CSPC53 Programming Assignment, Simulation of Flooding Routing Algorithm with solutions for overcoming looping

106119029, Dipesh Kafle

Contents

Code Link 1
Flooding routing techniques 1
Input and Output 2
Output(TTL Based Flooding Simulation) 4
Output (SNCF Simulation) 5
Code 5
Running instruction 5
Components(Classes/Structs) I Have in Code 6
TTLBased Flooding 7
Code 7
SNCF (Sequence Number Controlled Flooding) 10
Code 10
Components(Classes/Structs) I Have in Code 13
`network_node` 13
`Router` 14
`Host` 15
`IP` 17
`data` 18
`Segment` 18
`Packet` 19
Logging Functions 20

Code Link

Link for all codes

Flooding routing techniques

I'm using TTL(Max Hop Count) based flooding and SNCF(Sequence Number Controlled Flooding) in my simulation. SNCF will overcome looping issues by not allowing a packet to circulate more than once from a particular source.

Flooding with a hop count can produce an exponential number of duplicate packets as the hop count grows and routers duplicate packets they have seen before. A better technique for damming the flood is to have routers keep track of which packets have been flooded, to avoid sending them out a second time. One way to achieve this goal is to have the source router put a sequence number in each packet it receives from its hosts.

Input and Output

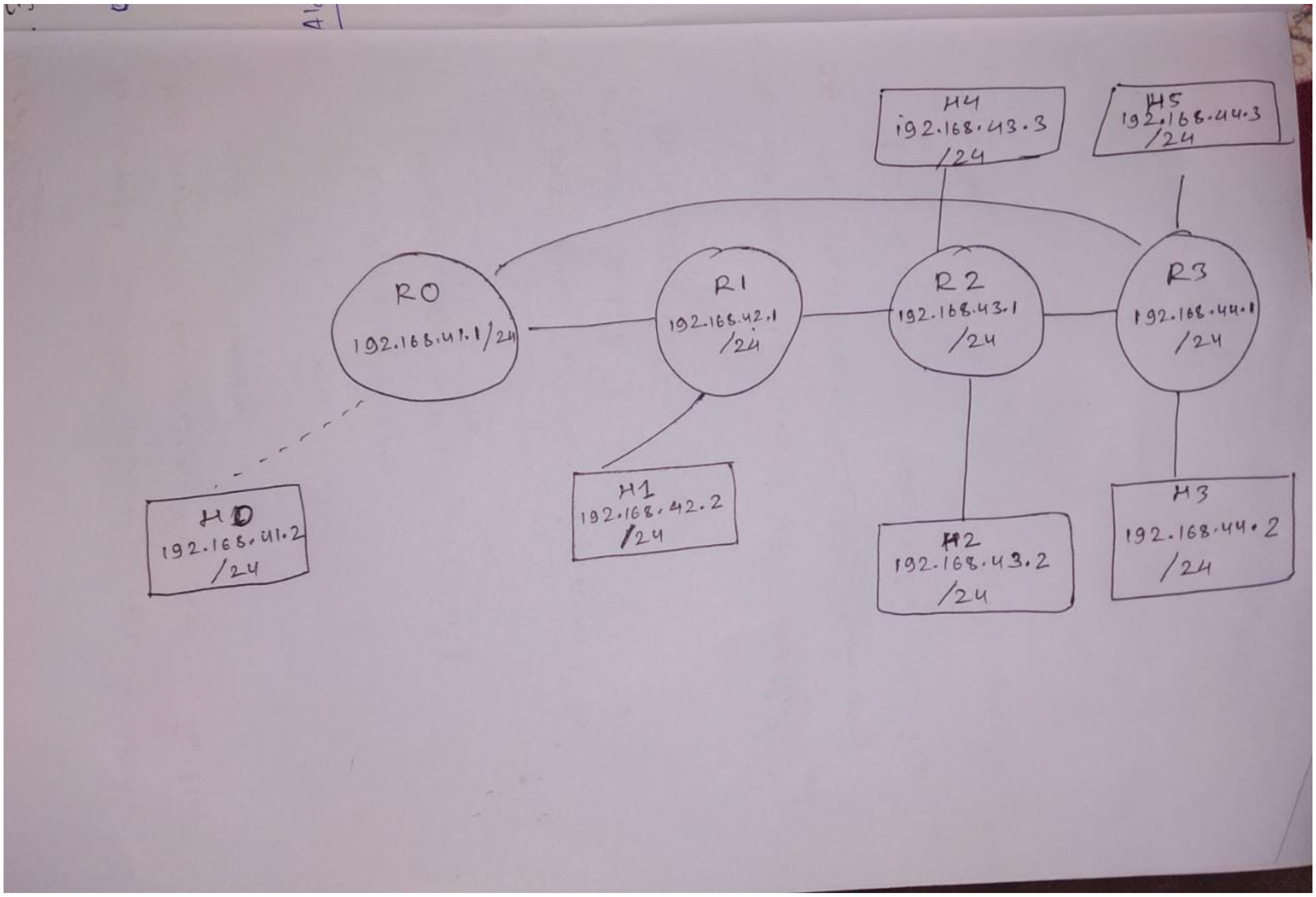
The input format for input.txt file is:

