// README This project implements a simplified simulation of an ALOHA-style MAC protocol using TCP sockets in C++.

Files

- `server.cpp`: The server application that sends a file in fixed-size frames using TCP to a shared channel.
- `channel.cpp`: The channel application that simulates the shared communication medium.
- `protocol.h`: Shared definitions for frame headers and payloads.
- `Makefile`: For building the project.

How to Compile

Run:

```
make
```

This will generate: - `my_Server` - `my_channel`

How to Run

Start the channel first:

```
./my_channel <chan_port> <slot_time>
```

Start one or more servers:

```
./my_Server <chan_ip> <chan_port> <file_name> <frame_size> <slot_time> <seed> <timeout>
```

Notes

- The channel handles collisions and simulates a shared medium.
- The server uses exponential backoff when it detects a collision.
- The frame includes a header for sender ID and sequence number.
- The simulation outputs statistics such as bandwidth, success, and retransmissions.

Example

```
./my_channel 6342 100
./my_Server 127.0.0.1 6342 testfile.txt 1500 1 123 5
```

Cleaning Up

```
make clean
```

Design Rationale & Implementation Highlights

Ethernet-style Frame Header

Although there's no dedicated receiver module, the assignment required us to design an **Ethernet-inspired header**. The `FrameHeader` in `protocol.h` includes:

- `source_id` and `dest_id` (6 bytes each): Emulate MAC addresses for identifying sender/receiver.
- `ether_type`: Specifies the type of payload (e.g., 0x0800 for IPv4). This field allows extensibility and mimics real Ethernet frames.
- `payload_type`: Distinguishes between data (`0x01`) and noise (`0xFF`) frames.

This structure allows frames to be self-descriptive and processable by a generic receiver if needed.

Collision Detection and Noise Frames

In the **channel** module (`channel.cpp`):

- When more than one server sends a frame during the same time slot, a **collision** is detected.
- A special **noise frame** (with `payload_type == NOISE_FLAG`) is broadcast to all servers to signal the collision.
- Each server that contributed a frame in that slot increments its collision counter.

Server Behavior and Exponential Backoff

In the **server** module (`server.cpp`):

- Files are split into frames of a fixed size (`frame_size`).
- After sending a frame, the server waits for an ACK (same frame echoed back) within a `timeout`.
- If no ACK is received or a noise frame is returned, it applies **exponential backoff**:
  The server waits `k × slot_time` milliseconds, where `k` is a random integer from `[0, 2^attempts - 1]`.
- The server attempts to send each frame up to 10 times before declaring failure.

Select-based Multiplexing

- The **channel** uses `select()` to monitor all sockets (including stdin for EOF/Ctrl+D).
- All sockets are set to **non-blocking** mode to avoid deadlocks and improve responsiveness.

Source Identification

- Each server encodes its `source_id` as the 4-byte process ID (`getpid()`) plus two zeros.
- This allows the channel to track statistics per server using the `source_id`.

Termination and Reporting

- The channel terminates cleanly on `EOF` (Ctrl+D).
- Upon exit, it prints a summary for each server:
  - Number of collisions
  - (Frames count can be easily re-enabled if needed)
- The server prints:
  - Whether transmission was successful
  - Total file size and transfer time
  - Transmission stats (average, max retries)
  - Average bandwidth

These choices follow the assignment guidelines and aim to simulate a realistic ALOHA-like protocol while maintaining clarity, extensibility, and adherence to networking best practices.