15-441/641: Networking and the Internet
Project 1: Mixnet

TAs: Alexis Schlomer, Darshil Kaneria, Yifan Guang, Daiyaan Arfeen, Shawn Chen

Last Update: September 6, 2024
Checkpoint 1 due: 6:00 pm on September 13th, 2024
Checkpoint 2 due: 6:00 pm on September 25th, 2024
Lab due: 6:00 pm on September 27th, 2024

1 Introduction

To communicate secretly over the Internet, journalists, activists, and secret agents (seriously) use *privacy-preserving network overlays* to keep their communications hidden. Let's say we have a user, Alice, and an eavesdropper in the network, Eve,[1] who is watching all of Alice's transmissions. Alice can use a privacy-preserving overlay to prevent Eve from learning both *what* Alice is saying and *who* she is talking to. These overlays use three key technologies to protect users' privacy.

1. **Encryption**: Encryption guarantees that when an eavesdropper looks at any bytes going over the network, the eavesdropper cannot interpret what the message says. We will learn more about encryption later in this course and don't need to worry very much about how it works in detail.

2. **Redirection**: If Alice were to transmit her data directly to her receiver (let's call the receiver Frank), then Eve would observe that Alice is talking to Frank. Instead, Alice sends her data to someone else in the network (let's call them Bob), who then forwards the message to Frank. This is called redirection.

3. **Mixing**: But, what if Eve is watching Bob? Eve will see that Bob took in Alice's message, then sent it on to Frank. To prevent Eve from figuring this out, Bob can wait to retransmit Alice's message until after Bob has received messages from lots of other people – Charlie, David, etc. After a while, Bob will have messages for lots of users, and he can forward them out all at once. Eve will have no idea whether it was Alice's message that went to Frank, or Charlie's message, or David's.

In this project, we are going to build a simple *mix network* (*mixnet* for short). This mixnet will not be truly secure – the technologies we're applying are no longer state-of-the-art – but, the core ideas they're based on once were, and there are modern systems that leverage the same basic principles. To learn more about modern privacy-preserving routing you can read about the Tor project (www.torproject.org), or, from academia, the Vuvuzela project (vuvuzela.io).

In reality, our goals are the following: for you to hone your software engineering and development skills, become familiar with some practical routing algorithms, and practice evaluating the performance of a system that you have built. The evaluation is split into two parts: an *autograder* component, where we will test functional correctness of your code using a suite of automated test-cases (on Gradescope), and an *experimental* component, where you will hypothesize about and evaluate the performance of your system on a collection of Amazon EC2 servers.

Important note: Unlike P0, this assignment entails a considerable amount of design and programming effort, so *please* start early. Further, this assignment has two checkpoints (CPs) and a lab component; we highly recommend that you don't wait until the deadline for CP1 to start working on the other deliverables!

[1] You may have heard the fictional names 'Alice', 'Bob', etc. in the context of cryptography, but did you know that the characters also have roles? You can see the full list of roles (https://en.wikipedia.org/wiki/Alice_and_Bob#Cast_of_characters) to get a glimpse into the kinds of threats the security community cares about!

2 Mixnet

This section is organized as follows. §2.1 gives a high-level overview of the overlay network you are going to help implement. Next, §2.2 describes in detail the routing algorithms used in our mixnet; your primary task for this project is to implement the two routing policies described in that subsection, so please read it carefully. §2.3 describes the structure of packets in mixnet; in order to pass the autotester, all packets your code sends over the network must adhere precisely to the described packet structure. Finally, §2.4 briefly highlights the mixnet implementation (starter code).

2.1 Overview

Our mixnet is composed of one or more servers, called *nodes*, identified by unique, 16-bit *mixnet addresses*. Each mixnet node is connected to zero or more other nodes, resulting in a graph structure known as the *network topology*. For instance, Figures 1, 2, and 3 depict five mixnet nodes (labelled A through E) arranged in different network topologies; links between each pair of nodes are bidirectional (i.e., support traffic in both directions), so you should think of the topology as an undirected graph. As a matter of terminology, we will call two nodes *neighbors* if they share a link (e.g., nodes A and B are neighbors in all three topologies depicted here, while B and D do not have a neighbor relationship in any of them).