- `ttl`
- `Host Count`
- `Router Count`
- `Edges count`
- `Source Host Destination Host`
- `Message to send`
- `Host Count number of lines with format IP followed by mask`
- `Router Count number of lines with format IP followed by mask`
- `Edges Count number of lines with in format Rx Hy or Rx Ry`

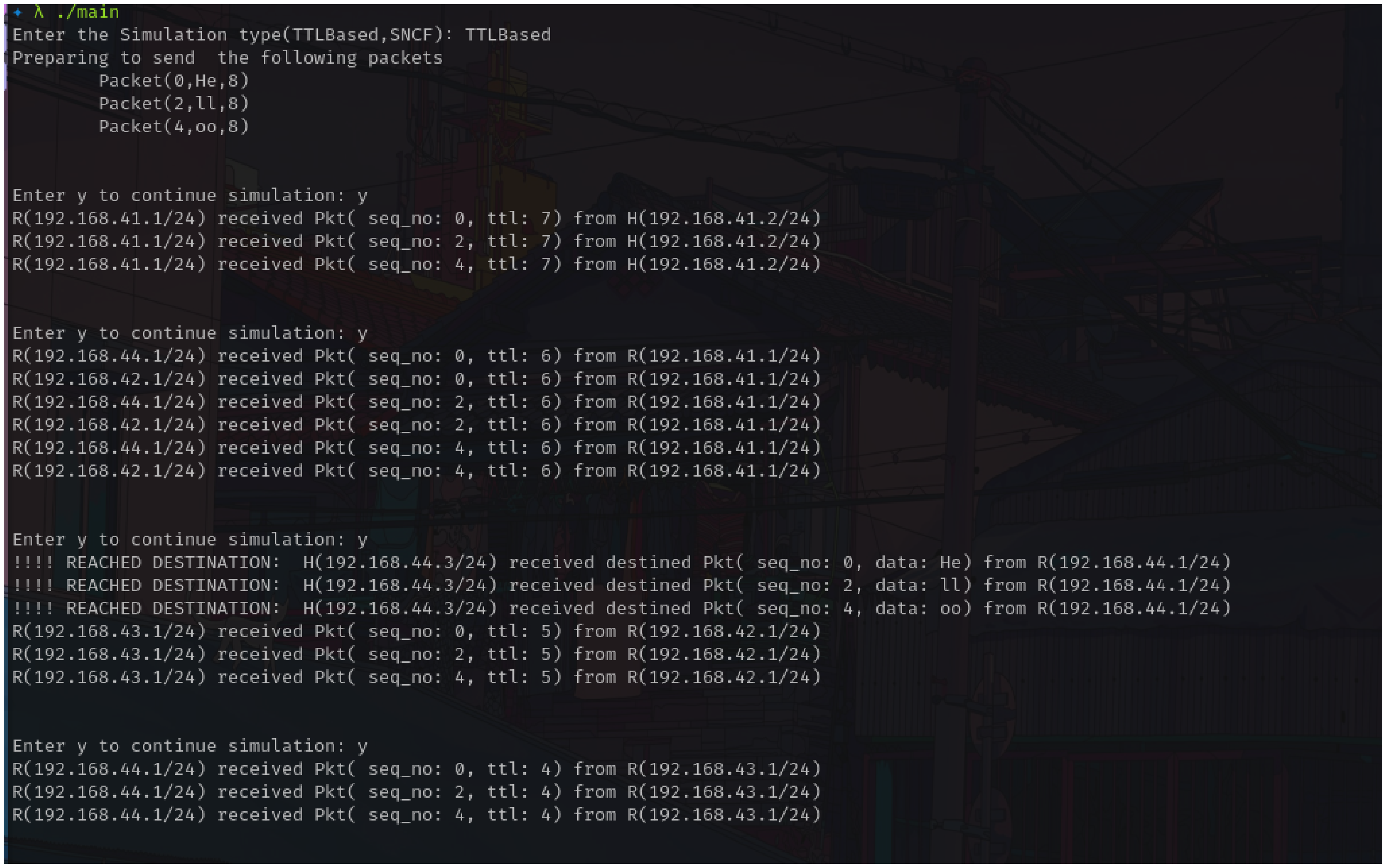
input.txt

```
8
6
4
10
H0 H5
Helloo
192.168.41.2 24
192.168.42.2 24
192.168.43.2 24
192.168.44.2 24
192.168.43.3 24
192.168.44.3 24
192.168.41.1 24
192.168.42.1 24
192.168.43.1 24
192.168.44.1 24
R0 H0
R1 H1
R2 H2
R3 H3
R0 R1
R1 R2
R2 R3
R0 R3
R2 H4
```

