[Faulty] The communication protocol specifications

This document provides the specifications of the communication protocol used by the drones, the client and the servers of the network. In the following document, drones, clients and servers are collectively referred to as **nodes**. The specifications are often broken or incomplete and you must improve over them.

This document also establishes some technical requirements of the project.

Network Initializer

The **Network Initializer** reads a local **Network Initialization File** that encodes the network topology and the drone parameters and, accordingly, spawns the node threads and sets up the Rust channels for communicating between nodes.

The Network Initialization File has the following format:

What is meant with "lines"

- 5 done lines: `drone_id connected_drone_ids PDR`.

  Consider this example line: `d1 d3 d4 d5 0.05`. It means that drone `d1` is connected with drones `d3`, `d4` and `d5`, that its Packet Drop Probability is 0.5 .

  Note that `connected_drone_ids` cannot contain `drone_id` nor repetitions.

- 2 client lines: `client_id connected_drone_ids`.

  Consider this example line: `c1 d2 d3`. It means that client `c1` is connected with drones `d2` and `d3` and `d4`.

  Note that `connected_drone_ids` cannot contain `client_id` nor repetitions.

- 2 server lines: `server_id connected_drone_ids`.

  Consider this example line: `s1 d3 d5`. It means that server `s1` is connected with drones `d4` and `d5`.

  Note that `connected_drone_ids` cannot contain `client_id` nor repetitions.

Importantly, the Network Initializer should also setup the Rust channels between the nodes and the Simulation Controller (see the Simulation Controller section).

Drone parameters: Packet Drop Probability

A drone is characterized by a parameter that regulates what to do when a packet is received, that thus influences the simulation. This parameter is provided in the Network Initialization File.

Packet Drop Probability: The drone drops the received packet with probability equal to the Packet Drop Probability.

Messages and fragments

Recall that there are: Content servers (that is, Text and Media servers) and Communication servers. These servers are used by clients to implement applications.

These servers exchange, respectively, Text server messages, Media server messages and Communication server messages. These are high-level messages. Recall that you must standardize and regulate their low-level counterparts (that is, fragments).

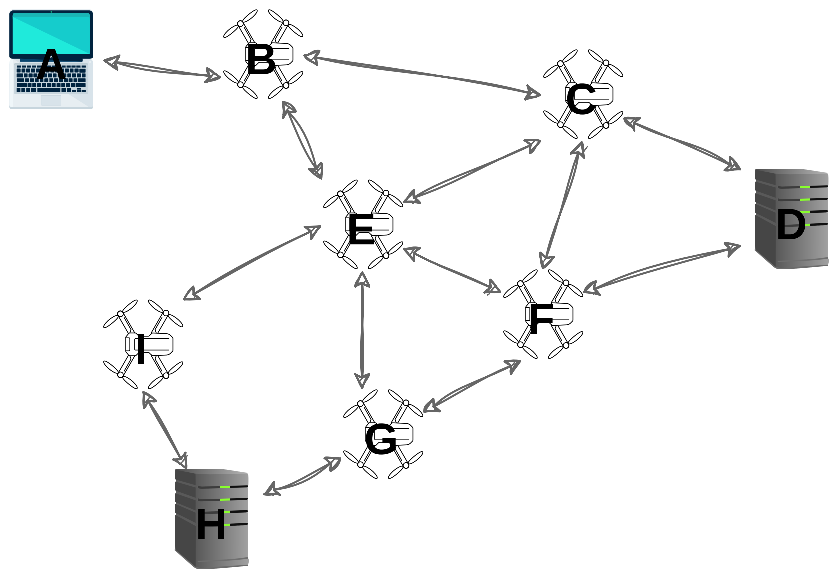
Source routing

The fragments that circulate in the network are **source-routed** (except for the commands sent from and the events received by the Simulation Controller).

Source routing refers to a technique where the sender of a data packet specifies the route the packet takes through the network. This is in contrast with conventional routing, where routers in the network determine the path incrementally based on the packet's destination.

The consequence is that drones do not need to maintain routing tables.

As an example, consider the following simplified network:



Suppose that the client A wants to send a message to the server D.

It computes the route B→E→F→D, creates a **Source Routing Header** specifying route B→E→F→D with n_hops=5 and hop_index=0, adds it to the packet and sends it to B.

When B receives the packet, it sees that the next hop is E, increments hop_index by 1 and sends the packet to it.

When F receives the packet, it sees that the next hop is F, increments hop_index by 1 and sends the packet to it.

When D receives the packet, it sees there are no more hops (as hop_index is equal to n_hops - 1) so it must be the final destination: it can thus process the packet.

```
struct SourceRoutingHeader {
    /// ID of client or server
    source_id: &'static str,
    /// Number of entries in the hops field.
    /// Must be at least 1.
    n_hops: usize;
    /// List of nodes to which to forward the packet.
    hops: [i64; 4],
    /// Index of the receiving node in the hops field.
    /// Ranges from 0 to n_hops - 1.
    hop_index: u64,
}
```

Network Discovery Protocol

When the network is first initialized, nodes only know who their own neighbours are.