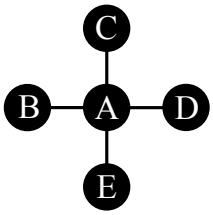

Figure 1: Hub-and-spoke

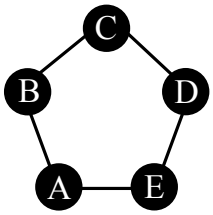

Figure 2: Ring

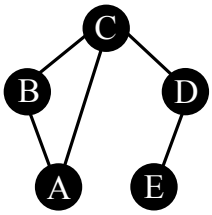

Figure 3: Partial mesh

Observe that our mixnet doesn't have any dedicated routers, only nodes. Thus, in addition to handling user packets originating at the node, each node in the mixnet also serves as a *forwarding node* for other packets sent over the network. For instance, in the partial mesh topology depicted in Figure 3, all packets destined for node E first arrive at D, which must then forward them to E. Our mixnet uses ***source routing*** (also known as *path addressing*), which means that the path to reach a given destination is explicitly specified by the source when it first injects a packet into the network.[2] Putting this all together, when a packet arrives at a given node, it performs one of three actions depending on node's role in the packet's lifecycle:

1. **Source**: If this node is the *source* node for this packet (i.e., the node received the packet directly from the user), it selects and encodes into the packet *the complete path to reach the destination node*. For example, if B wants to send a packet to E in the ring topology depicted in Figure 2, it first selects a path to E (say, $B \rightarrow C \rightarrow D \rightarrow E$), writes the path to the packet, and finally sends the packet to the next hop in the selected path (in this example, C). We will discuss path selection in more detail shortly.

2. **Destination**: If this node is the *destination* node for this packet, it simply relays the packet to the user. This marks the end of the packet's lifecycle.

3. **Forwarding**: If this node lies along the source-routed path but is *not* the destination node, the node reads the path encoded in the packet, finds the next-hop node in the path, then forwards it to the corresponding neighbor. In order to determine the next-hop node, the packet also contains a *hop index*; this is a 16-bit value that is initialized to 0 by the source node, and is incremented by 1 by every forwarding node along the path.

Path selection: The primary objective of most networks is to relay packets from source to destination as quickly and efficiently as possible. As you have seen in class, routers typically implement some form of *shortest-path routing* (i.e., make routing decisions that minimize the hop count or total delay) to achieve this

[2] In class, you also learned *hop-by-hop* routing, where next-hop decisions are made independently by each router. Can you think of cases where source routing may be preferred over hop-by-hop routing, and vice versa?

goal. We are going to have you support shortest path routing in your mixnet so that you can gain a deeper understanding of shortest-path algorithms.

Unfortunately, shortest path routing is not sufficient to preserve privacy. For instance, consider the hub-and-spoke topology depicted in Figure 1. If every node always uses shortest-path routing, an eavesdropper can tell precisely which pairs of nodes are communicating by observing A alone. Hence, we will also allow the nodes to use *random routing*; that is, in selecting a source route, these nodes will pick a random path to the destination (for the same hub-and-spoke topology, node A might randomly select a circuitous path to E that looks like $A \rightarrow B \rightarrow A \rightarrow E$, preventing the eavesdropper from knowing whether A is actually communicating with B or E).

Mixing: A source node is able to achieve limited privacy by choosing a non-direct path to the destination (i.e., *redirection*). However, as we saw earlier, this isn't sufficient to thwart an eavesdropper watching every node. Thus, our mixnet also needs to implement *mixing*. Each node in the mixnet is initialized with a *mixing factor* (an integer value between 1 and 16). The node's mixing factor determines the *exact number of packets* (received from the user layer, a neighbor, or both) that a node must 'collect' before sending them out over the network.

2.2 Routing

As described above, our mixnet implements both *shortest-path routing* (§2.2.1) and *random routing* (§2.2.2).

2.2.1 Shortest-Path Routing

In class, you learned about two classes of algorithms to perform shortest-path routing. First, there are *link-state algorithms*, where every node independently constructs a topological map of the network and then uses Dijkstra's algorithm to compute the shortest path to every destination. The second class comprises of *distance-vector algorithms*, where no one node stores information about the entire topology, but uses a distributed version of the Bellman-Ford algorithm to compute the next-hop node for every destination. Our mixnet will use a link-state algorithm inspired by *Open Shortest Path First* (OSPF).

As described in §2.1, our mixnet uses source addressing instead of hop-by-hop routing. This is possible because using a link-state algorithm gives every node a 'global view' of the network. Using this global view, a source node can compute and encode the full route to a destination in its outgoing packets, thus telling the network how it wants them routed.[3] Conceptually, routing in our mixnet involves the four high-level tasks described below.

Neighbor discovery: When the mixnet is first initialized, each node is only aware of the *number* of its neighbors, but not their mixnet addresses. The first step is for each node to 'learn' the addresses of its neighbors. In practice, this will be a small extension to the control path you will build to enable flooding (below).

Enabling safe flooding: Recall that link-state algorithms (e.g., OSPF) require every node to have a global view of the entire network topology. Initially, each node only knows about its own neighbors. In order to achieve a global view, every node informs every other node in the network about its neighbors via a process known as *flooding*. When a node receives a flood message from its neighbor, it sends a copy of the message to *each of its other neighbors*. However, we run into a problem here: if the mixnet topology has *loops* (i.e., cyclic components in a graph), then nodes along the loop will repeatedly flood their neighbors with messages, leading to the *broadcast storm* problem you have seen in class... yikes! An example of the broadcast storm problem is depicted in Figure 4.