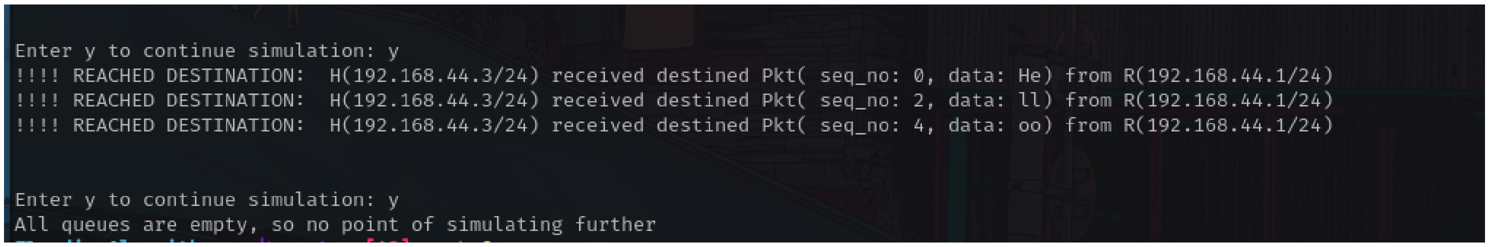
R3 H5


H4
192.168.43.3
/24
H5
192.168.44.3
/24
R0
192.168.41.1/24
R1
192.168.42.1
/24
R2
192.168.43.1
/24
R3
192.168.44.1
/24
H0
192.168.41.2
/24
H1
192.168.42.2
/24
H2
192.168.43.2
/24
H3
192.168.44.2
/24

Output(TTL Based Flooding Simulation)

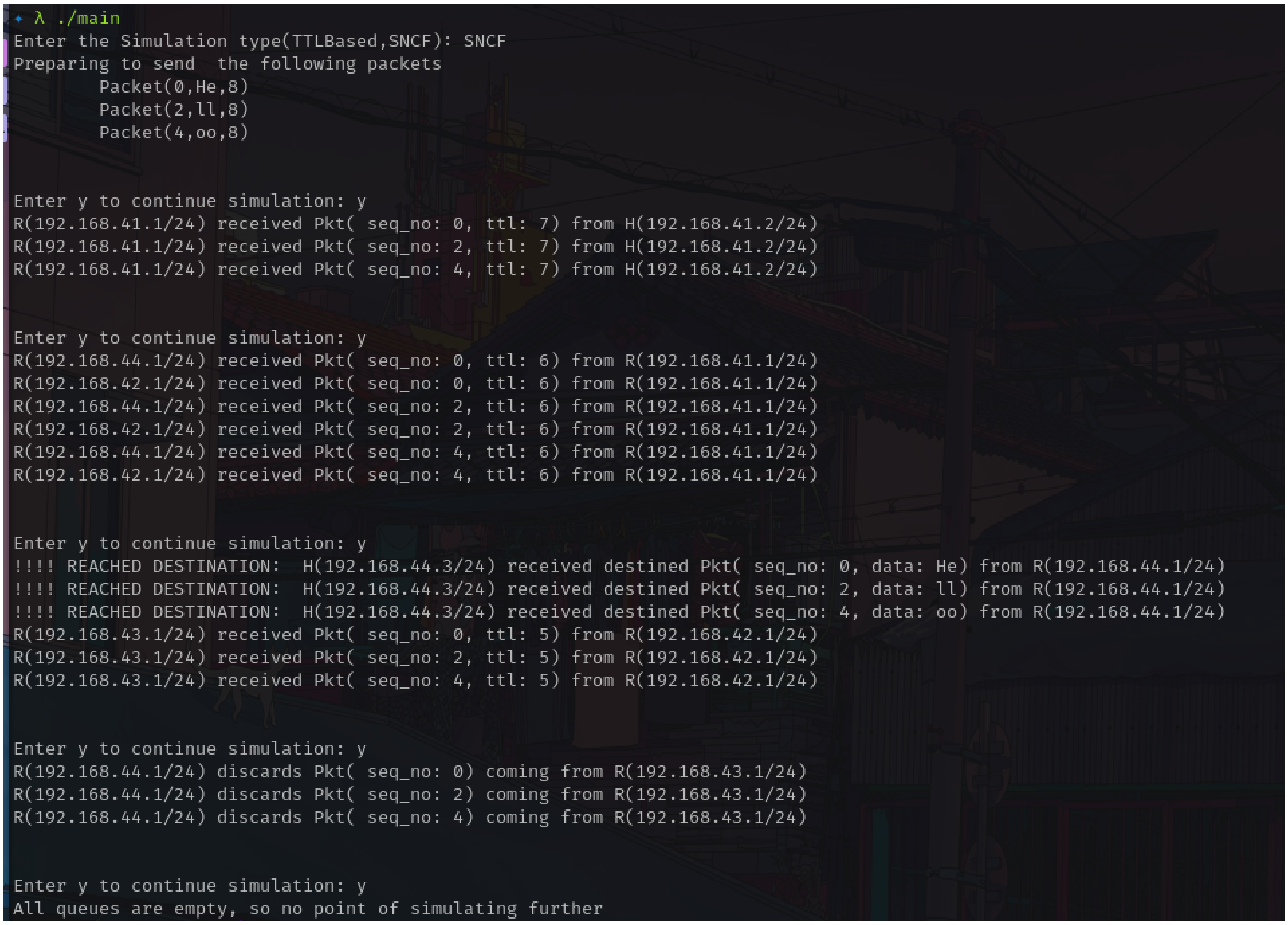


```
✦ λ ./main
Enter the Simulation type(TTLBased,SNCF): TTLBased
Preparing to send  the following packets
        Packet(0,He,8)
        Packet(2,ll,8)
        Packet(4,oo,8)


Enter y to continue simulation: y
R(192.168.41.1/24) received Pkt( seq_no: 0, ttl: 7) from H(192.168.41.2/24)
R(192.168.41.1/24) received Pkt( seq_no: 2, ttl: 7) from H(192.168.41.2/24)
R(192.168.41.1/24) received Pkt( seq_no: 4, ttl: 7) from H(192.168.41.2/24)


Enter y to continue simulation: y
R(192.168.44.1/24) received Pkt( seq_no: 0, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 0, ttl: 6) from R(192.168.41.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 2, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 2, ttl: 6) from R(192.168.41.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 4, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 4, ttl: 6) from R(192.168.41.1/24)


Enter y to continue simulation: y
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 0, data: He) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 2, data: ll) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 4, data: oo) from R(192.168.44.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 0, ttl: 5) from R(192.168.42.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 2, ttl: 5) from R(192.168.42.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 4, ttl: 5) from R(192.168.42.1/24)


Enter y to continue simulation: y
R(192.168.44.1/24) received Pkt( seq_no: 0, ttl: 4) from R(192.168.43.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 2, ttl: 4) from R(192.168.43.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 4, ttl: 4) from R(192.168.43.1/24)
```



```
Enter y to continue simulation: y
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 0, data: He) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 2, data: ll) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 4, data: oo) from R(192.168.44.1/24)


Enter y to continue simulation: y
All queues are empty, so no point of simulating further
```

