Homework 3

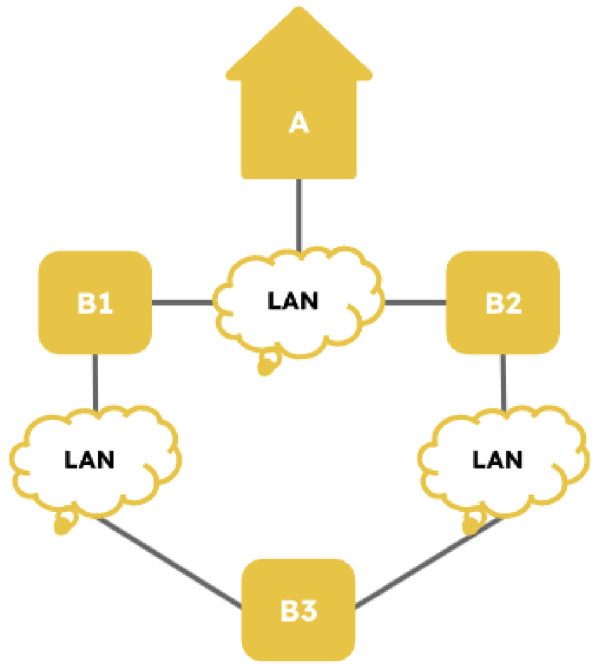
Tejas Kamtam

305749402

Question 1 - Bridging and Loops

> The Spanning Tree algorithm cannot prevent temporary loops (for example, if two separate LANs are connected by a bridge, and then someone plugs them together using a repeater). Eventually, the loop will be broken, and the right bridges will turn off. However, packets can circulate at very high speeds until the loop is broken.

Alyssa P. Hacker has thought of a way to improve the situation during temporary loops. Consider a temporary loop of bridges shown above and assume that bridges **B1**, **B2**, and **B3** all think they are **ON**. Suppose **A** sends a multicast packet. Both **B1** and **B2** pick up the packet, and the packet will circulate in two directions, both clockwise and anti-clockwise. This will also happen if the destination is unknown.



Q1.1 - Modifying Bridge Learning

> Based on Alyssa's observation, how could you modify the bridge learning and forwarding to prevent multicast and unknown destination packets from circulating continuously in a temporary loop?

There are a couple solutions I can think of, but only 1 that takes advantage of Alyssa P. Hacker's observation: Given there is a temporary loop, we could modify the bridge packet forwarding code to check for duplicate packets. In this example, we know B1 and B2 will cycle the same multicast packet in opposite directions. This means that at some point in the cycle B3 will have received both packets. Because switches/bridges have buffers, we can use the buffers to check whether two identical packets (identified by possible their sequence number) exist on the same bridge at different buffers (e.g. left and right buffers). This means that the same packet was received from both endpoints implying a loop so we can drop one of the packets from one of the buffers. Applying this code to all bridges B1, B2, and B3, each bridge would, by the end of 1 cycle, detect that it had received and transmitted the same packet on 2 opposing buffers and be notified of a temporary loop and decide to drop 1. Which, if done at every buffer would drop all instances of the packet by the end of 1 loop meaning it would have also been sent to every LAN that it needed to go to (because it was multicast or flooded if the dest. is unknown).

Q1.2 - Dropped Packets

> What goes wrong with your method if packets can be dropped?

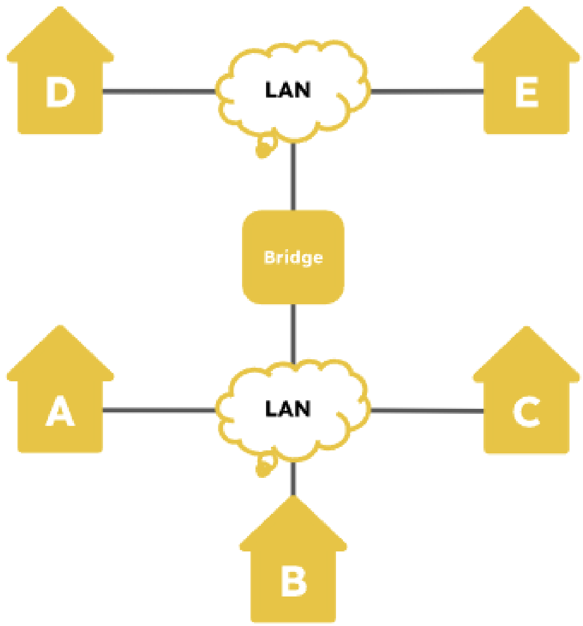
If packets can be dropped, my solution above fails because a bridge may fail to detect a packet on 2 opposing buffers. Suppose for B3 the buffer on the left receives the multicast packet but not on the right. Then B3 would never have known there was actually a loop and would continue to forward cyclically, counter-clockwise. This could be mitigated by implementing something like a hop-count or packet time-to-live.