Thus, to implement link-state routing in our mixnet, we need to first solve the broadcast storm problem. Fortunately, we already know an algorithm to do this: the *Spanning Tree Protocol* (STP)! Running STP on the original mixnet topology (which may contain loops) allows us to create a loop-free, *virtual* topology (i.e., a tree) which our link-state algorithm can use to safely flood neighbor information to all the other nodes in the network. We say that STP has converged when all nodes agree on the root node *and* have established a loop-free virtual topology. Note that the loop-free topology will *only* be used for flooding messages; source-routed messages may traverse links that STP would normally disable.

[3]Note that source routing is not possible in distance vector protocols because no node has a global view of the network.

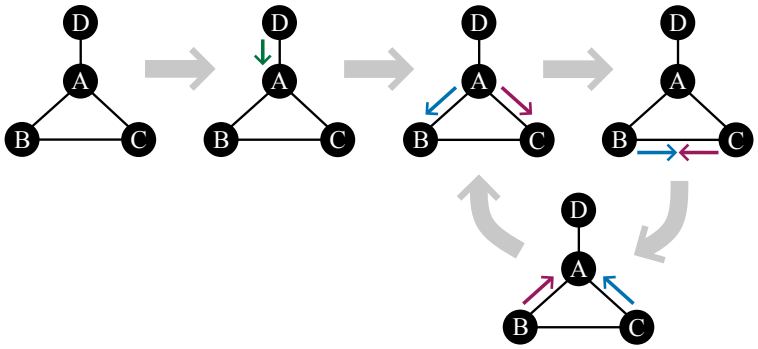

Figure 4: D sends A a message containing its neighbor information, which A then floods to B and C. Due to the loop in the topology, A receives copies of D's message from both B and C, which it faithfully floods to the other port, causing a broadcast storm.

We will use the same algorithm to implement STP as the one you learned in class, but with one small nuance. In order to detect and handle link failures (e.g., due to a malfunctioning cable), the *root node* of the spanning tree will periodically send a message (appropriately called a 'hello' message) to all its neighbors. Conversely, all nodes that are *not* the root will: a) listen for the root's message on their path to the root, and b) on receiving the message, immediately broadcast it on all *other* links. If a node doesn't receive the root's 'hello' message after a predetermined amount of time, it assumes that it's the new root, and spanning-tree establishment begins all over again. We will call the periodicity with which the root sends out 'hello' messages the *Root Hello Interval*, and the interval before which a node discards the old root information the *Reelection Interval*. You will see these parameters again when we discuss implementation in §2.4.

> 'Hello' messages are ordinary STP packets (§2.3.1) containing the root node's STP bid.

Computing shortest paths: Next, you will implement shortest-path computation so as to *minimize total routing cost on a weighted graph* (the link costs, or 'weights', will be given to you when the network is initialized). The process is very similar to OSPF: each node encodes its neighbor information in a message called a *Link State Advertisement* (LSA), which it then floods to every other node on the network *along the spanning-tree*. Each node also maintains a graph data-structure representing the network topology; when it receives a new LSA from a neighbor, the node inserts this information into the data-structure, and runs Dijkstra's algorithm to compute minimum-cost paths to *every other node in the network*. Finally, the node memoizes the shortest path to every node in a directory that can be indexed by the destination address (sometimes called a *forwarding information base*, or FIB). We say that the link-state algorithm has converged when all nodes have a global view of the mixnet topology, and have independently computed shortest-paths to every possible destination.[4]

> In the event that a source node has two or more identical-cost paths to a given destination, it should prioritize the path through the neighbor with the *smallest mixnet address*.

Routing along the shortest path: Once a node has computed shortest paths, source routing is simple: given a packet by the user, the node looks up the destination in its shortest-path directory (FIB), encodes the route in the packet, then forwards the packet to the next hop along the path. Finito.

2.2.2 Random Routing

For better privacy, nodes in our mixnet topology may be configured to use *random* routing. Simple as it sounds, generating truly random paths is a non-trivial problem! For this project, we will use a highly relaxed

[4]It is important to note that the shortest path may include links that are *not* part of the spanning tree – otherwise, there would be only one route to every destination! The spanning tree is exclusively used to forward *flood* messages (e.g., LSAs), while data messages can be routed along any link.

definition of 'randomness'. We will say that a mixnet node correctly implements 'random' routing if, given a sufficiently large number of packets to route, the node generates *at least two* distinct routes. Subject to this constraint, your implementation of random routing can be *as simple* or *as complex* as you wish. The most creative (and correct!) implementation will earn a shout-out on Ed and candy of your choice!