Output (SNCF Simulation)



```
✦ λ ./main
Enter the Simulation type(TTLBased,SNCF): SNCF
Preparing to send  the following packets
        Packet(0,He,8)
        Packet(2,ll,8)
        Packet(4,oo,8)


Enter y to continue simulation: y
R(192.168.41.1/24) received Pkt( seq_no: 0, ttl: 7) from H(192.168.41.2/24)
R(192.168.41.1/24) received Pkt( seq_no: 2, ttl: 7) from H(192.168.41.2/24)
R(192.168.41.1/24) received Pkt( seq_no: 4, ttl: 7) from H(192.168.41.2/24)


Enter y to continue simulation: y
R(192.168.44.1/24) received Pkt( seq_no: 0, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 0, ttl: 6) from R(192.168.41.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 2, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 2, ttl: 6) from R(192.168.41.1/24)
R(192.168.44.1/24) received Pkt( seq_no: 4, ttl: 6) from R(192.168.41.1/24)
R(192.168.42.1/24) received Pkt( seq_no: 4, ttl: 6) from R(192.168.41.1/24)


Enter y to continue simulation: y
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 0, data: He) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 2, data: ll) from R(192.168.44.1/24)
!!!! REACHED DESTINATION:  H(192.168.44.3/24) received destined Pkt( seq_no: 4, data: oo) from R(192.168.44.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 0, ttl: 5) from R(192.168.42.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 2, ttl: 5) from R(192.168.42.1/24)
R(192.168.43.1/24) received Pkt( seq_no: 4, ttl: 5) from R(192.168.42.1/24)


Enter y to continue simulation: y
R(192.168.44.1/24) discards Pkt( seq_no: 0) coming from R(192.168.43.1/24)
R(192.168.44.1/24) discards Pkt( seq_no: 2) coming from R(192.168.43.1/24)
R(192.168.44.1/24) discards Pkt( seq_no: 4) coming from R(192.168.43.1/24)


Enter y to continue simulation: y
All queues are empty, so no point of simulating further
```

- In SNCF , the packets are also not getting delivered to the same destination twice unlike in TTL Based Simulation.

Code

Running instruction

Link for all codes

- After getting all codes, do the following:
  - For Linux:

    `g++ -o main -std=c++14 main.cpp && ./main`

– For Windows:

```
g++ -o main -std=c++14 main.cpp && .\main
```

Components(Classes/Structs) I Have in Code

These are the classes that I have used in my code mainly.

- `data` : defined in `application_layer.hpp`
- `Segment` : defined in `transport_layer.hpp`
- `Packet` : defined in `network_layer.hpp`
- `IP` : defined in `network_layer.hpp`
- `network_node` : defined in `network.hpp`
- `Router` : defined in `network.hpp` and inherits from network_node
- `Host` : defined in `network.hpp` and inherits from network_node

In my code, I'm doing a automata based approach. I'll do iteration of simulation. After every simulation, the queues of all the nodes(Router and Host) will be updated and get ready for next simulation.

I tried to simulate how the message gets split as well. Basically whatever message is to be sent, first we'll make application_layer `data` out of it. After that in network layer we'll split the data into multiple segments(the size of data in segment is decided by mss variable in the `Segment` class). After all the segments(they'll have a sequence number too, which indicates the byte at which they're starting) are created, we'll enclose the segment in `Packet` with some more detail. I felt like going below network layer would overcomplicate the things, so only simulated the process till this there.

Hosts can be connected to multiple routers and the Routers can be connected to other routers and will have hosts that are connected to them. These are the details that are stored in Host and Router class along with network_queue.