Question 2 - IP Broadcast Storms

> (Adapted from Perlman's book) A broadcast storm is an event that causes a flurry of messages. One implementation that caused broadcast storms was the Berkeley UNIX endnode IP implementation. In this implementation, an endnode attempts to forward a packet that it mysteriously receives with a network layer (IP) address that is different from itself. This is what you would do if you found a neighbor's letter wrongly placed in your mailbox. However, this seemingly helpful policy can cause problems.

Consider the figure below which shows 2 LANs connected by a bridge, with several IP endnodes on each LAN. There are no IP routers. All IP endnodes are configured with the same mask and so can tell that they have the same net number/prefix.

Suppose IP endnode A is incorrectly configured and incorrectly thinks its data link address is all 1's. The data link address of all 1's is the broadcast address: any packet sent to such an address is received by all stations on a LAN (it is the ultimate multicast address!).



Q2.1 - ARP

> What happens when another IP endnode D decides to send a packet to IP endnode A? Assume that D initially does not have A's data link address in its cache, and so must do the ARP protocol. Give the sequence of events.

1. D broadcasts an ARP request on its with it's IP addr, it's MAC addr, and A's IP addr, and the broadcast MAC (all 1s)
2. The bridge reads the line and observes that the destination IP for the ARP request is on the other line and forwards it to A's LAN
3. The ARP request is broadcast on A's LAN and A correctly receives the ARP request
4. A's bad configuration causes it to send an ARP reply on its LAN with its IP addr, the broadcast MAC (its

misconfigured MAC addr), D's IP addr, and D's MAC addr

5. The bridge routes the ARP reply from A to D
6. D now caches A's MAC addr as the broadcast addr
7. Now any data packets D transmits to A will contain the destination MAC addr as the broadcast addr so every node on D's LAN and every node on A's LAN will receive every data packet D transmits

Given the mailbox example above, if any node on any of the LANs tries to retransmit or forward any data packet from D to A, it will mistakenly broadcast the packet to all nodes on all connected LANs because of A's misconfigured MAC addr. This may continue indefinitely depending on how each node deals with receiving a packet not addressed to its IP.

Q2.2 - IP Router

> Suppose the bridge is replaced by an IP router. (Of course, the masks at the nodes must be changed so that there are now two masks, one for each LAN. Note a mask is just a bitmap of 1’s in the most significant bits that tells you how long the prefix for that subnet is) The problem does not disappear, but it does get a little better. Explain as precisely as you can the improvement using two parameters: T the total number of endnodes in the network of Ethernets, and M the maximum number of endnodes in a single Ethernet.

Now that the bridge is replaced by an IP router, it means that each LAN on either side of the router is on a distinct subnet. This means that data packets with the broadcast MAC will be constrained to broadcasting within their own subnet. In terms of T and M, the bridged network enabled broadcasts to occur across the bridge allowing T endnodes to receive the mistakenly broadcasted packet. However, with the router network, the broadcast packet is limited to its subnet limiting the number of devices that receive the broadcasted packet(s) to at most M.

Question 3 - IP Forwarding

To prepare for the Arista interview, you give ChatGPT the prompt:

> “Can you write C code to do longest matching prefix of a 32-bit destination IP address in an Ethernet packet with an IP routing header which contains the destination IP address at the right offset. Do longest matching prefix on a database of prefixes called ForwardingTable implemented as a binary trie. The result should return the output interface and IP address of the next hop.”

