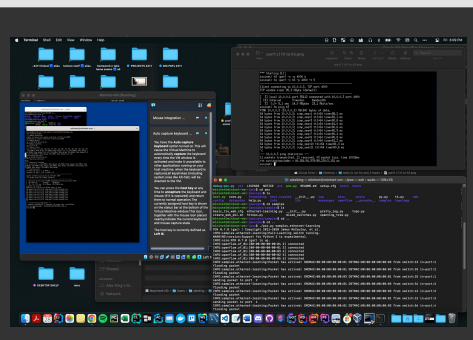
Part 3 Results for each topology:

topology a

L1

BW (Mbits/sec) : 19.3

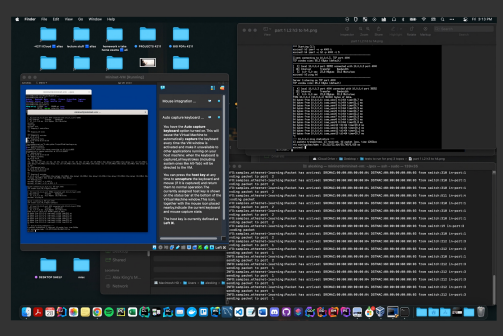
Latency(ms) : 80.6



L2

BW (Mbits/sec) : 37.0

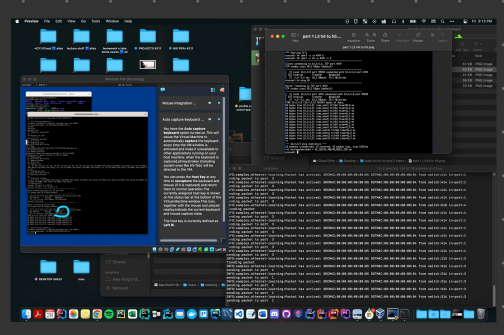
Latency(ms) : 21.6



L3

BW (Mbits/sec) : 25.0

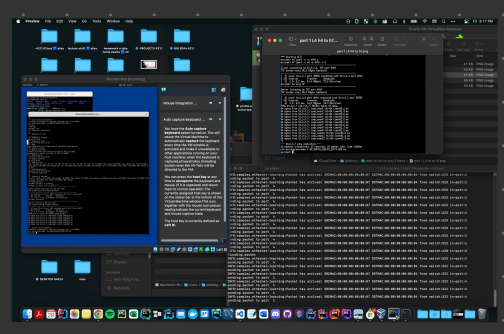
Latency(ms) : 41.3



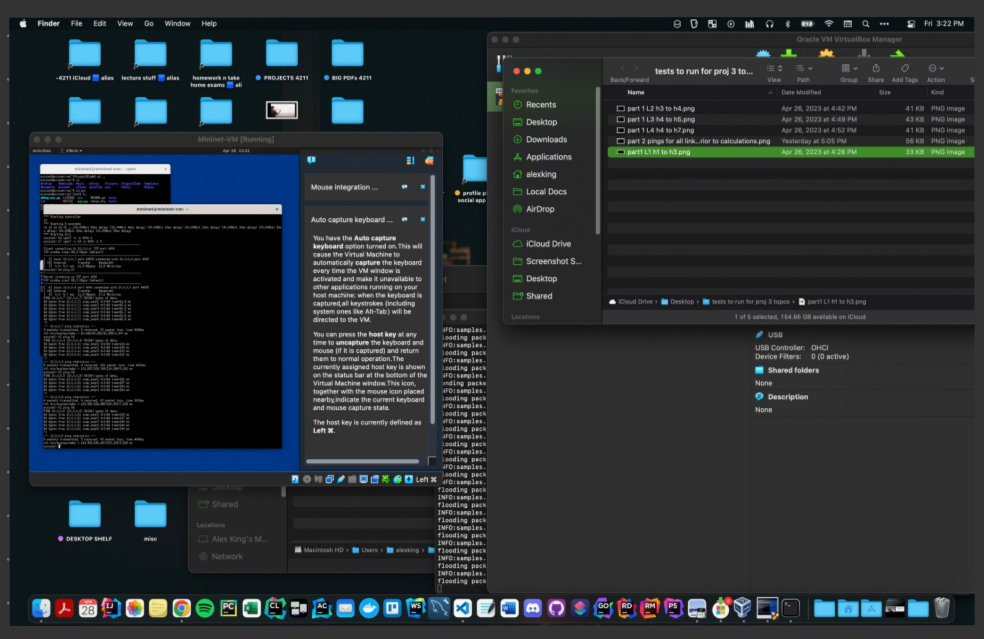
L4

BW (Mbits/sec) : 17.6