2.3 Packets

Like any packet-switched network, packets sent over our mixnet must adhere to certain specifications (or risk being dropped as malformed packets!). All mixnet packets have a common 12-byte (96-bit) header, whose format is depicted in Figure 5.[5] The header fields are as follows:

- *Total Size*: Size in bytes of the entire packet (*including* the header). The maximum total size for mixnet packets is given by the macro `MAX_MIXNET_PACKET_SIZE`, defined in mixnet/packet.h.
- *Packet Type*: Integer between 0 and 4, corresponding to one of the 5 supported packet types (§2.3.1).
- *Reserved*: This field is reserved for exclusive use by the framework code. Any data written here will be bleached (written over), so you must not attempt to use this header field.

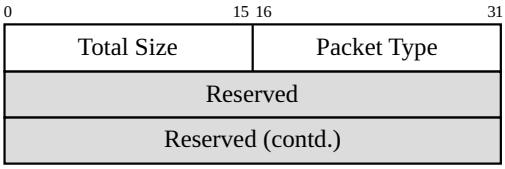

Figure 5: Mixnet header format.

The (*Total Size* − 12) bytes appearing immediately after the mixnet header constitute the variable-length packet payload, and the contents of the payload itself depend on the *Packet Type*.

2.3.1 Packet Types

Our mixnet supports 5 types of packets:

1. `PACKET_TYPE_STP`: STP packets will be used to facilitate both neighbor discovery as well as construction of mixnet's spanning tree (see **neighbor discovery** and **enabling safe flooding** in §2.2.1). The packet payload for STP packets contains three attributes: *Root Address*, *Path Length*, and *Node Address*. *Root Address* is the mixnet address of the spanning tree's root node, *Path Length* is the number of hops required to reach the root, and *Node Address* is the mixnet address of the node sending this packet. The payload format is depicted in Figure 6.

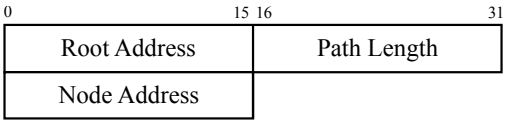

Figure 6: Payload format for STP packets.

   For STP packets, *Packet Type* must be set to `PACKET_TYPE_STP`, and *Total Size* set to $12 + 6 = 18$.

2. `PACKET_TYPE_FLOOD`: FLOOD packets will be used to debug and test your STP implementation. There is only one requirement for FLOOD packets: they *must* be flooded on all links corresponding to the mixnet's spanning-tree. Nodes that receive FLOOD packets on non-user ports should also forward them out over the user port (see §2.4 for details about ports). *Packet Type* must be set to `PACKET_TYPE_FLOOD`, and *Total Size* must be set to 12 (empty payload).

[5] You will often see these types of figures in protocol standards (e.g., page 14 of https://www.ietf.org/rfc/rfc793.txt, the RFC for TCP published back in 1981). The figure is read from left to right, and top to bottom. Each row corresponds to 32 bits (4 bytes) of packet data, ranging from bits 0 through 31 in the first row, bits 32 through 63 in the second row, and so on.

3. `PACKET_TYPE_LSA`: LSA (or *Link State Advertisement*) packets encode a node's neighbor information that it sends to every other node in order to construct a topological map of the network (see **computing shortest paths** in §2.2.1). LSA packets must also be flooded through the network along the spanning tree. The packet payload for LSA packets contains three attributes: *Node Address*, *Neighbor Count*, and *Neighbor List*. *Node Address* is the mixnet address of the advertising node, and *Neighbor Count* is the number of nodes neighboring the advertising node. Finally, *Neighbor List* is a list of tuples of the form (*Neighbor Address*, *Link Cost*), where *Neighbor Address* corresponds to a neighbor's mixnet address, and *Link Cost* is a 16-bit value representing the cost of routing on that link.

   For instance, in the hub-and-spoke topology depicted in Figure 1, assuming that the routing cost to all its neighbors was 2, node A's LSA would be as follows: (*Node Address* = A, *Neighbor Count* = 4, *Neighbor List* = [(B, 2), (C, 2), (D, 2), (E, 2)]). The payload format is depicted in Figure 7.

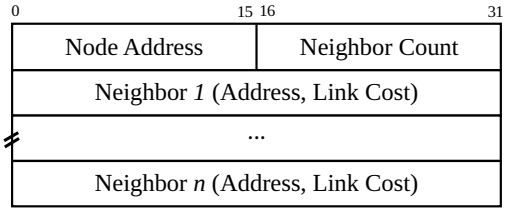

Figure 7: Payload format for LSA packets.

   For LSA packets, *Packet Type* must be set to `PACKET_TYPE_LSA`, and *Total Size* must be set to $12 + (4 + (4 \cdot n))$, where n is the advertising node's neighbor count.