ChatGPT's reply:

> Here's a basic implementation of a longest prefix match using a binary trie data structure in C. This code will parse the destination IP address from the Ethernet packet, traverse the trie to find the longest matching prefix, and return the output interface and next hop IP address.
> In this example, I assume:

1. Each trie node has children for 0 and 1 bits.
2. Each leaf node contains the next hop information, including the output interface and next hop IP.
3. The trie is built ahead of time and passed to the longest prefix match function.
   Here's the code:

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

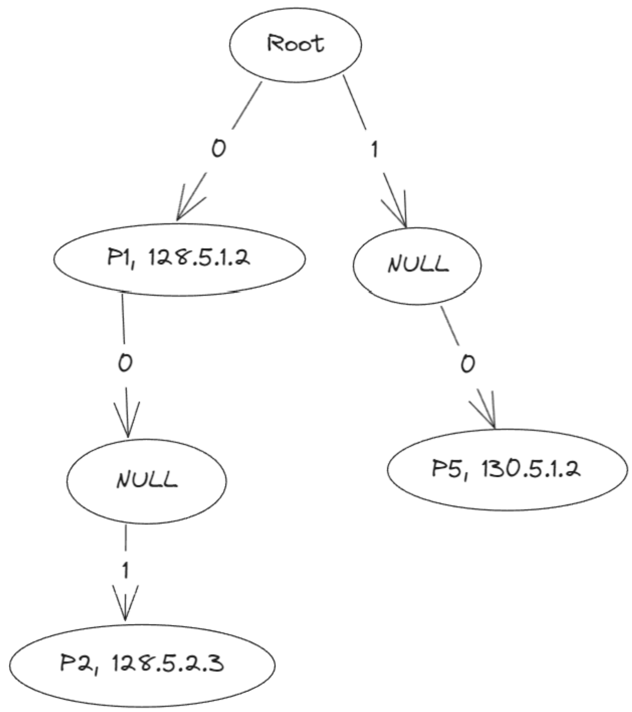
```c
// Define a structure for the trie node
typedef struct TrieNode
{
    struct TrieNode *children[2]; // 0 and 1 children
    int is_end_of_prefix;         // Q2: Why is this field needed?
    uint32_t next_hop_ip;         // IP address of the next hop
    int output_interface;         // Output interface
} TrieNode;

// Initialize a new trie node
TrieNode *create_node()
{
    TrieNode *node = (TrieNode *)malloc(sizeof(TrieNode));
    node->children[0] = node->children[1] = NULL;
    node->is_end_of_prefix = 0;
    node->next_hop_ip = 0;
    node->output_interface = -1;
    return node;
}

// Insert a prefix into the trie
void insert_prefix(TrieNode *root, uint32_t prefix, int
prefix_length, uint32_t next_hop_ip, int output_interface)
{
    TrieNode *node = root;
    for (int i = 31; i >= 32 - prefix_length; i--)
    {
        int bit = (prefix >> i) & 1;
        if (!node->children[bit])
        {
            node->children[bit] = create_node();
        }
        node = node->children[bit];
    }
    node->is_end_of_prefix = 1; // Q3: Why do this?
    node->next_hop_ip = next_hop_ip;
    node->output_interface = output_interface;
}

// Perform the longest prefix match
int longest_prefix_match(TrieNode *root, uint32_t dest_ip,
```

```c
uint32_t *next_hop_ip, int *output_interface)
{
    TrieNode *node = root;
    TrieNode *last_matching_node = NULL;
    for (int i = 31; i >= 0; i--)
    {
        int bit = (dest_ip >> i) & 1;
        if (!node->children[bit])
            break;
        node = node->children[bit];
        if (node->is_end_of_prefix)
        {
            last_matching_node = node;
        }
    }
    if (last_matching_node)
    { // Q4: Why do this?
        *next_hop_ip = last_matching_node->next_hop_ip;
        *output_interface = last_matching_node->output_interface;
        return 1; // Match found
    }
    return 0; // No match found
}
typedef struct
{
    uint8_t destination_mac[6];
    uint8_t source_mac[6];
    uint16_t ethertype;
    uint8_t payload[1500];
} EthernetPacket;

// Extract destination IP from an IP header within the Ethernet payload
uint32_t extract_destination_ip(EthernetPacket *packet)
{
    return ((uint32_t)packet->payload[16] << 24) |
           ((uint32_t)packet->payload[17] << 16) |
           ((uint32_t)packet->payload[18] << 8) |
           ((uint32_t)packet->payload[19]);
} // Q5: Explain why the code uses the constants 16, 17, 18, 19. Explain shifts and ORs.
```

```c
int main()
{
    TrieNode *root = create_node();

    // Example: Insert prefixes into the trie
    insert_prefix(root, 0xC0A80000, 16, 0xC0A80001, 1);
    // 192.168.0.0/16 -> next hop 192.168.0.1, interface 1
    insert_prefix(root, 0xC0A80100, 24, 0xC0A80101, 2);
    // 192.168.1.0/24 -> next hop 192.168.1.1, interface 2

    // Example Ethernet packet with destination IP 192.168.1.5
    EthernetPacket packet;
    memset(&packet, 0, sizeof(EthernetPacket));
    packet.payload[16] = 192;
    packet.payload[17] = 168;
    packet.payload[18] = 1;
    packet.payload[19] = 5;

    uint32_t dest_ip = extract_destination_ip(&packet);
    uint32_t next_hop_ip;
    int output_interface;

    if (longest_prefix_match(root, dest_ip, &next_hop_ip,
&output_interface))
    {
        printf("Match found:\n");
        printf("Next hop IP: %u.%u.%u.%u\n",
               (next_hop_ip >> 24) & 0xFF,
               (next_hop_ip >> 16) & 0xFF,
               (next_hop_ip >> 8) & 0xFF,
               next_hop_ip & 0xFF);
        printf("Output interface: %d\n", output_interface);
    }
    else
    {
        printf("No matching prefix found.\n");
    }
    // Free trie memory (not shown here)
    return 0;
}
```

