Advanced Programming 2024/2025 Course Project

This document describes the course project of the 2024/2025 A.Y.

At a high level, the project consists in developing drones, which are deployed in networks, and client-server applications that use the drone network to communicate. In the following document, drones, clients and servers are collectively referred to as **nodes**.

In real-world, drones are used to deploy communication infrastructure in places where conventional technology does not offer good connectivity. Examples include environmental and natural disaster monitoring, border surveillance, emergency assistance, search and rescue missions, relay communications, military applications for surveillance and reconnaissance; on-demand network for real-time communication of city-related information such as car and pedestrian.

Always adhere to these **coding principles**:

- No unsafe code.
- No undocumented panics.
- The code must be extensively tested.
- Do not preclude crates downstream to test their code.
- Expose through public interfaces only what is strictly needed.
- Write idiomatic code.

Configure your IDE or editor to use rustfmt so that the code of from all team members is consistently formatted.

Furthermore, try to minimize the warnings produced by clippy (you can also configure your IDE or editor to show clippy's warnings): `cargo clippy -- -W clippy::pedantic`

Short description

There is a network of drones, clients and servers. The clients and servers are at the edge of the network.

The servers provide differents functionalities. The clients use the server functionalities to implement applications. The drones route packets from clients to servers and vice versa.

The communication between the nodes in network is regulated by various **protocols** which are described in the Protocols document. Communication is **bidirectional** and is implemented by **mpsc channels**.

The network is unreliable. Drones can **drop packets or crash** according to per-drone simulation parameters (the Packet Drop Praobability, described in the Protocols document) or external input. This mimics real-world scenarios where trees and natural obstacles can interfere with communication and people can shoot at drones.

Although the network is unreliable, we assume that **network partitions** that separate clients from servers never happen so that it is always possible to eventually deliver packets.

Clients and servers use **source routing**: the path that a packet takes through the drone network is determined by the client and server themselves instead of routers. The consequence is that drones do not maintain routing tables.

Clients and servers can repeatedly use the Network Discovery Protocol to discover the other nodes in the network.

The packets must be **eventually delivered** at servers and clients. This must be accomplished by clients and servers with the use protocols. In real world, clients and servers use timeouts to infer that a packet has been dropped or that a drone has crashed. In this project, for simplification, these failures are notified to clients and servers as specific messages that cannot be dropped by drones.

That is, the protocols and the code should answer these questions: What happens if a packet gets lost? What happens if the acknowledgement of a certain packet gets lost?

When a client (respectively, a server) sends a message to a server (respectively, a client), the message is **serialized** and **fragmented** if it exceeds a certain size and

reassembled at the destination. The clients and servers need taking care of these aspects as well.

Project components and contributions

The project consists of several components, each described in its own section below.

Each component is either a group contribution or an individual contribution. The role of each contribution is explained later in this document.

Component	Contribution
Drones	Group
Network Initializer	Group
Servers (+ Assembler, if needed)	Individual
Clients (+ Assembler, if needed)	Individual
Simulation Controller	Individual

Ideally, each drone is called like the group that developed it.

Network initializer

Each group will have to buy 10 drones from other groups.

The **Network Initializer** reads a local **Network Initialization File** that encodes the network topology and the drone parameters (that is, Packet Drop Probability) and, accordingly, starts the drones bought from other groups and sets up the Rust channels for communicating between nodes.

The drone network of a group cannot include the drone of that group.

When starting, each server is connected to at least two drones, and each client is connected to at most two drones.

It is your responsibility to test your system with different Initialization Files.

During the exam, your system will be tested with different Initialization Files of my own creation.

We describe some of the topologies that can be supported in the Topologies file.

Topologies

Servers

Servers are connected at the edges of the drone network, provide different functionalities and are used by clients to implement applications.

When starting, each server is connected to at least two drones, for enhanced reliability.

Servers connect to the drone network with the Network Discovery Protocol which we describe in the Protocols document.

Servers are either Content servers, which further consist of a Text and a Media server, or Communication servers.

Content servers

When implementing a Content server, one must implement two servers:

- A **Text server**, which serves basic text files with references to media.
- A **Media server**, which contains the media referenced by the text.