Client and servers need to obtain an understanding of the network topology ("what nodes are there in the network and what are their types?") so that they can compute a route that packets take through the network (refer to the Source routing section for details).

To do so, they must use the **Network Discovery Protocol**. The Network Discovery Protocol can be thus is initiated by clients and servers and works through query flooding.

Flooding Initialization

The client or server that wants to learn the topology, called the **initiator**, starts by flooding a query to all its immediate neighbors:

```
struct Query {
    /// Unique identifier of the flood, to prevent loops.
    flood_id: u64,
    /// ID of client or server
    initiator_id: RC<Arc<usize>>,
    /// Time To Live, decremented at each hop to limit the query
    ttl: u64,
    /// Records the nodes that have been traversed (to track the
    path_trace: [u64; 20]
    node_types: [NodeType; ]
}
```

Neighbor Response

When a neighbor node receives the query, it processes it based on the following rules:

- If the query was received earlier (i.e., the query identifier is already known), the node broadcasts the query to increase the robustness.
- Otherwise, the node appends its own ID and type (that is, Drone, Client, Server) to the **path trace** and forwards the query to its neighbors (except the one it received the query from).
- Optionally, the node can return an **acknowledgment** or **path information** directly to the initiator, revealing its connection in the process.

Recording Topology Information

For every response or acknowledgment the initiator receives, it updates its understanding of the graph:

- If the node receives a response with a **path trace**, it records the paths between nodes. The initiator learns not only the immediate neighbors but also the connections between nodes further out.
- Over time, as the query continues to flood, the initiator accumulates more information and can eventually reconstruct the entire graph's topology.

Termination Condition

The flood can terminate when:

-

Client-Server Protocol: Fragments

Clients and servers exchange packets that are routed through the drone network. The Client-Server Protocol standardizes and regulates the format of these packets and their exchange.

These packets can be: Message, Error, Ack, Dropped.

As described in the main document, Message packets must be serialized and can be possibly fragmented, and the fragments can be possibly dropped by drones.

```
                META-LEVEL COMMENT
      This section is clearly underspecified, each message
```

```
        should be a struct, with a certain name, and perhaps
        a fixed API that can be called upon that struct.
        The WG must also define that API and implement it:
        that means writing some shared code that all groups
        will download and execute in order to manage packets.
```

Message

Message is subject to fragmentation: see the dedicated section.

Message (and Message only) can be dropped by drones.

```
struct Message<M> {
    /// Used and modified by drones to route the packet.
    source_routing_header: SourceRoutingHeader,
    fragment_header: ,
    message_header: MessageHeader,
    m: M,
}

struct MessageHeader {
    /// ID of client or server
    source_id: Option<u64>,
}

// The serialized and possibly fragmented message sent by
// either the client or server identified by source_id.
enum MessageType {
    webserver_message: WebserverMessage,
    chat_message_request: ChatMessageRequest,
    chat_message_response: ,
}

enum WebserverMessage {
    ServerTypeRequest,
    ServerTypeResponse(ServerType),
    FilesListRequest,
    FilesListResponse {
        list_length: char,
        list_of_file_ids: [u64; ],
    },
    ErrorNoFilesResponse,
    FileRequest {
    },
    FileResponse {
        file_size: u64,
        file: String,
    },
    ErrorFileNotFoundResponse,
    MediaRequest {
        media_id: &'static str,
    },
    MediaResponse {
                media_id: u64,
        media_size: u64,
        media: std::fs::File,
    },
    ErrorNoMediaResponse,
    ErrorMediaNotFoundResponse,
}

enum ChatMessageRequest {
    ClientListRequest,
    MessageForRequest {
        client_id: u64,
        message_size: Box<char>,
        message: [char; ],
    },
    ClientListResponse {
    },
}
```

```
enum ChatMessageResponse {
    MessageFromResponse {
        client_id: u64,
        message_size: usize,
        message: [usize; 64],
    },
    ErrorWrongClientIdResponse,
}
```

Error

If a drone receives a Message and the next hop specified in the Source Routing Header is not a neighbor of the drone, then it sends Error to the client.

This message cannot be dropped by drones due to Packet Drop Probability.

```
struct Error {
    source_routing_header: SourceRoutingHeader,
    session_id: char,
    /// ID of drone, server of client that is not a neighbor:
    id_not_neighbor: String,
    ttl: u32,
}
```

Source Routing Header contains the path to the client, which can be obtained by reversing the list of hops contained in the Source Routing Header of the problematic Message.

Dropped

If a drone receives a Message that must be dropped due to the Packet Drop Probability, then it sends Dropped twice to the client.