Q3.1 - Trie Structure

> Draw an example of the trie data structure after you insert three prefixes:
> `0*` whose next hop is `128.5.1.2` and interface is `P1`,
> `001*` whose next hop is `128.5.2.3` and interface is `P2`, and
> `10*` whose next hop is `130.5.1.2` and interface is `P5`.



Q3.2 - End of Prefix

> What is the point of `is_end_of_prefix` (see comment)? This seems useless at first because the loops for insertion and search terminate when they see a null pointer. Drawing the picture for part 1 will help as it will for any code you get from ChatGPT.

The `is_end_of_prefix` attribute of the nodes in the trie inform the prefix matching algorithm if a particular node is a valid end to a prefix. This is important because although we may get addresses masked to some prefix, we want to ensure the prefix matching will not accidentally select a NULL node as the matched prefix. In this case, we would want to treat the match the same as if the prefix were not in the trie to begin with.

Q3.3 - Resetting End of Prefix

> Again, when you insert a prefix the code sets `is_end_of_prefix` to 1 (see comment). Why is that needed and why does it not need to be reset?

Taking the example from the outputted graph in Q3.1, it's possible to make NULL nodes for prefixes like `00*` which should not actually count as a matched prefix despite a node existing for that prefix branch. It doesn't need to be reset (e.g. on P1) when new nodes, like P2, are created because the algorithm stops when it gets to the end of the address input anyways so it only needs to check whether the end of the input address matches with a node marked as the end of a stored prefix.

Q3.4 - Last Matching Node

> At the end of search (see comment), the code retrieves the next hop from `last_matching_node`. Why not pick it up directly as the code walks the trie?

The tree that ChatGPT's code generates does not ensure that the next hop addresses follow the prefix pattern. We can see this in the example where if the algo picked up the next hop address by walking the branches, it would pick up `128.5.1.2` (which should match the entire address), but we see that there exists a later prefix match with more prefix values that has a next hop with different values for the last 16 bits of the address (`128.5.2.3`). So, the `last_matching_node` attribute is required to ensure the correct address is picked. This could be changed if the address itself was dispersed along the prefix trie.

Q3.5 - IP Addresses

> The code that retrieves the Dest IP address (see comment) has some interesting constants including 16, 17, 18, and 19. It also does some shifting and ORing. Explain why.

The shifting and ORing construct the address from the payload. The byte selection with 16, 17, 18, and 19 selects the byte of the address (e.g., in `128.5.1.2` we can extract `128`, `5`, `1`, `2`). Shifting by the correct amount of bytes allows us to position the address bytes in the correct positions, padding with 0s. The ORing then constructs the address in a single value of 4 bytes.

Q3.6 - Missing Code

> Explain what code may be missing besides deallocating memory.