You are free to initialize Text servers and Media servers as you prefer; that is, you can provide any kind of text with any kind of media, so long as it is supported by the communication protocols. This content needs not be particularly articulated.

Communication servers

Communication servers forward communication from one client to another. In order to use a Communication server, a client must register to it. A communication server can provide the list of clients registered so far.

An example communication for a Communication server is:

- Client A registers.
- Client B registers.
- Client A asks the Communication server for the list of registered clients.
- The Communication server replies A, B to A.
- Client A sends a message <“Hello”, to: B> to the Communication server,

- The Communication server sends message <”Hello”, from: A> to B.

Depending on the nature of the client application, B displays "Hello" and continue the communication.

Client applications

Clients are connected at the edges of the drone network and use the server functionalities to implement applications.

When starting, each client is connected to at most two drones.

Servers connect to the drone network with the Network Discovery Protocol which we describe in the Protocols document.

Clients are either Web browsers or Chat clients.

Web browsers

A Web browser can retrieve the list of all text files from a Text server and can display the file of choice. This can require retrieving and displaying the relevant media as well.

Chat clients

A Chat client can register itself to a Communication servers and retrieve the list of registered clients that are available for chatting. Once it has discovered other clients, it can send a message to the client of choice.

Source Routing and the Network Discovery Protocol

The path that a packet takes through the drone network is determined by the client and server themselves instead of routers. The consequence is that drones do not maintain routing tables. This technique is called source routing.

Clients and servers are at the edges of the network and need to obtain and maintain an understanding of the network topology (”what nodes are there in the network and what are their types”) so that they can determine the route that packets take through the network. This is achieved by the Network Discovery Protocol, which is described in the Protocols document.

Clients and nodes can infer that a drone has crashed. Therefore, they can repeatedly use the protocol to update their understanding of the network topology.

Simulation Controller

The Simulation Controller has two purposes.

It can **send commands** to drones; for example, send a command to **crash a drone**, **add a new neighbor**, change the Packet Drop Probability. These commands change the network topology and thus affect the simulation.

The Simulation Controller can **receive events** from nodes; for example, receive an event that a node has sent a message or a new node has been added as a neighbor. That is, the Simulation Controller is also a monitoring component.

The Simulation Controller must allow the user to send the aforementioned commands to drones.

The Simulation Controller must provide a graphical display of all the nodes in the network; that is, the drones that are deployed, the clients, the servers, and the connections between nodes.

Finally, the Simulation Controller must show the packets that are sent through the channels, the packets that are dropped, the drones that are crashed, etc. Ideally, it also shows the kind of packets being exchanged, since the communication protocols comprise different kinds of packets.

Conceptual distinction: High-level and low-level packets

Clients and servers exchange “high-level” messages (for example, a client sends to a server a command to retrieve the list of files, and the server replies to the client with the list of files).

As anticipated in the description, when a client (respectively, a server) sends a message to a server (respectively, a client), the message is **serialized** and **fragmented** if it exceeds a certain size and **reassembled** at the destination.

Serialization means converting a packet containing dynamically-sized data structures to one containing **fixed-sized arrays** instead. In real world, packets