Latency(ms) : 60.9



Extra pings to demonstrate each host can reach any of the other hosts



topology b

LATENCY

level 1 link(s): 61.0

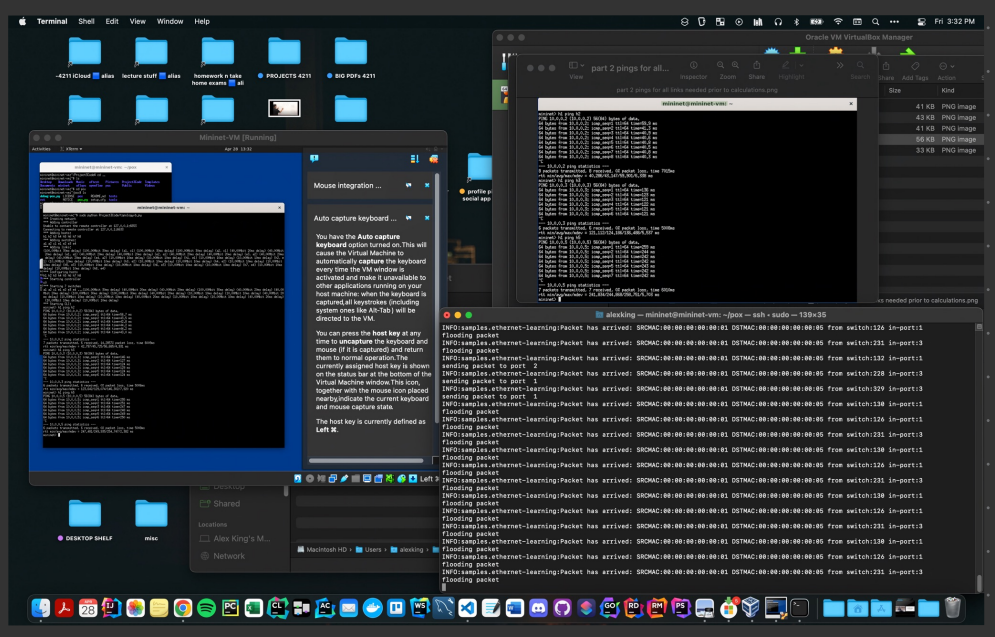
level 2 link(s): 41.6

calculation...

total = (250)

calculation

total = (129.1)



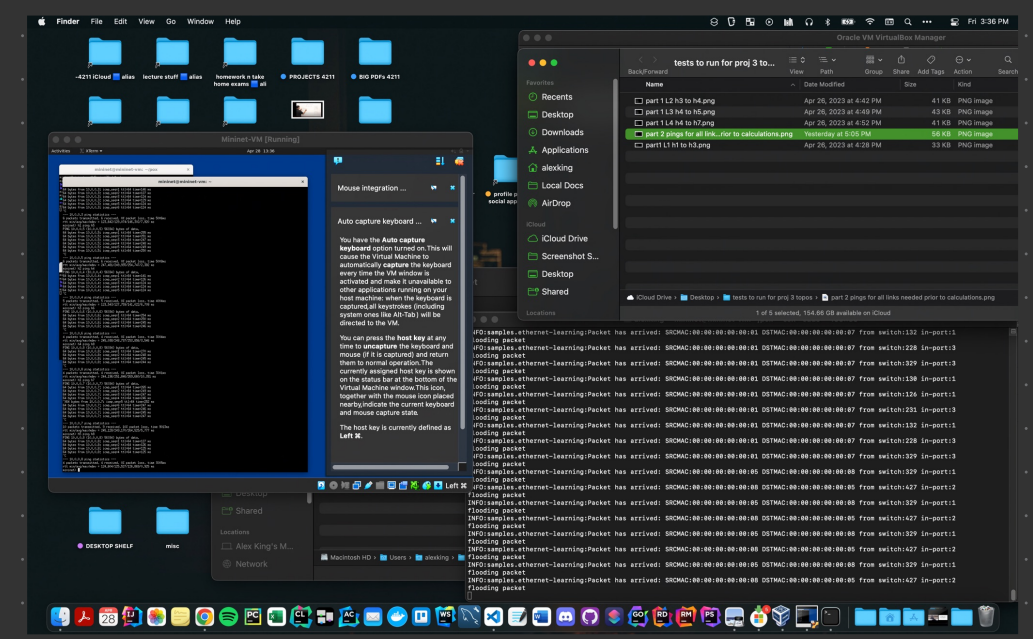
$$\text{cost of layer2 link} = \frac{(\text{total} - 2(\text{layer3 link}))}{2} = \text{cost of L1}$$