- `simulate` function

There's a temporary queue instantiated for every Router at the start of the function. I'll then call functions that handles and process the hosts and routers and fill the temporary queue for the next operation. I had to resort to using temporary queue because it will otherwise disturb the current simulation. After that is done, I'll fill the queue associated with every router finally and exit the function. This way all the things are ready for the next simulation. This function will take in `handle_hosts` and `handle_routers` function and call them with the temporary queue and they'll put whatever they have to put in that queue and log all the activities done. The `handle_hosts` and `handle_routers` function are shown in next section.

```
using arg_type = unordered_map<network_node, vector<PacketAndSrc>>;

void simulate(void (*handle_hosts)(arg_type &),
              void (*handle_routers)(arg_type &)) {

  arg_type router_queue_for_next_time_unit;

  handle_hosts(router_queue_for_next_time_unit);
  handle_routers(router_queue_for_next_time_unit);

  for (auto &p : Router::routers) {
    auto router = p.second;
    const network_node &router_as_node = *router;
    router->network_queue = queue<PacketAndSrc>();
    for (auto &x : router_queue_for_next_time_unit[router_as_node]) {
      router->network_queue.push(x);
    }
  }
  //
}
```

TTLBased Flooding

This code can be found in `simulation_with_ttl_count.hpp` file

Code

What does a host do in simple flooding(having a fixed ttl)?

- For a packet to be sent by the host, I'm already checking if its destination lies in the same network. This I did in put_data_to_network_queue method of the Host class itself.
- So we know for sure whatever data is in the network_queue, it is supposed to go through a router.
- So its purpose is to go through all the routers and flood the packets, after decrementing ttl by 1 .

What does a router do in simple flooding(having a fixed ttl)?

- It will check if the packet that it has in its queue is supposed to be given to a host which it is connected with.
- If thats the case it will send the packet to that Host
- Else, It will flood the packet to all the routers in its reach, after decrementing ttl by 1 for the packet

```
namespace ttl_based {

inline void handle_hosts(unordered_map<network_node, vector<PacketAndSrc>>
                             &router_queue_for_next_time_unit) {

  /*
   * What does a host do in simple flooding(having a fixed ttl)?
   *
   * For a packet to be sent by the host, I'm already checking if its
   * destination lies in the same network. This I did in
   * put_data_to_network_queue method of the Host class itself.
   *
   * So we know for sure whatever data is in the network_queue, it is supposed
   * to go through a router.
   *
   * So its purpose is to go through all the routers and flood the packets,
   * after decrementing ttl by 1 .
   *
   *
   */
  // Go through every host in the world
  for (auto &elem : Host::hosts) {
    auto host = elem.second;
    const network_node &host_as_node = *host; // polymorphism at use

    while (!host->network_queue.empty()) {

      if (host->network_queue.front().packet.ttl == 0) {
        host->network_queue.pop();
        continue;
      }
      PacketAndSrc front_packet = host->network_queue.front();

      for (auto router : host->hosts_routers) {

        const network_node &router_as_node = *router; // polymorphism

        // new packet that is to be sent should has ttl one less than the
        // current packet
        Packet new_packet = front_packet.packet;
        new_packet.ttl--;

        // Put the new packet and the source node(i.e the current host ) to
        // temporary queue that will be used in the next simulation
        router_queue_for_next_time_unit[router_as_node].push_back(
            PacketAndSrc(new_packet, host_as_node));

        LOG(*host, *router, new_packet);
      }
      host->network_queue.pop();
    }
  }
}

inline void handle_routers(unordered_map<network_node, vector<PacketAndSrc>>
                               &router_queue_for_next_time_unit) {

  /*
   * What does a router do in simple flooding(having a fixed ttl)?
   *
   * - It will check if the packet that it has in its queue is supposed to be
   * given to a host which it is connected with.
   * - If thats the case it will send the packet to that Host
   * - Else, It will flood the packet to all the routers in its reach, after
   * decrementing ttl by 1 for the packet
   */
  for (auto &elem : Router::routers) {
    auto router = elem.second;

    const network_node &router_as_node = *router; // polymorphism at use

    while (!router->network_queue.empty()) {
      if (router->network_queue.front().packet.ttl == 0) {
        router->network_queue.pop();
        continue;
      }
      PacketAndSrc front_packet = router->network_queue.front();

      bool we_have_the_destination_host = false;

      // Goes through all the  hosts in reach and check if the packet is
      // intended for them. If yes, RECEIVE it and pop from queue
      for (auto host : router->hosts) {
        const network_node &host_as_node = *host; // polymorphism
        if (front_packet.packet.destination == host->ip) {
          we_have_the_destination_host = true;
          // RECEIVE;
          RECEIVE(*router, *host, front_packet.packet);
        }
      }
      if (we_have_the_destination_host) {
        router->network_queue.pop();
        continue;
      }

      // Flood the packet to all the routers that are reachable
      for (auto dest_rtr : router->connected_routers) {
        const network_node &rtr_as_node = *dest_rtr; // polymorphism
        Packet new_packet = front_packet.packet;
        new_packet.ttl--;
        if (front_packet.source.ip != dest_rtr->ip) {
          router_queue_for_next_time_unit[rtr_as_node].push_back(
              PacketAndSrc(new_packet, router_as_node));
          LOG(*router, *dest_rtr, new_packet);
        }
      }

      router->network_queue.pop();
    }
  }
}
```

```
} // namespace ttl_based
```

SNCF (Sequence Number Controlled Flooding)

This code can be found in `sncf_simulation.hpp` file

Code

What does a host do in SNCF ?