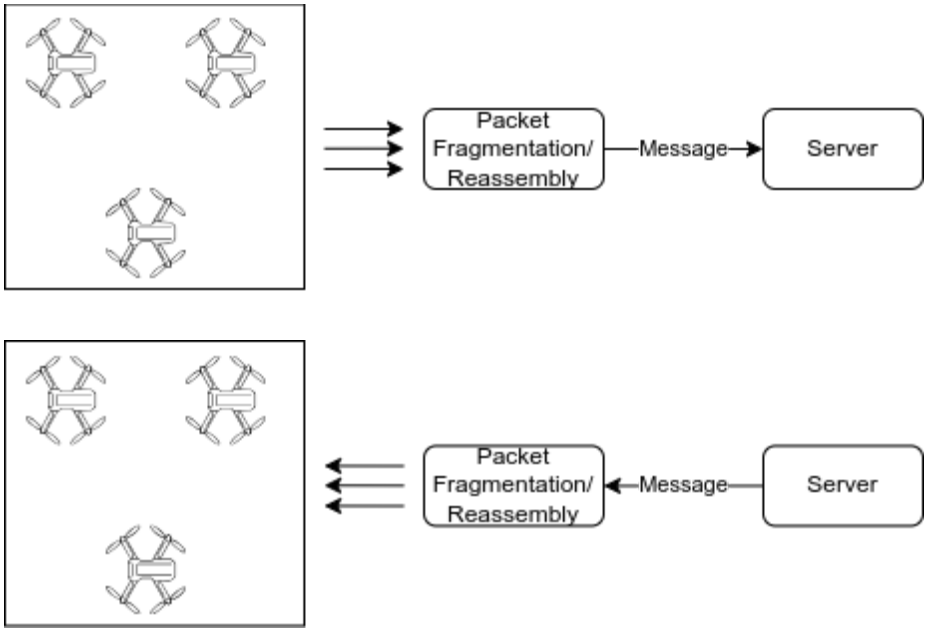
must be serialized to be sent in the network. Pointers and references to memory are not serializable.

Note that fragmentation will likely require some form of packet sequencing.

A fragment is thus a low-level packet. Certain fragments of packets, clarified in the Protocols document, can be dropped.

Thus, clients and servers are also concerned with serialization, fragmentation and reassembly of packets. To ease the distinction and, possibly, the development, one can introduce a sub-component in clients and servers called **Assembler**.

If it is convenient to you, you can use the following structure in your components:



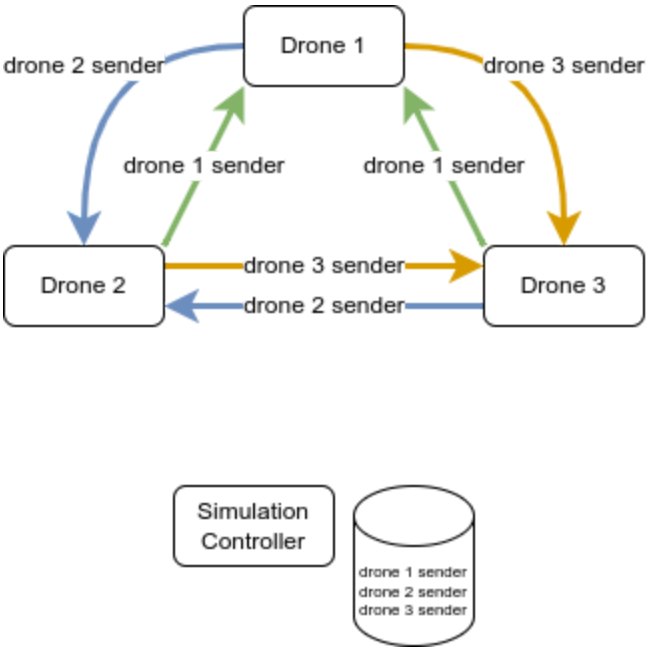
Technical requirements: threads and crossbeam channels

Each node runs on an own Rust thread.

The nodes communicate by exchanging messages through Rust channels.

Remember that, in this project, the communication between two connected nodes is bidirectional.

Rust channels have a `Sender` and a `Receiver` end. The `Sender` end can be cloned. If three drones are connected, the channels look as follows:



New nodes and links between nodes can be added by the Simulation Controller at runtime through simulation commands. Therefore, the Simulation Controller should keep a clone of each sender so that it can provide it to other nodes when needed.

Each node is expected to **listen to several channels**: one for receiving packets from other nodes, and one for receiving simulation commands from the Simulation Controller. Unfortunately, this cannot be achieved with standard Rust channels. Therefore, you will use **crossbeam channels** and the `select!` macro instead. Crossbeam is a popular Rust library that provides a set of tools for concurrent programming. You are recommended to read the following documentations:

https://github.com/crossbeam-rs/crossbeam

https://docs.rs/crossbeam/0.8.1/crossbeam/channel/index.html

https://docs.rs/crossbeam/0.8.1/crossbeam/macro.select.html