Updated individual goals

1. Creating our own modified version of the Rust nRF24l01+ library. The existing Rust library does not seem to be functional, most likely because it is too outdated. But we would still prefer to use Rust in this project for a few reasons:

   - Rust would also run much faster than Python, while being easier to make memory- and runtimesafe than C
   - We both currently take the course in functional programming, where Haskell is used. So it would be nice to solidify own knowledge by using Rust in this course
   - We think that we could make the existing library more effective, by decreasing the address length used in the nRF24l01+ transciever from 5 bytes to 3 bytes

   This could would be evaluated by running the code. If it managed to transmit data between the transcievers, our implementation of the library works. To determine if the decrease in address length is successful, we could show the code determining the address length in the library and that the program still works with it.

   Result:

   We successfully implemented the library, and the code is available in the `rust-nrf24l01` directory. We managed to decrease the address length to 3 bytes.

   We tried to decrease the number of bytes used for CRC, but this caused the system to no longer work. We still believe this to be possible, however it would probably require substantial changes throughout the library.

   In the same way, we tried to disable the ACK, with the idea that this could dramatically increase the throughput with the disadvantage of losing more packets. We think we succeeded in this implementation, as we noticed a small increase in transmission rate while losing more packets. However, we cannot verify this as it does not seem possible to know for sure if the NRF24L01+ transcievers are sending ACKs or not. We tried reading different registers to determine this, but we found it difficult to interpret the data in the registers.

2. Implementing CI/CD to make the development process as easy as possible, by rebuilding and redeploying the code with a single command. Depending on how well this is done, the initial setup could also be minimized through deploying scripts to the Rasberry Pis

   This could be evaluated by running the CI/CD pipeline and checking that

the code is deployed to the Raspberry Pis. If the code is deployed to both Raspberry Pis, where one Pi becomes the base station and the other one a mobile unit, the CI/CD pipeline works.

Result:

We have implemented this in two different ways. First of all, we have a build script and deploy script that we can run to quickly build the code locally using `cross` and deploy to the Pis when testing different changes. Additionally, we have abstracted this into a Makefile so we only need to run the command `make build deploy`.

However, this is not a real CI/CD pipeline. Therefore we also created a GitHub Actions workflow that builds the code and deploys it to the Raspberry Pis. For all practical purposes in this project, the first option is prefered. However, we wanted to understand how to implement a CI/CD pipeline and how it could be used in a real project.

In a real company setting, the first option could be used in developing changes on testing machines, and the second option could be used to actually deploy the code to the production machines.

Testing was also implemented using GitHub Actions, which runs both a `cargo test` as well as a `cargo clippy` every time a new push is made to the main branch. This ensures that every time we push new code, we can be sure that it is working and that it is following the Rust guidelines. Not every file contains unit tests, and we are not able to perform integration testing, but we still think it is a viable proof of concept.

3. Using tailscale to use the same IP address for the Raspberry Pi in the lab and at home. This would simplify the setup when moving the Pis between networks.

   This could be evaluated by connecting to the Raspberry Pis using the tailscale IP address instead of the home- or lab addresses.

   Result:

   Tailscale was installed on both the base and mobile units and made the deployment process much easier. Not only could we use the same IP no matter where we were, but it also made the deployment process for the CI/CD pipeline possible by installing tailscale on the runner.

   Tailscale required some additional configuration by setting the DNS servers manually. It also required some additional software on both the development machines and the Raspberry Pis. However, we think this extra setup has been worth it for the convenience it provides.

4. Being able to set the longge interface as the default network interface, which would allows us to connect to the mobile unit without an Ethernet or WLAN connection. This would not bring us any performance benefit, but would however provide an additional opportunity to access the Pi even without the Ethernet cable. We would also learn even more about network interfaces through implementing this feature.

   This could be evaluated by checking the result from the `ip route list`. If our longge is the default interface, the implementation works. We could also check it by connecting to the Raspberry Pi without an Ethernet or WLAN connection.

   Result:

   Setting longge as the default interface can be done on the mobile unit by passing `--longge-default true`. This will create a new `ip route` entry setting the longge IP as the default gateway.

   We tested it and it worked. However, this caused the deployment of files to go through the longge interface by default, even when the Ethernet cable was connected. The low throughput therefore made the development process easier with the setting turned off. Instead, we used the Ethernet connection by default and specify the longge interface when needed. For example `ping -I longge 8.8.8.8`.