- So we know for sure whatever data is in the network_queue, it is supposed to go through a router.
- So its purpose is to go through all the routers and flood the packets, after decrementing ttl by 1 . If the router has already seen the seq_no and source before, it'll ignore

What does a Router do in SNCF ?

- It will check if the packet that it has in its queue is supposed to be given to a host which it is connected with.
- If thats the case it will send the packet to that Host
- Else, It will flood the packet to all the routers in its reach, after decrementing ttl by 1 for the packet. But the destination router will check the seen_sequences first to make sure the seq_no and source is not seen already. If it has already seen it, it'll discard. Else it'll put to its network queue

```
using seq_number = int;

template <> struct std::hash<pair<network_node, seq_number>> {
  size_t operator()(const pair<network_node, seq_number> &p) const noexcept {
    return std::hash<network_node>()(p.first);
  }
};

namespace sncf {

// Global variable to keep track of all the seen sequences of all the routers
std::unordered_map<Router *, unordered_set<pair<network_node, seq_number>>>
    seen_sequences;

inline void handle_hosts(unordered_map<network_node, vector<PacketAndSrc>>
                             &router_queue_for_next_time_unit) {
  /*
   * What does a host do in SNCF ?
   *
   * So we know for sure whatever data is in the network_queue, it is supposed
   * to go through a router.
   *
   * So its purpose is to go through all the routers and flood the packets,
   * after decrementing ttl by 1 . If the router has already seen the seq_no and
   * source before, it'll ignore
   *
   */

  // Go through every host in the world
  for (auto &elem : Host::hosts) {
    auto host = elem.second;

    const network_node &host_as_node = *host; // polymorphism at use

    // Check the hosts network queue
    // Process every request in the queue
    while (!host->network_queue.empty()) {
      if (host->network_queue.front().packet.ttl == 0) {
        host->network_queue.pop();
        continue;
      }

      PacketAndSrc front_packet_and_src = host->network_queue.front();
      for (auto router : host->hosts_routers) {
        const network_node &router_as_node = *router; // polymorphism

        if (seen_sequences[router].find(
                {network_node(front_packet_and_src.packet.source),
                 front_packet_and_src.packet.segment.seq_no}) !=
            seen_sequences[router].end()) {
          continue;
        }

        Packet new_packet = front_packet_and_src.packet;
        new_packet.ttl--;

        router_queue_for_next_time_unit[router_as_node].push_back(
            PacketAndSrc(new_packet, host_as_node));

        seen_sequences[router].insert(
            {network_node(front_packet_and_src.packet.source),
             front_packet_and_src.packet.segment.seq_no});

        LOG(*host, *router, new_packet);
      }
      host->network_queue.pop();
    }
  }
}

inline void handle_routers(unordered_map<network_node, vector<PacketAndSrc>>
                               &router_queue_for_next_time_unit) {

  /*
   * What does a Router do in SNCF ?
   *
   * - It will check if the packet that it has in its queue is supposed to be
   * given to a host which it is connected with.
   * - If thats the case it will send the packet to that Host
   * - Else, It will flood the packet to all the routers in its reach, after
   * decrementing ttl by 1 for the packet. But the destination router will check
   * the seen_sequences first to make sure the seq_no and source  is not seen
   * already. If it has already seen it, it'll discard. Else it'll put to its
   * network queue
   *
   */

  for (auto &elem : Router::routers) {
    auto router = elem.second;

    const network_node &router_as_node = *router; // polymorphism at use

    while (!router->network_queue.empty()) {
      bool we_have_the_destination_host = false;

      for (auto host : router->hosts) {
        const network_node &host_as_node = *host; // polymorphism
        PacketAndSrc front_packet = router->network_queue.front();
        if (front_packet.packet.destination == host->ip) {
          we_have_the_destination_host = true;
          // RECEIVE;
          RECEIVE(*router, *host, front_packet.packet);
        }
      }

      if (router->network_queue.front().packet.ttl == 0) {
        router->network_queue.pop();
        continue;
      }

      PacketAndSrc front_packet_and_src = router->network_queue.front();
      if (!we_have_the_destination_host) {
        for (auto dest_rtr : router->connected_routers) {
          const network_node &rtr_as_node = *dest_rtr; // polymorphism

          Packet new_packet = front_packet_and_src.packet;
          new_packet.ttl--;
```

```
          if (front_packet_and_src.source.ip != dest_rtr->ip) {

            if (seen_sequences[dest_rtr].find(
                    {network_node(front_packet_and_src.packet.source),
                     front_packet_and_src.packet.segment.seq_no}) !=
                seen_sequences[dest_rtr].end()) {
              DISCARD(*dest_rtr, *router, front_packet_and_src.packet);
              continue;
            }

            router_queue_for_next_time_unit[rtr_as_node].push_back(
                PacketAndSrc(new_packet, router_as_node));
            seen_sequences[dest_rtr].insert(
                {network_node(front_packet_and_src.packet.source),
                 front_packet_and_src.packet.segment.seq_no});
            LOG(*router, *dest_rtr, new_packet);
          }
        }
      }

      router->network_queue.pop();
    }
  }
}

}; // namespace sncf
```