4. `PACKET_TYPE_DATA`: DATA packets will be used to test your routing implementation. These packets are source-routed, so the sender must encode the *complete path to the destination* in the packet itself. The packet payload for DATA packets contains six attributes: *Source Address*, *Destination Address*, *Route Length*, *Hop Index*, *Route*, and *Data*. *Source Address* is the mixnet address of the node that first injects the packet into the network, and *Destination Address* is specified by the user (i.e., given to you). *Route Length* is the number of hops in the path *excluding both the source and destination*. *Hop Index* is a zero-indexed value identifying the current hop in the route. *Route* is a list of mixnet addresses corresponding to hops in the route (once again excluding both the source and destination). Finally, *Data* is the information that the source wants to convey. The payload format is depicted in Figure 8. Note that the first five attributes (Source Address, Destination Address, Route Length, Hop Index, and Route) together constitute the *Routing Header*, a structure we will reuse shortly.

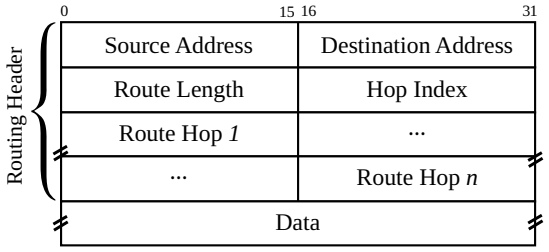

Figure 8: Payload format for DATA packets.

   For DATA packets, *Total Size* must be set to $12 + (8 + (2 \cdot n)) + \text{size(DATA)}$, where n is the number of hops in the route excluding the source and destination.

5. `PACKET_TYPE_PING`: You will use PING packets to evaluate RTTs in the mixnet. Like DATA packets, PING packets are source-routed, so the sender must encode the path in the packet itself. For the purpose of routing, we will reuse the routing header structure we first saw in Figure 8. However, in place of the *Data* field, PING packets will have two different attributes: *Is Request?* (IR) and *Request Time*. *IR* encodes whether the message is a *request* (value `true`) or a response (value `false`). *Request Time* encodes the *local time at which the ping was initially sent* at the sender side as a 64-bit integer. The payload format for PING packets is depicted in Figure 9.

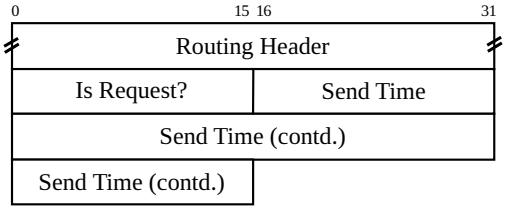

Figure 9: Payload format for PING packets.

For PING packets, the *Source Address* is the mixnet address of the node that first injects the packet into the network, and the *Destination Address* is specified by the node's user (i.e., given to you). *Total Size* must be set to $12 + (8 + (2 \cdot n)) + 10$, where n is the number of hops in the route excluding the source and destination.

Similar to the `ping` tool you saw in Project 0, PING works as follows in our mixnet: a node in the network generates a request by creating a PING packet with *Is Request?* set to `true`, and *Send Time* set to its local time (encoded as a 64-bit integer). The request is then source-routed to the required destination. At this point, the destination node swaps the *Source Address* and *Destination Address*, reverses the route in the Routing Header, sets *Is Request?* to `false` (indicating response),[6] then forwards the packet back out over the network. On receiving this response, the sender computes RTT as the difference between the *current time* and the *Send Time* encoded in the packet.

2.4 Implementation

Our mixnet is written in the C programming language; your goal is to implement the control- and data-plane functionality for the nodes on this network. By the end of this project, you will have realized a mixnet capable of: (1) self-establishing a spanning-tree that is robust to sporadic link failures, (2) performing source routing using two different routing algorithms (shortest-path and random), and (3) preserving privacy (i.e., identities of communicating node pairs) by applying *packet redirection* and *mixing*. Let's get started!

The entry-point to your node implementation is the `run_node(...)` function in mixnet/node.c, the function signature for which is shown below (declared in mixnet/node.h).

```
// Node implementation
void run_node(void *handle, volatile bool *keep_running, const struct mixnet_node_config config);
```

The `mixnet_node_config` struct (shown below, defined in mixnet/config.h) contains all the relevant configuration parameters for each mixnet node. By this point, you should be familiar with what all of these parameters mean; if not, please read through this section again! For reference, the STP parameters are described in the paragraphs on **enabling safe flooding** in §2.2.1, and the routing parameters are described in the paragraphs on **path selection** and **mixing** in §2.1.

```
// Node configuration
struct mixnet_node_config {
    mixnet_address node_addr;       // Mixnet address of this node
    uint16_t num_neighbors;         // This node's total neighbor count

    // STP parameters
    uint32_t root_hello_interval_ms;  // Time (in ms) between 'hello' messages
    uint32_t reelection_interval_ms;  // Time (in ms) before starting reelection

    // Routing parameters
    bool do_random_routing;         // Whether this node should perform random routing
    uint16_t mixing_factor;         // Exact number of (non-control) packets to mix
    uint16_t *link_costs;           // Per-neighbor routing costs, in range [0, 2^16)
};
```

[6] This is important! Otherwise, the PING packet will bounce back and forth between the source and destination forever, and your network will take a real... sto(r)mping.