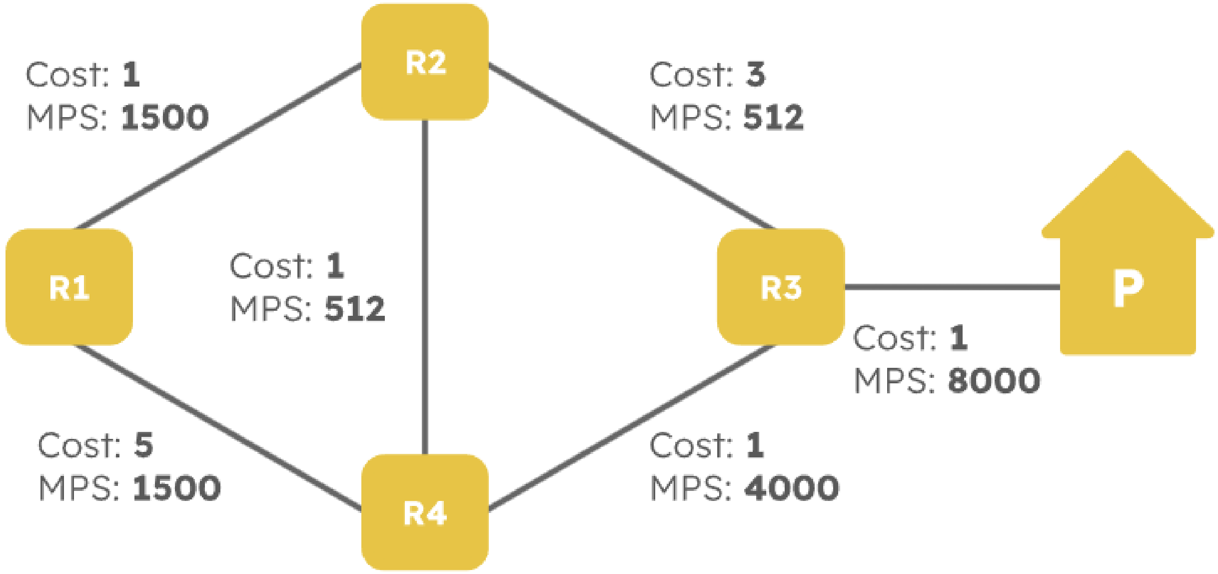
The big functionality missing is removing prefixes/nodes. We are able to create and modify nodes with the `insert_prefix` function but we can't actually remove nodes or reorganize the branch without reinitializing the entire trie. There is also no support for multi-bit tries and no error handling for bad address, interfaces, or prefixes.

Question 4 - Routing Fragmentation

> We learned in class that in order to avoid fragmentation, IP tries various packet sizes until it finds one that works. Alternatively, we could modify the routing protocol to compute the minimum of the maximum packet sizes of all links on the best route to each destination.

In distance vector routing, a router **R** computes its own distance **Distance(P, R)** to a destination prefix **P** using the distances sent by its neighbors as follows: **Distance(P, R) =**

Minimum across all neighbors **N** of **Distance(P, N) + Distance(R, N)**. We want to see how to modify this protocol to also compute the minimum max packet size on the shortest distance route to **P**.



Q4.1 - Shortest Path

> What is the shortest path between R1 and P? What is the largest packet size that is guaranteed to get through without fragmentation on this path?

The shortest path `R1→P = R1→R2→R4→R3→P`
The largest packet size guaranteed to route `R1→P` is the minimum MPS across the path: 512.

Q4.2 - Min Max Packet Size

> Assume each router has receives the additional variables Distance(P, N) and MinMaxPacketSize (P, N) from each neighbor N. Write an equation to compute these two variables from the corresponding variables of all a router's neighbors.

For a router R that has N edges with values $\text{Cost}(R, n)$, $\text{MPS}(R, n)$ for each of $n \in N$ neighbors:

$$\mathrm{Distance}(P,R) = \min_{n\in N}\Big(\mathrm{Cost}(R,n) + \mathrm{Distance}(P,n)\Big)$$