Components(Classes/Structs) I Have in Code

`network_node`

```
struct network_node {
  network::IP ip;
  /*
   * This is a network queue
   *
   * The packets that the particular network_node(be that router or host) has to
   * process are in this
   *
   * If the queue has something in it it will immediately be processed during
   * the simulation. There's no concept of dropping packets and bandwith in the
   * miniature simulation that I'm doing
   *
   */
  queue<PacketAndSrc> network_queue;

  /*
```

```
   * Copy constructor for network_node class
   */
  network_node(network::IP ip) : ip(ip) {}

  /*
   * Equal to comparison operator for network_node, it basically compares the
   * IPs of two network_node objects
   */
  bool operator==(const network_node &other) const { return ip == other.ip; }

  /*
   * This will use subnet mask and IP to determine whether two network_node are
   * in the same network
   *
   * Since Router and Host classes both derive from network_node, we can use
   * polymorphism here to know whether those belong to same network
   */
  static bool is_same_network(const network_node &node1,
                              const network_node &node2) {
    auto x = node1.ip.ip & ((1 << 31) >> (node1.ip.n - 1));
    auto y = node2.ip.ip & ((1 << 31) >> (node2.ip.n - 1));
    return x == y;
  }
};
```

Router

```
struct Router : network_node {
  /*
   * This is something I'm using to keep track of all the Routers that have ever
   * been instantiated. This will ensure that I have no two Routers with same IP
   * as well
   */
  static unordered_map<network_node, Router *> routers;

  // All the hosts that the router is connected to
  std::unordered_set<Host *> hosts;
  // All the routers that this router has direct connection to
  std::unordered_set<Router *> connected_routers;

  Router(network::IP ip) : network_node(ip) {
    // This is similar to what I am doing in Host's constructor
    // To ensure no duplicacy
    if (routers.find(static_cast<network_node>(*this)) != routers.end()) {
      throw std::runtime_error("A Router with same IP already exists.\n"
                               "There cant be two routers with same IP\n"
```

```
                              "It will cause issues\n");
    }
    routers[static_cast<network_node>(*this)] = this;
  }
  // Move constructor for router
  Router(Router &&router) : network_node(router.ip) {
    this->network_queue = move(router.network_queue);
    this->connected_routers = move(router.connected_routers);
    this->hosts = move(router.hosts);
    routers[static_cast<network_node>(*this)] = this;
  }

  Router *get() { return this; }

  void connect(Router &r) {
    this->connected_routers.insert(r.get());
    r.connected_routers.insert(this);
  }
  void connect(Host &h) {
    this->hosts.insert(h.get());
    h.hosts_routers.insert(this);
  }
};
```

Host

```
struct Host : network_node {

  /*
   * Stores all the pointers to router the host is connected to
   */
  unordered_set<Router *> hosts_routers; // stores router ids

  /*
   * This is something I'm using to keep track of all the host that have ever
   * been instantiated. This will ensure that I have no two Hosts with same IP
   * as well
   */
  static unordered_map<network_node, Host *> hosts;

  Host(network::IP ip) : network_node(ip) {
    /*
     * I will check if there's already a host with same IP
     * If yes, I will throw error crashing the program
     */
    if (hosts.find(static_cast<network_node>(*this)) != hosts.end()) {
```

```
    throw std::runtime_error("A Host with same IP already exists.\n"
                             "There cant be two hosts with same IP\n"
                             "It will cause issues\n");
  }
  /*
   *Else, I will add the entry of this Host along with its address
   */
  hosts[static_cast<network_node>(*this)] = this;
}

/*
 * Move constructor of Host
 *
 * This is especially useful  when we are using Host with vector i.e
 * vector<Host>
 *
 * vector will resize from time to time if we're pushing to it, and while
 * resizing it needs to move the Host to different memory location. To do that
 * it will invoke its move constructor which should be able to address the
 * memory change and update the hosts unordered_map with new memory address
 */
Host(Host &&h) : network_node(h.ip) {
  this->network_queue = move(h.network_queue);
  this->hosts_routers = move(h.hosts_routers);
  hosts[static_cast<network_node>(*this)] = this;
}

/*
 * Gets the address of the Host
 */
Host *get() { return this; }

/*
 * Takes in raw message,destination and ttl
 *
 * - Converts the message to application layer data
 * - Splits the application layer data to segments of size mss
 * - Converts all the segments to network layer packets
 *
 *  ttl here specifies the maximum hop count a packet can take
 */
void put_data_to_network_queue(string message, Host &destination, int ttl) {
  using application::data;
  using network::Packet;
  using transport::Segment;
```

```
    data msg(message);

    // Divide message into segments of size Segment::mss(maximum segment size)
    vector<Segment> segments = Segment::make_segments(msg);
    vector<Packet> packets;
    // Converts every single segment to packet
    transform(segments.begin(), segments.end(), back_inserter(packets),
              [&](const Segment &segment) {
                return Packet(this->ip, destination.ip, ttl, segment);
              });

    cout << "Preparing to send  the following packets\n";

    for (auto &packet : packets) {
      cout << "\tPacket(" << packet.segment.seq_no << ","
           << packet.segment.get_data_as_string() << "," << packet.ttl << ")\n";
    }

    for (auto &packet : packets) {
      // If the packet's destination is somewhere in the same network,
      // We just log it
      if (packet.destination == this->ip ||
          is_same_network(*this, network_node(packet.destination))) {
        // Logging that the packet is received instantly
        cout << "Host(" << packet.destination.to_cidr() << ") received "
             << packet.segment.get_data_size() << " from "
             << "Host(" << this->ip.to_cidr() << ")\n"
             << "\tinstantly because they're in same network"
                "\n";

      } else { // if it needs to go through router, we put it in network queue
               // the IP 0.0.0.0 was chosen to make it distinct and to ensure no
               // other node in the network had it
        this->network_queue.push(
            PacketAndSrc(packet, network_node(network::IP("0.0.0.0", 32))));
      }
    }
  }
};
```