Observe that `run_node()` takes two additional arguments: an opaque pointer (`void *handle`), and a pointer to a boolean (`bool *keep_running`). You will use `handle` to communicate with mixnet's network API (described below), but you must NOT modify it in any way – or your own program will crash! The second parameter, `keep_running`, is used to signal to your program when it should exit gracefully. Initially, (`*keep_running`) will yield true, but you should periodically check its contents, returning from the function when it becomes false (but not before!). A small portion of your autotester grade will be assigned based on whether or not your node implementation terminates gracefully.

2.4.1 Sending and Receiving Packets

Our mixnet provides two functions to communicate with other nodes on the network: `mixnet_send(...)` and `mixnet_recv(...)`, the function signatures for which are shown below (declared in mixnet/connection.h). The header contains more detailed documentation, but we provide a brief overview of their usage here.

```
// Send a packet over the Mixnet network
int mixnet_send(void *handle, const uint8_t port, mixnet_packet *packet);

// Receive a packet sent to this node over the Mixnet network
int mixnet_recv(void *handle, uint8_t *port, mixnet_packet **packet);
```

Both `mixnet_send()` and `mixnet_recv()` are non-blocking (i.e., return almost immediately), and return the number of packets sent/received (either 0 or 1). `mixnet_send()` returns -1 on error (malformed packets).

Sending packets: `mixnet_send()` allows a node to send a packet to one of its neighbors. Given a pointer to a properly-constructed mixnet packet (`packet`) and a zero-indexed neighbor ID (`port`), this function sends the packet to the corresponding neighbor. As an example, consider the hub-and-spoke topology in Figure 1. Node A has 4 neighbors, and assume that you find (via neighbor discovery) that they are ordered by port ID as follows: [C, B, D, E]. We say that `C` is A's 0th neighbor, and to send a packet `p` to node C, A invokes `mixnet_send(handle, 0, p)`. Note that `port` is not the mixnet address of the neighbor, but rather the *index* at which it appears in the node's neighbor list.

Receiving packets: Receiving packets is a similar process. `mixnet_recv()` takes as arguments a pointer to a port (`uint8_t *port`) and a double-pointer to a mixnet packet (`mixnet_packet **packet`). It then populates `*packet` with a *pointer to heap-allocated mixnet packet*, and `*port` with the *index* on which the packet was received.

2.4.2 User Port

Finally, if a node has n neighbors, mixnet designates the last (n'th) port as an *input/output* port on which packets are received from and sent to the user. Once again, consider the hub-and-spoke topology example described above. When node A invokes `mixnet_recv()`, it may receive packets on ports 0, 1, 2, or 3 (corresponding to its neighbors, C, B, D, or E, respectively), **or** it might receive a packet on port 4, which corresponds to user-provided packets it must route to the appropriate destinations.

You may assume that packets you receive from the user will only be either FLOOD, DATA, or PING packets. Since the route is not known to the user when they send you a packet, DATA and PING packets you receive on the input port will have a *Route Length* of 0. You are responsible for computing the route and initializing the routing header (for reference, please see descriptions of `PACKET_TYPE_DATA` and `PACKET_TYPE_PING` in §2.3.1). For DATA packets, you are also responsible for copying the packet data to its correct position in the payload.[7]

When you receive a FLOOD, DATA, or PING packet that is destined for this node (either has this node's *Destination Address*, or is a FLOOD packet that would not cause a broadcast storm), you must propagate it back to the user using `mixnet_send()`. The docstrings in mixnet/connection.h provide a more detailed explanation of these functions.

[7] If you're modifying packets in-place, make sure you don't accidentally clobber the data while initializing the routing header!

3 Evaluation

This project is broken into two checkpoints and a lab to help you maintain pace with the required work. Your grade for each checkpoint will be determined by a suite of automated test-cases on Gradescope (hereafter referred to as the *autograder*); in the lab component, you will hypothesize about, experiment with, and answer questions relating to your mixnet implementation.