$$= \frac{(129.1 - 2(22.8))}{2} = (41.6)$$

$$\text{Cost of layer 1 link} = \frac{(\text{total} - (2z + 2y))}{2}$$

$$= \frac{(250 - (2(41.6) + 2(22.8)))}{2} = \frac{(250 - 128)}{2} = (61)$$

Extra pings to demonstrate each host can reach any of the other hosts



Comparisons

Comparing part 1 results to part 3 results (topology-a)

PART 1 | PART 3

L1 Bandwidth: 23.1 Mbits/sec
L1 Latency: 81.6 ms

L2 Bandwidth: 39.5 Mbits/sec
L2 Latency: 22.4 ms

L3 Bandwidth: 30.2 Mbits/sec
L3 Latency: 41.5 ms

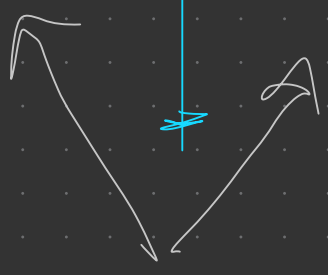
L4 Bandwidth: 22.6 Mbits/sec
L4 Latency: 61.5 ms

L1
BW (Mbits/sec) : 19.3
Latency (ms) : 80.6

L2
BW (Mbits/sec) : 37.0
Latency (ms) : 21.6

L3
BW (Mbits/sec) : 25.0
Latency (ms) : 41.3

L4
BW (Mbits/sec) : 17.6
Latency (ms) : 60.9



The similar link values show our part-3 algorithm worked for topology-a

Comparing part 2 results to part 3 results (topology-b)

PART 2

LATENCY

total for level 1 test : 244.9
level 1 link(s) : 60.6

total for level 2 test : 124.2
level 2 link(s) : 40.4

total for level 3 test : 43.6
level 3 link(s) : 21.6

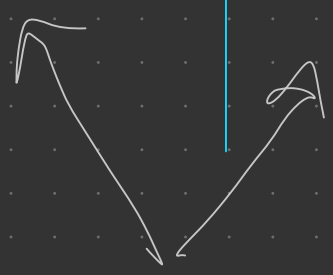
PART 3

LATENCY

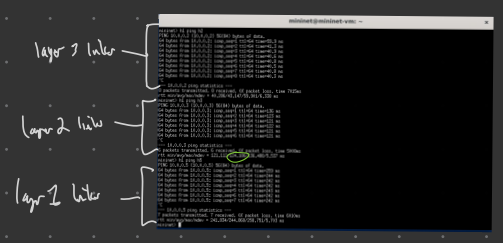
total for level 1 test : 250
level 1 link(s) : 61.0