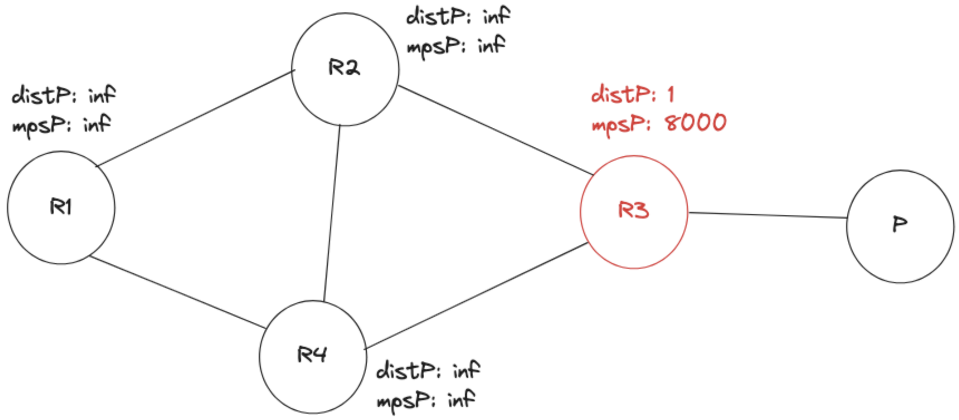
$$\mathrm{MinMaxPacketSize}(P,R) = \min_{n\in N}\Big(\mathrm{MPS}(R,n) + \mathrm{MinMaxPacketSize}(P,n)\Big)$$

Q4.3 - Packet Size Propagation

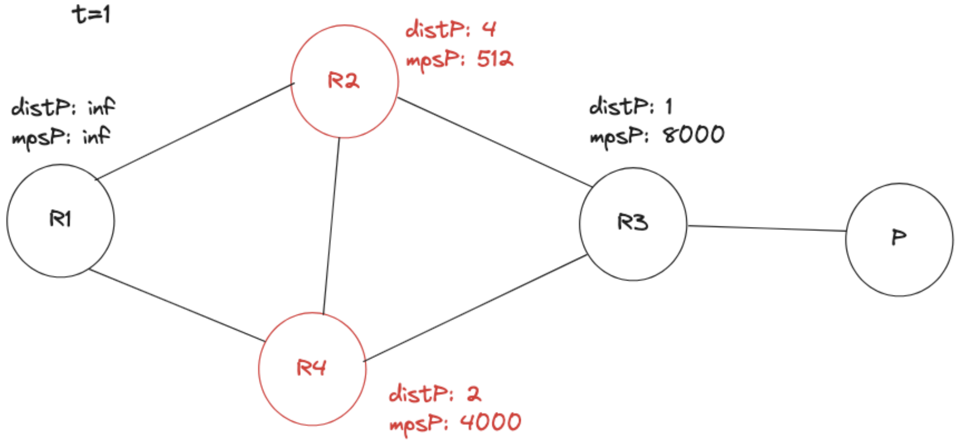
> Assume we are calculating these estimates only for distances to P. Assume that at t = 0, router R3 has Distance (P, R3) = 1 and MinMaxPacketSize (P, R3) = 8000 and all other routers have the distance and min packet size to P set to a default of infinity. Assume that at t = 0, each router sends an update to each neighbor. Draw several pictures of the same topology with the changing estimates of each router for its two variables based on the equation you wrote down until all estimates stop changing. After how much time do all the estimates converge (i.e. do not change any more)?

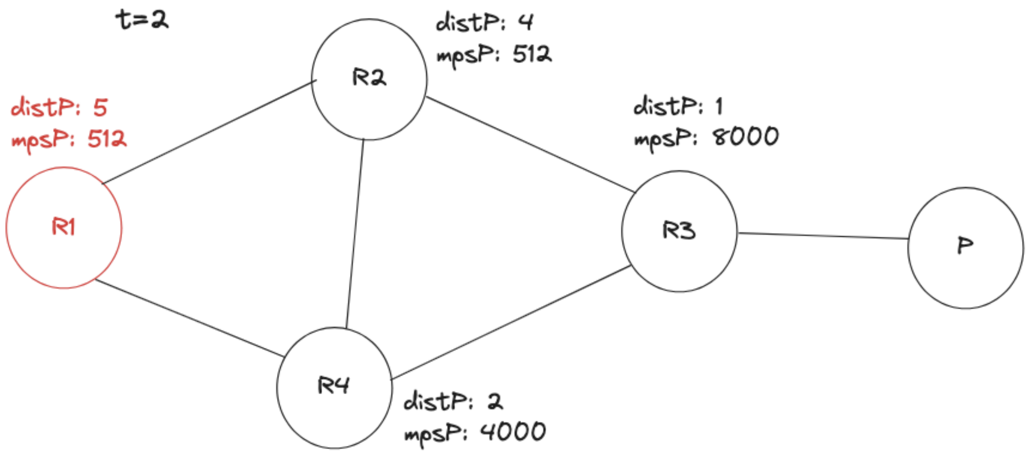
I've only highlighted the time steps in which distance vector propagation causes the Distance to P and Minimum MPS to P to change. There are some timesteps after that propagate the latest changed node's distance vectors but does not result in the state of other nodes changing.