3.1 Checkpoints

For each checkpoint, you will submit an archive (.zip) containing *only* the source (.c) and (.h) files required to build and run your code. The autograder assumes that all required files (including headers) reside in the mixnet directory without additional nesting. For instance, if your implementation spans two source files (*source1.c* and *source2.c*) which depend on two headers (*header1.h* and *header2.h*), you can create the zip as follows:

```
zip -j submission.zip source1.c source2.c header1.h header2.h
```

> The autograder uses the console output to grade your implementation. Please remove all `printf` statements from your Gradescope submission to avoid interfering with the autograder.

Checkpoint 1: Spanning Tree Protocol

The first checkpoint evaluates your implementation of the Spanning Tree Protocol (STP). You must implement everything described in the paragraphs on **neighbor discovery** and **enabling safe flooding** in §2.2.1, and handle mixnet packets of type `PACKET_TYPE_STP` and `PACKET_TYPE_FLOOD` (please see the relevant descriptions in §2.3.1). In addition, your node should handle occasional link failures (using the *Root Hello Interval* and *Reelection Interval* timers to reestablish the spanning-tree when links go down), but you may assume that you will not receive FLOOD packets from the user until your spanning-tree has converged.[8]

Checkpoint 2: Source Routing

The second checkpoint evaluates your full mixnet implementation. You must implement everything described in §2.1 and §2.2, and handle packets of the three remaining types (`PACKET_TYPE_LSA`, `PACKET_TYPE_DATA`, and `PACKET_TYPE_PING`). For this checkpoint, you may assume that *links will never fail* (i.e., the underlying network is fully reliable), and that you will not receive DATA and PING packets from the user until your link-state routing algorithm has converged.

3.2 Lab

For the experimental component, evaluate the following properties of your mixnet implementation.

Evaluation 1: STP Convergence Rate

Hypothesis: Rank, in increasing order of the number of control messages exchanged, the following topologies: line, binary tree, ring, and fully-connected. Why do you expect this particular relative ordering?

Experiment: Set up a mixnet cluster with each of the above topologies (line, binary tree, ring, and fully-connected) on **8 EC2 servers in the same region**, and measure the convergence time and number of control messages exchanged between nodes in the network. Create two bar plots: one with the topology on the X-axis and *STP convergence time* on the Y-axis, and another with the topology on the X-axis and the *number of STP messages exchanged* on the Y-axis.

Inference: Does the data you measure corroborate your hypothesis? Why or why not?

Evaluation 2: Dependence of RTT on Topology

[8] Once STP convergence begins, we will wait a reasonable amount of time (at least 2X the time required by the reference solution's STP implementation to converge) before sending you FLOOD packets.

Hypothesis: For our mixnet, we'll define the *worst-case RTT* as the RTT between the two furthest nodes in the network (i.e., the nodes with the maximum number of hops on the shortest path between them). Rank, in increasing order of worst-case RTT, the following topologies: line, binary tree, ring, and fully-connected. Why do you expect this particular relative ordering?

Experiment: Set up a mixnet cluster with each of the above topologies (line, binary tree, ring, and fully-connected) on **8 EC2 servers in the same region**, and measure the worst-case RTT.[9] Create a bar plot with the topology on the X-axis and the *worst-case RTT* on the Y-axis.

Inference: Does the data you measure corroborate your hypothesis? Why or why not?

4 Logistics

Framework code: We reserve the right to change the framework code as the project progresses to fix bugs and to introduce new features that will help you debug your code (we commit to making the changes as transparent as possible so as to not break your working implementation :)). You are responsible for checking Ed to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to Ed. ***Important***: we are releasing the harness code in good faith (instead of, e.g., just giving you a static library) so you can starting investigating issues on your own without waiting on the TAs for help; we will, however, be manually inspecting your code, and any attempts at subverting the autograder will result in, at minimum, a score of zero for the entire project. Please don't do this!

Reusing code: You are permitted to use any code *you* have previously written (e.g., from 15-213/513, a hackathon, etc.), but you may not use code from anyone or anywhere else. You are welcome to clone the project repository to stay up-to-date with changes (e.g., bug-fixes), but please do not push your solution code to a public fork. We take academic integrity violations *very* seriously, and you will be held just as liable for sharing code with someone as you will be for using it.

> In keeping with the course policy on Generative AI, you are *not* permitted to use code assistants (*including* GitHub Copilot) to generate boilerplate code. *Any* code that triggers our plagiarism software will be manually inspected by the course staff; for any code that does not pass these secondary checks, the onus will be on you to demonstrate compliance with the academic integrity policy.

Testing: The starter code includes a suite of public test-cases to help you get started, but the autograder may use private test-cases to grade your solution (you will see the final score at the time of submission, though). Please come up with your own tests; it'll save both you and your partner a lot of debugging pain in the long term!

Good luck, and may the *C* Gods be ever in your favor!

[9] You will use the PING functionality you implemented to perform this experiment.