total for level 2 test : 129.1
level 2 link(s) : 41.6

total for level 3 test : 45.7
level 3 link(s) : 22.8

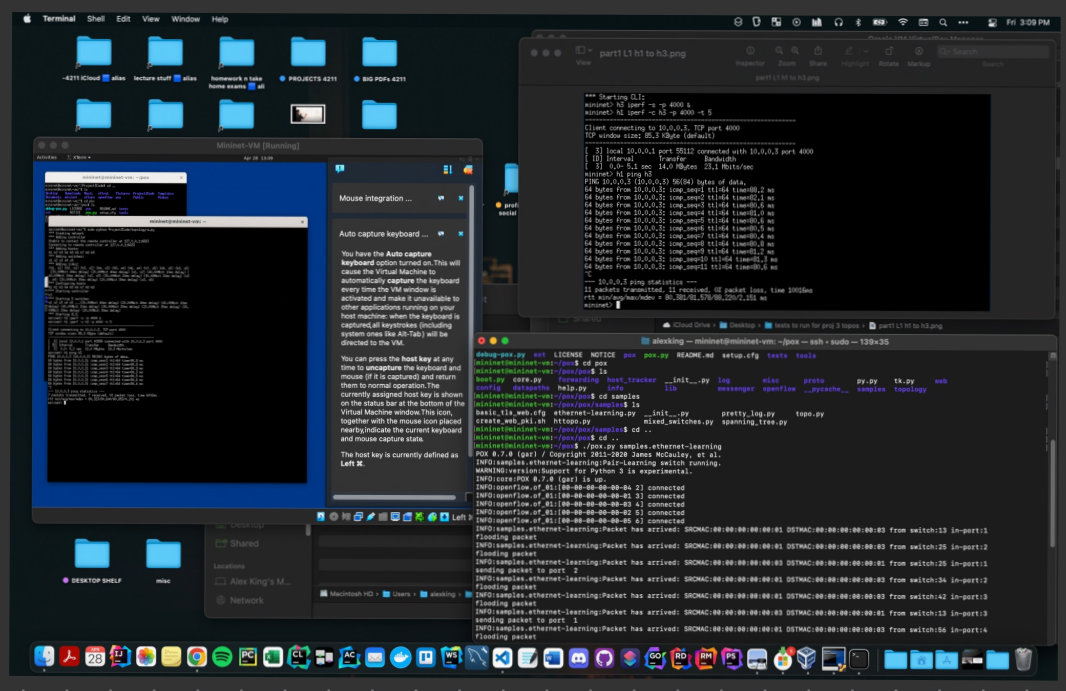


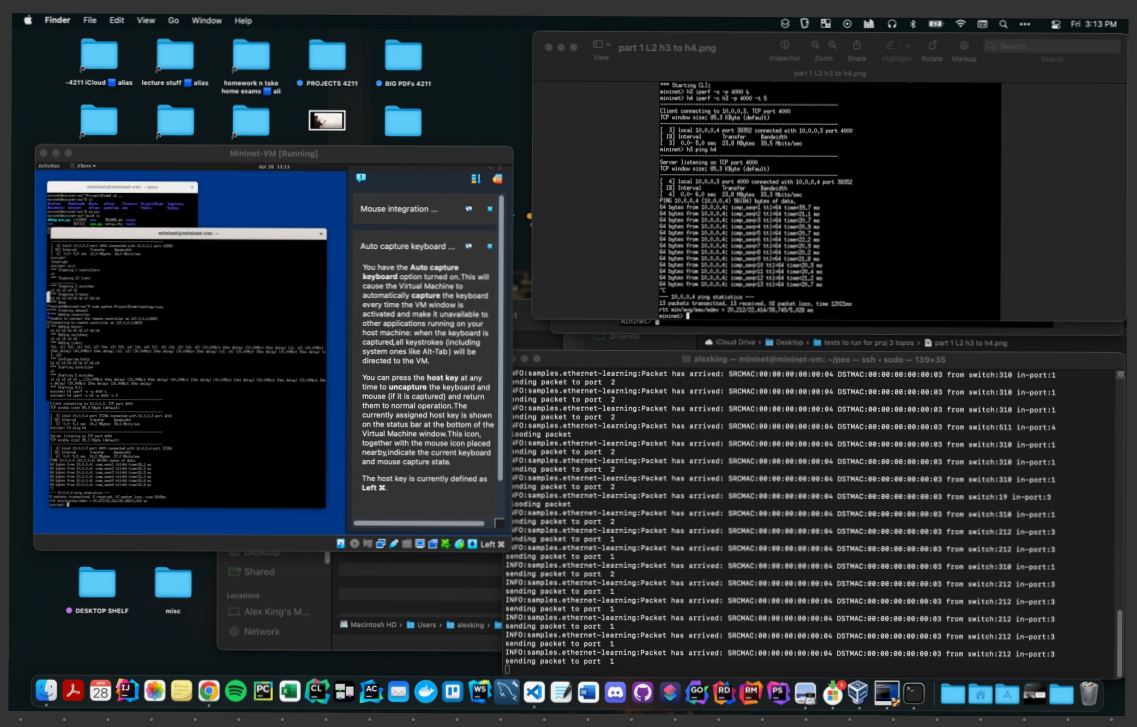
The similar link values show our part-3 algorithm worked for topology-b