IP

```
struct IP {
  int32_t ip;
  int n;
  IP(string ip_addr, int n) : n(n) { ip = IP::to_int(move(ip_addr)); }

  static int32_t to_int(string ip_addr) {
    vector<uint32_t> octets;
    auto l = ip_addr.begin();
    auto r = ip_addr.end();
    do {
      auto it = find(l, r, '.');
      auto s = string(l, it);
      octets.push_back(stoi(string(l, it)));
      l = min(++it, r);
    } while (l != r);
    assert(octets.size() == 4);
    return (octets[0] << 24) | (octets[1] << 16) | (octets[2] << 8) |
           (octets[3] << 0);
  }

  string addr_to_string() {
    vector<string> octets;
    octets.push_back(to_string(ip & (0xff)));
    octets.push_back(to_string((ip & (0xff00)) >> 8));
    octets.push_back(to_string((ip & (0xff0000)) >> 16));
    octets.push_back(to_string((ip & (0xff000000)) >> 24));
    reverse(octets.begin(), octets.end());
    return octets[0] + "." + octets[1] + "." + octets[2] + "." + octets[3];
  }

  string to_cidr() { return addr_to_string() + "/" + to_string(n); }

  bool operator==(const IP &other) const {
    return this->ip == other.ip && this->n == other.n;
  }
  bool operator!=(const IP &other) const { return !(*this == other); }
};
```

data

```
struct data {
  vector<uint8_t> raw_data;

  data(string s) : raw_data(s.begin(), s.end()) {}
};
```

Segment

```
struct Segment {
  // maximum segment size
  static size_t mss;

  int32_t seq_no;

  vector<uint8_t> data;

  // Constructors
  Segment(vector<uint8_t> data, int32_t seq_no)
      : data(move(data)), seq_no(seq_no) {}

  size_t get_data_size() { return data.size(); }

  string get_data_as_string() { return string(data.begin(), data.end()); }

  /*
   * Takes in application layer data
   * Splits it into multiple segments
   * The last segment will have FIN flag set, to indicate its the last one
   */
  static vector<Segment> make_segments(const application::data &data) {
    auto l = data.raw_data.begin();
    auto r = min(data.raw_data.begin() + Segment::mss, data.raw_data.end());
    vector<Segment> segments;
    do {
      segments.push_back(
          Segment(vector<uint8_t>(l, r), distance(data.raw_data.begin(), l)));
      l = r;
      r = min(r + Segment::mss, data.raw_data.end());
    } while (l != data.raw_data.end());
    return segments;
  }
  //
};
```

Packet

```
struct Packet {
  IP source;
  IP destination;
  int ttl;
  transport::Segment segment;

  Packet(IP src, IP dest, int ttl, transport::Segment segment)
      : source(src), destination(dest), ttl(ttl), segment(move(segment)) {}
};
```

Logging Functions

```
using network::Packet;

// Loggers for simulation
inline void RECEIVE(Router &src, Host &dest, Packet &p) {
  cout << "!!!! REACHED DESTINATION:  H(" << dest.ip.to_cidr()
       << ") received destined Pkt( seq_no: " << p.segment.seq_no
       << ", data: " << p.segment.get_data_as_string() << ")"
       << " from R(" << src.ip.to_cidr() << ")\n";
}
inline void LOG(Router &src, Router &dest, Packet &p) {
  cout << "R(" << dest.ip.to_cidr()
       << ") received Pkt( seq_no: " << p.segment.seq_no << ", ttl: " << p.ttl
       << ")"
       << " from R(" << src.ip.to_cidr() << ")\n";
}
inline void LOG(Host &src, Router &dest, Packet &p) {
  cout << "R(" << dest.ip.to_cidr()
       << ") received Pkt( seq_no: " << p.segment.seq_no << ", ttl: " << p.ttl
       << ")"
       << " from H(" << src.ip.to_cidr() << ")\n";
}

// DISCARD Packet, used in sncf to log discarding a packet when we've already
// seen the packet before
inline void DISCARD(Router &r1, Router &r2, Packet &p) {

  cout << "R(" << r1.ip.to_cidr()
       << ") discards Pkt( seq_no: " << p.segment.seq_no << ") coming from R("
       << r2.ip.to_cidr() << ")\n";
  //
}
```