5 Summary

There are several moving parts to this project! Figure 10 is a flowchart to help you recap what the individual parts are, how they interact, and what you are responsible for in each checkpoint. The flowchart is drawn from the perspective of a mixnet node that has just received a packet.

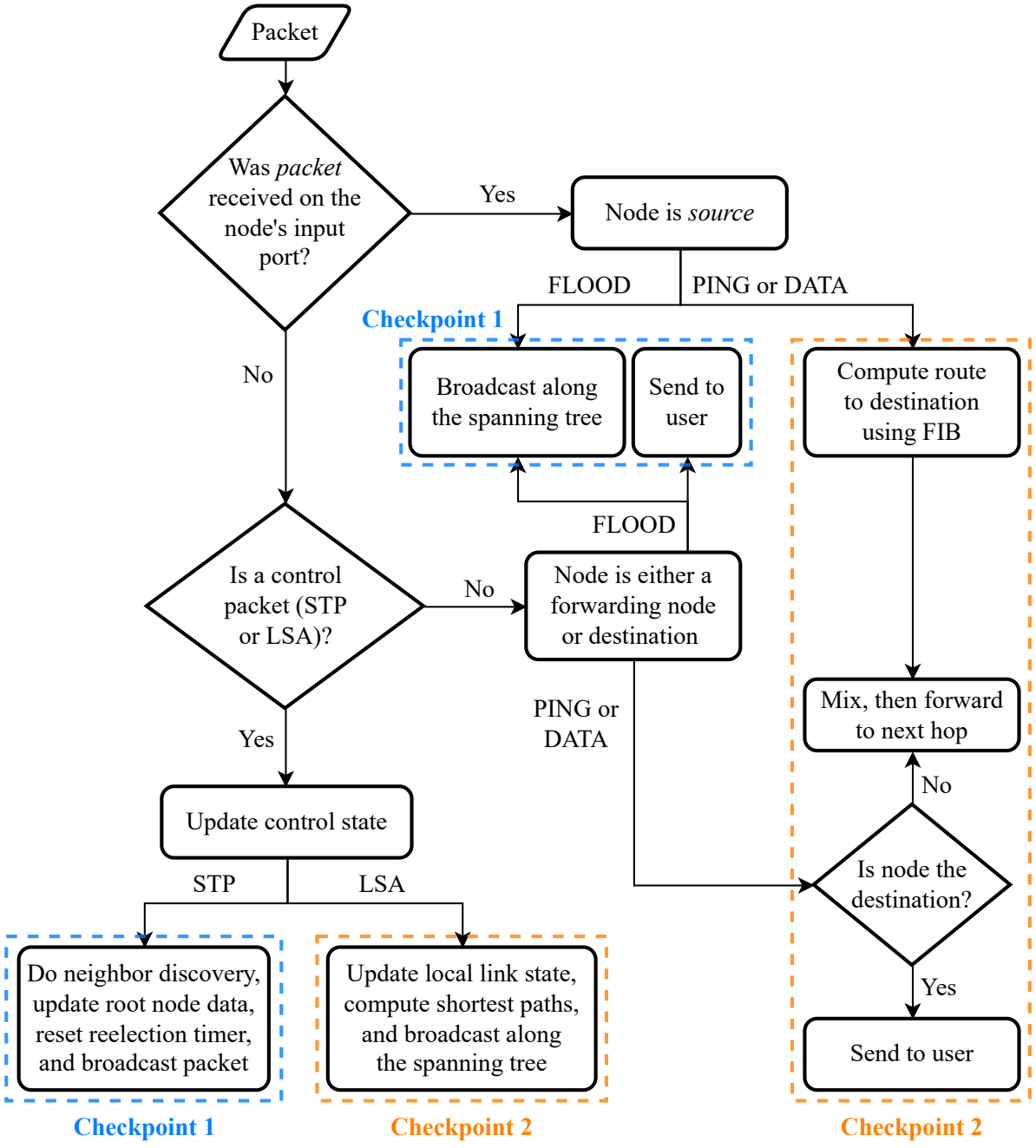

Figure 10

A Resources

You may find the following resources to be helpful:

- Data structures: linked lists [1], hash tables [2], graphs and adjacency lists [3]
- Algorithms: STP [4], Dijkstra's algorithm [5]
- Debugging: `gdb` [6], `valgrind` [7], Google's AddressSanitizer [8]

References

[1] https://en.wikipedia.org/wiki/Linked_list

[2] https://en.wikipedia.org/wiki/Hash_table

[3] https://en.wikipedia.org/wiki/Adjacency_list

[4] https://people.cs.umass.edu/~phillipa/CSE390/p44-perlman.pdf

[5] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[6] https://linux.die.net/man/1/gdb

[7] https://linux.die.net/man/1/valgrind

[8] https://github.com/google/sanitizers/wiki/AddressSanitizer