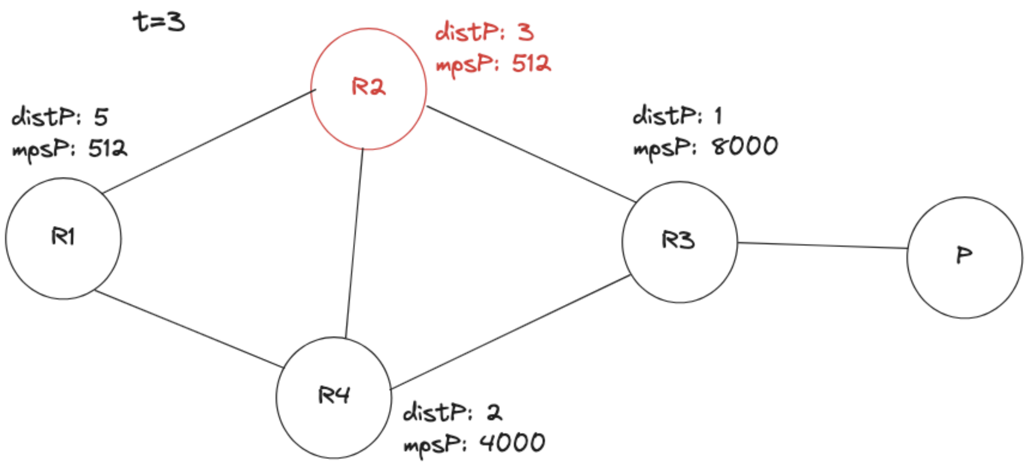
t=0

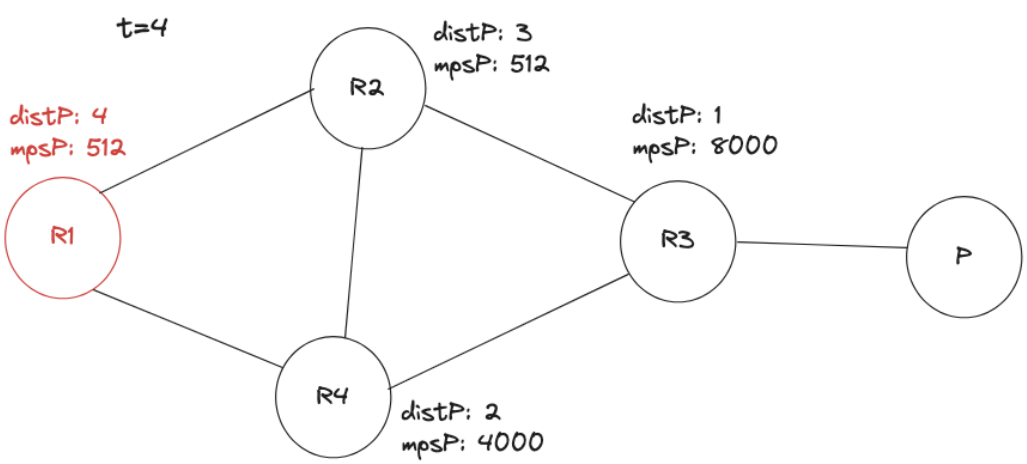

distP: inf
mpsP: inf
R2
distP: inf
mpsP: inf
R1
distP: 1
mpsP: 8000
R3
P
R4
distP: inf
mpsP: inf

t=1


distP: 4
mpsP: 512
R2
distP: inf
mpsP: inf
R1
distP: 1
mpsP: 8000
R3
P
R4
distP: 2
mpsP: 4000







Q4.4 - Crash

> Now assume the link to R2 to R4 crashes at time t = 7. Draw similar pictures for the time it takes to converge after the crash.

I've only highlighted the time steps in which distance vector propagation causes the Distance to P and Minimum MPS to P to change. There are some timesteps after that propagate the latest changed node's distance vectors but does not result in the state of other nodes changing.