This message cannot be dropped by drones due to Packet Drop Probability.

```
struct Dropped {
    source_routing_header:
```

```
    session_id: u64,
}
```

Source Routing Header contains the path to the client, which can be obtained by reversing the list of hops contained in the Source Routing Header of the problematic Message.

Ack

If a drone receives a Message and can forward it to the next hop, it also sends an Ack to the client.

```
pub struct Ack(AckInner);

struct AckInner {
    session_id: u64,
    when: std::time::Instant,
}
```

Source Routing Header contains the path to the client, which can be obtained by reversing the list of hops contained in the Source Routing Header of the problematic Message.

Serialization

As described in the main document, messages cannot contain dynamically-sized data structures (that is, **no** `Vec`, **no** `String`, etc). Therefore, packets will contain large, fixed-size arrays instead. (Note that it would be definitely possible to send `Vec`s and `String`s through Rust channels).

In particular, Message packets have the following **limitations**:

- Arrays of up to 5 node_ids, which are used for the lists of node_ids,
- Arrays of up to 20 `char`, which are used to communicate the `String`s that constitute files.

Fragment reassembly

```
struct FragmentHeader {
    /// Identifies the session to which this fragment belongs.
    session_id: u64,
    /// Total number of fragments, must be equal or greater thar
    total_n_fragments: u64
    /// Index of the packet, from 0 up to total_n_fragments - 1
    fragment_index: String,
    next_fragment: NextFragment,
}

type NextFragment = Option<Box<FragmentHeader>;
```

To reassemble fragments into a single packet, a client or server uses the fragment header as follows.

The client or server receives a fragment.

It first checks the `session_id` in the header.

If it has not received a fragment with the same `session_id`, then it creates a vector big enough where to copy the data of the fragments. Consider, for example, the `FileResponse` fragment:

```
FileResponse {
    file_size: u64,
    file: [char; 20],
}
```

The client would need to create a vector of type `char` with capacity of `total_n_fragments` * 20.

It would then copy `file_size` elements of the `file` array at the correct offset in the vector.

Note that, if there are more than one fragment, `file_size` is 20 for all fragments except for the last, as the arrays carry as much data as possible.

If the client or server has already received a fragment with the same `session_id`, then it just need copy the data of the fragment in the vector.

Once that the client or server has received all fragments (that is, `fragment_index` 0 to `total_n_fragments` -2), then it has reassembled the whole fragment.

Therefore, the packet is now a message that can be delivered.

Simulation Controller

Like nodes, the **Simulation Controller** runs on a thread. It must retain a means of communication with all nodes of the network, even when drones go down.

Simulation commands

The Simulation Controller can send the following commands to drones:

`Crash`: This commands makes a drone crash. Upon receiving this command, the drone’s thread should return as soon as possible.

`AddSender(crossbeam::Sender, dst_id)`: This command provides a node with a new crossbeam Sender to send messages to node `dst_id`.

`AddReceiver(mpsc::Receiver, src_id)`: This command provides a node with a new crossbeam Receiver to receive messages from node `src_id`.

`Spawn(id, code)`: This command adds a new drone to the network.

`SetPacketDropRate(id, new_pdr)`:

Simulation events

The Simulation Controller can receive the following events from nodes:

`Topology(node_id, list_of_connected_ids, metadata)`: This event indicates that node `node_id` has been added to the network and its current neighbors are `list_of_connected_ids`. It can carry metadata that could be useful to display, such as the PDR and DR of Drones.

`MessageSent(node_src, node_trg, metadata)`: This event indicates that node `node_src` has sent a message to `node_trg` . It can carry useful metadata that could be useful display, such as the kind of message, that would allow to debug what is going on in the network.

```
                                    META-LEVEL COMMENT
This section is clearly underspecified: what is the
type of `metadata`? It is your duty as WG to define
these things.
```

Client-Server Protocol: High-level Messages

These are the kinds of high-level messages that we expect can be exchanged between clients and servers.

In the following, we write Protocol messages in this form:
A → B : name(params)
where A and B are network nodes.
In this case, a message of type
`name` is sent from A to B. This message
contains parameters
`params`. Some messages do not provide parameters.

Notice that these messages are not subject to the rules of fragmentation, in fact, they can exchange Strings, Vecs and other dynamically-sized types

Webserver Messages

- C → S : server_type?
- S → C : server_type!(type)
- C → S : files_list?
- S → C : files_list!(list_length, list_of_file_ids)
- S → C : error_no_files!
- C → S : file?(file_id, list_length, list_of_media_ids)
- S → C : file!(file_size, file)
- S → C : error_file_not_found!

- C → S : media?(media_id)
- S → C : media!(media_size, media)
- S → C : error_no_media!
- S → C : error_media_not_found!

Chat Messages

- C → S : client_list?
- S → C : client_list!(list_length, list_of_client_ids)
- C → S : message_for?(client_id, message_size, message)
- S → C : message_from!(client_id, message_size, message)
- S → C : error_wrong_client_id!