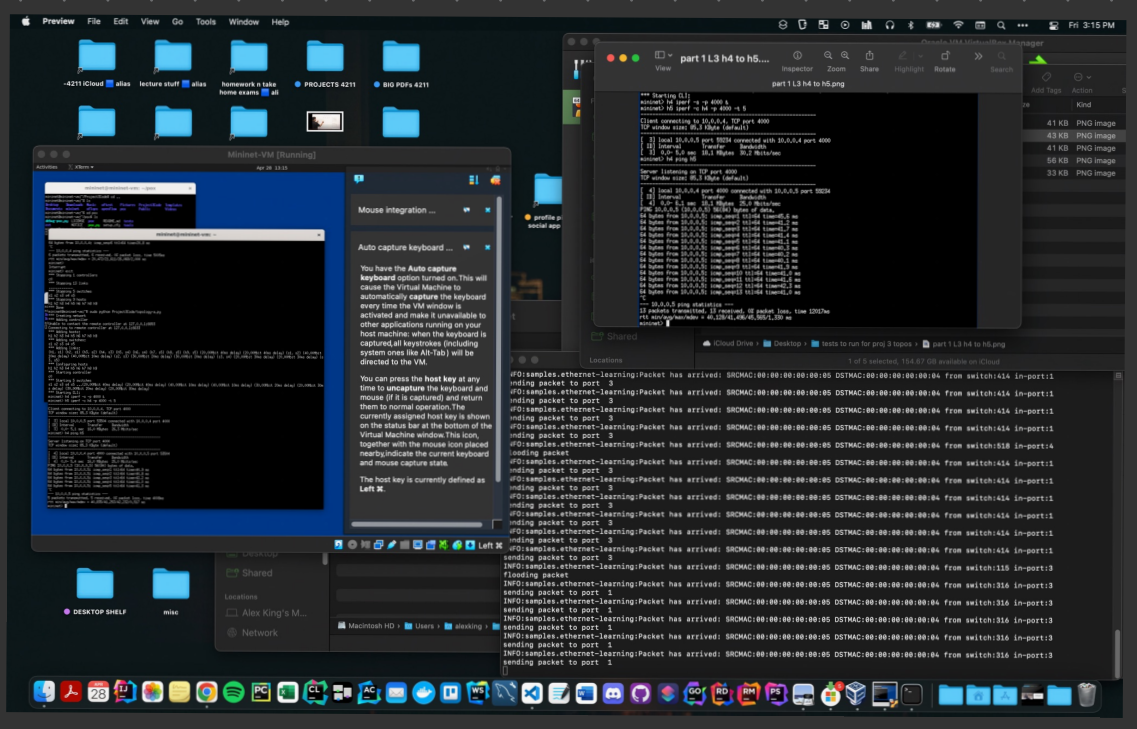


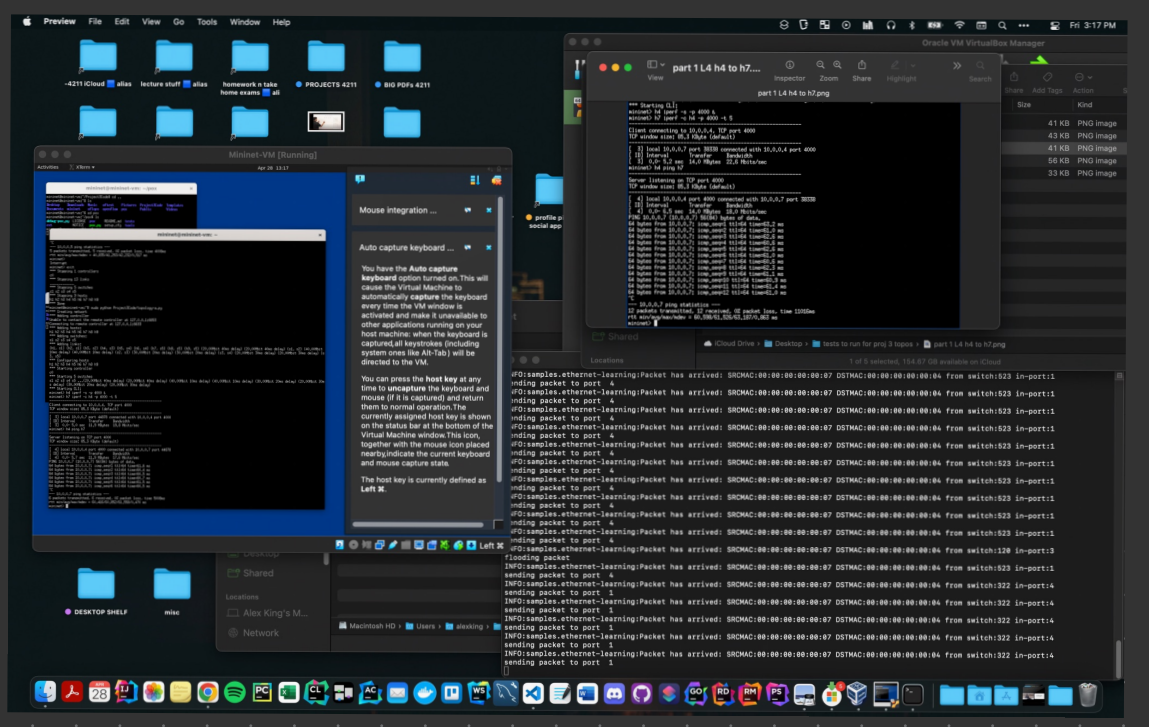
ENLARGED SCREENSHOTS

topology a





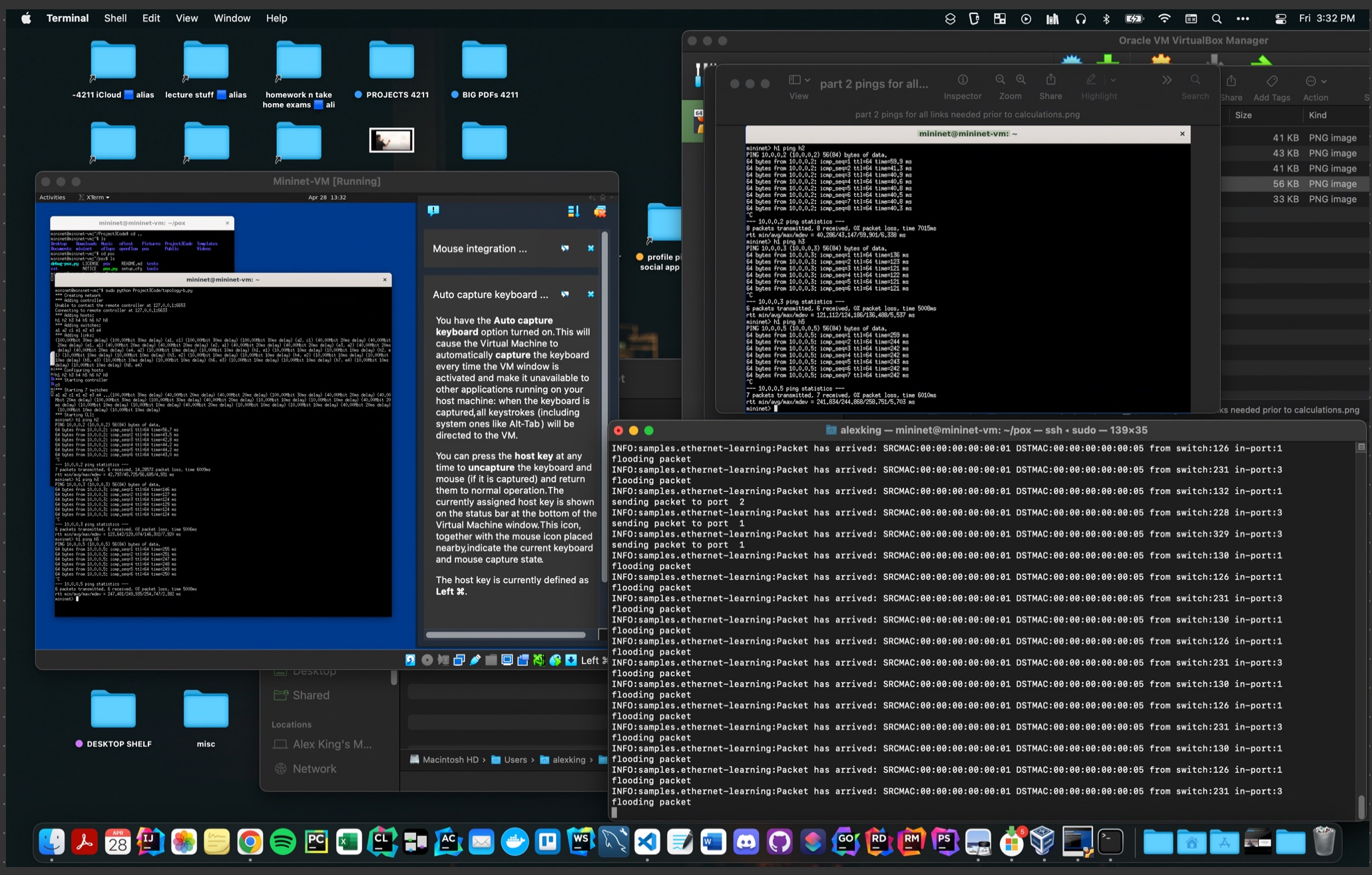


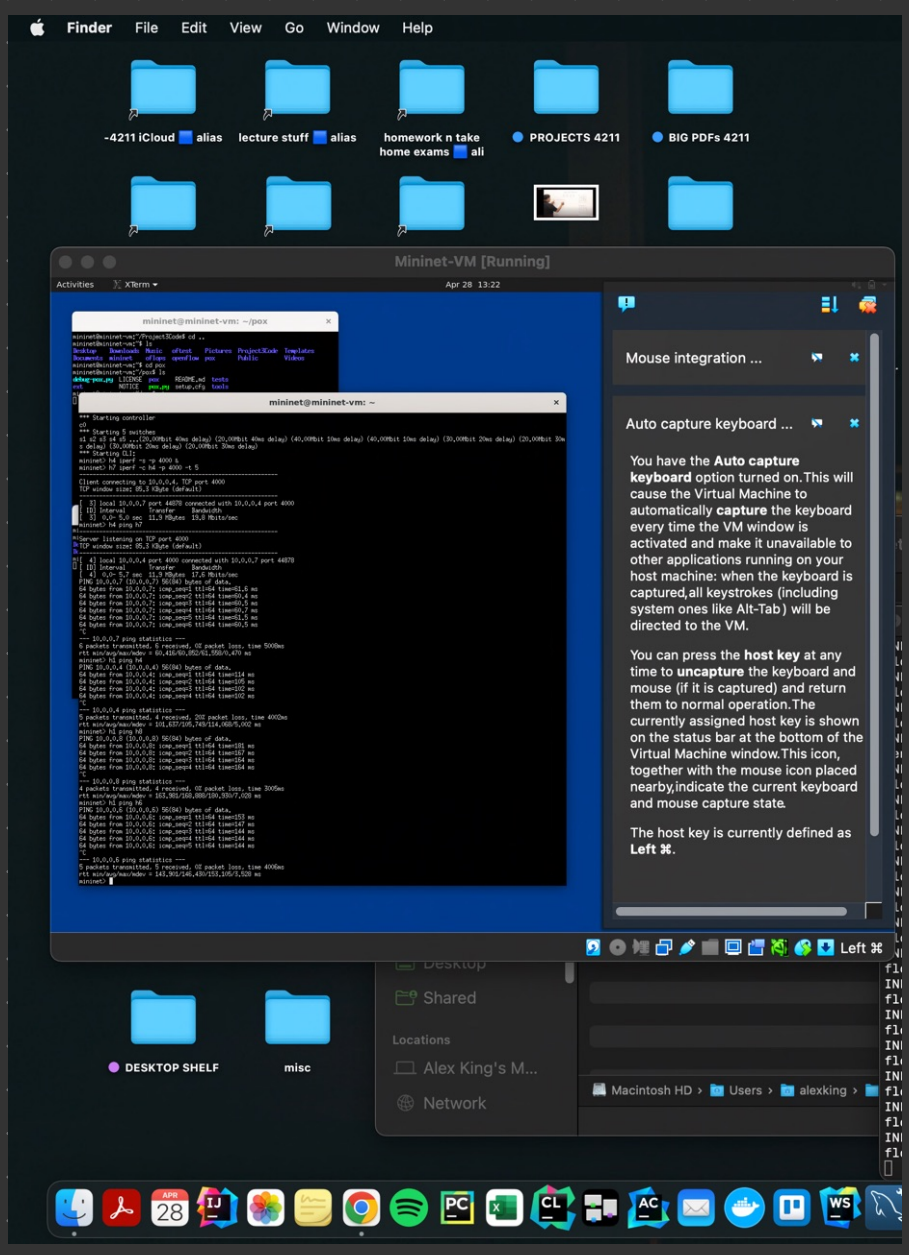


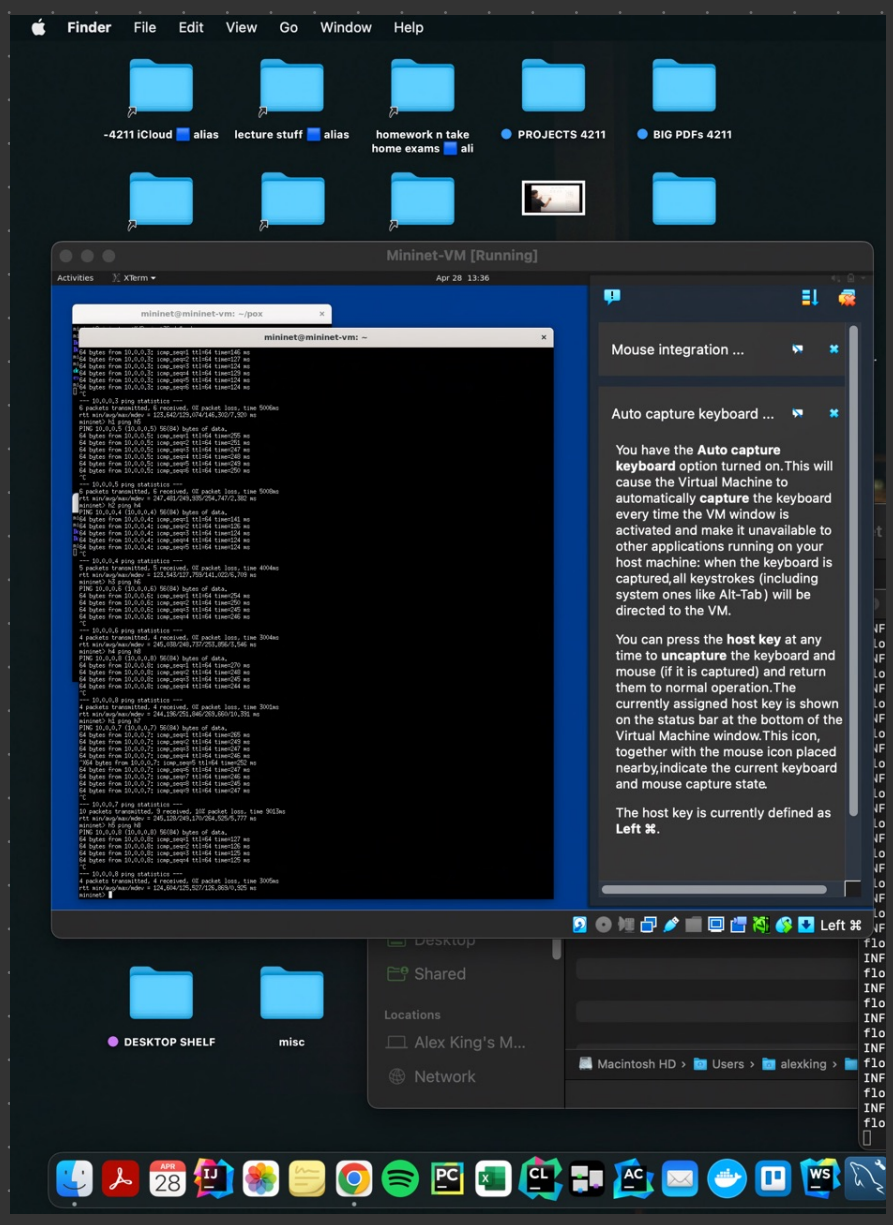
ENLARGED SCREENSHOTS

topology b

+ Extra PINGS





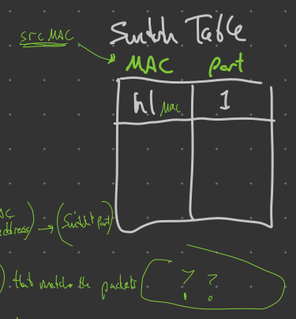


SCRATCHWORK

accepts 

(b/c switch sending it has no entry for dest)

OFPacketIn



Switch Table (Dictionary)

mac port

① First add mapping of (SRC MAC address) → (Switch Port)

② Look in Table for (DEST MAC) that matches the packet

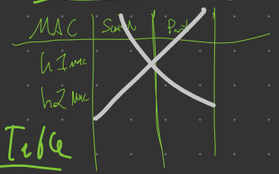
Entry found

NOT FOUND

Tell switch to Flood all its ports except

List, where each elem is a Dictionary

Controller Table



sw#1 table . append ({ Key : Value , Key : Value })

Controller Table

1 elem in the switch Table List

List

sw#1 table . append ({ Key (MAC) : Value , Key (Port) : Value })

Switch Table = { : , }

Dict with 2 elements

each elem has a key/val pair