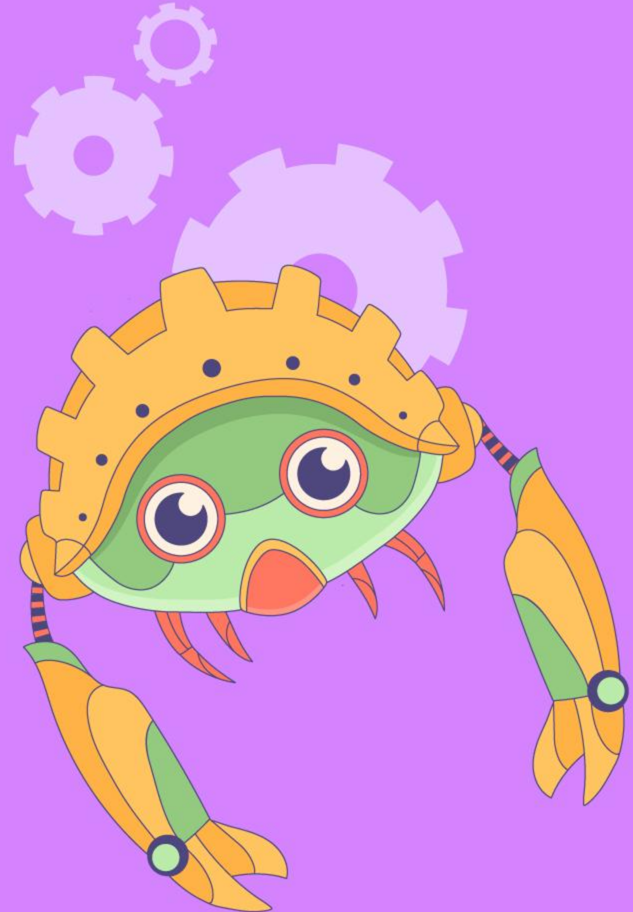
Jukka Taimisto

SW Developer, Founder @ SensorFleet

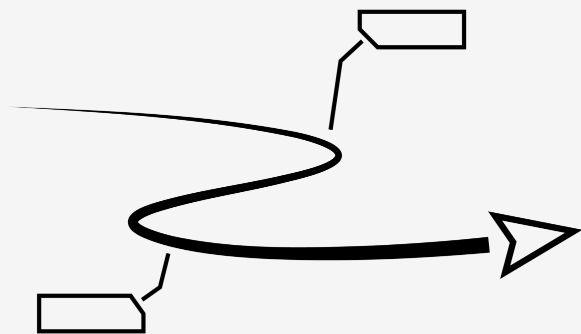
Composable and safe architecture for network packet monitoring



Whoami

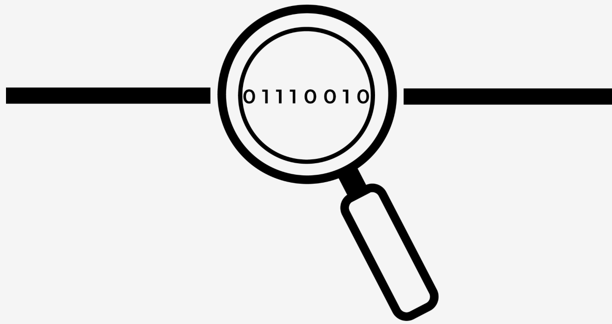
- Founder & SW Developer at SensorFleet.
- Over 20 years of experience on writing and breaking protocol implementations.
- Background in C, Java, Python, Go, focusing more on Rust for few years.
- jtaimisto@gmail.com

Roadmap



- Architecture
- Benefits
- Parsing network data
- Testing

The problem



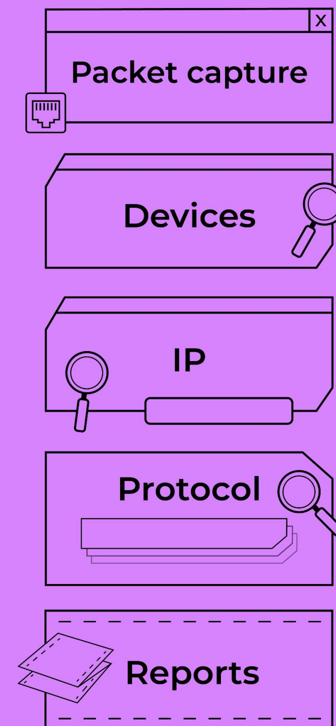
- Listen for network traffic from interface.
- Identify communicating devices (by MAC address).
- Collect protocol -specific information.
- Relay collected information for further processing.

Design goals

- Simple architecture.
- Easy to expand with new protocol analyzers / functionality.
- Allow handling multiple packets concurrently.

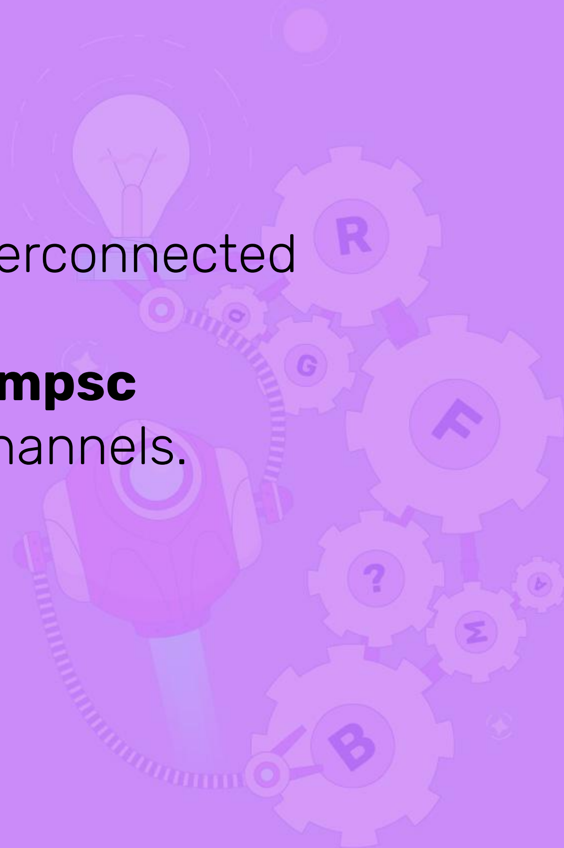
Components

- Read packets from network.
- Detect communicating devices.
- Collect IP addresses.
- IP protocol analysis.
- Output collected information.

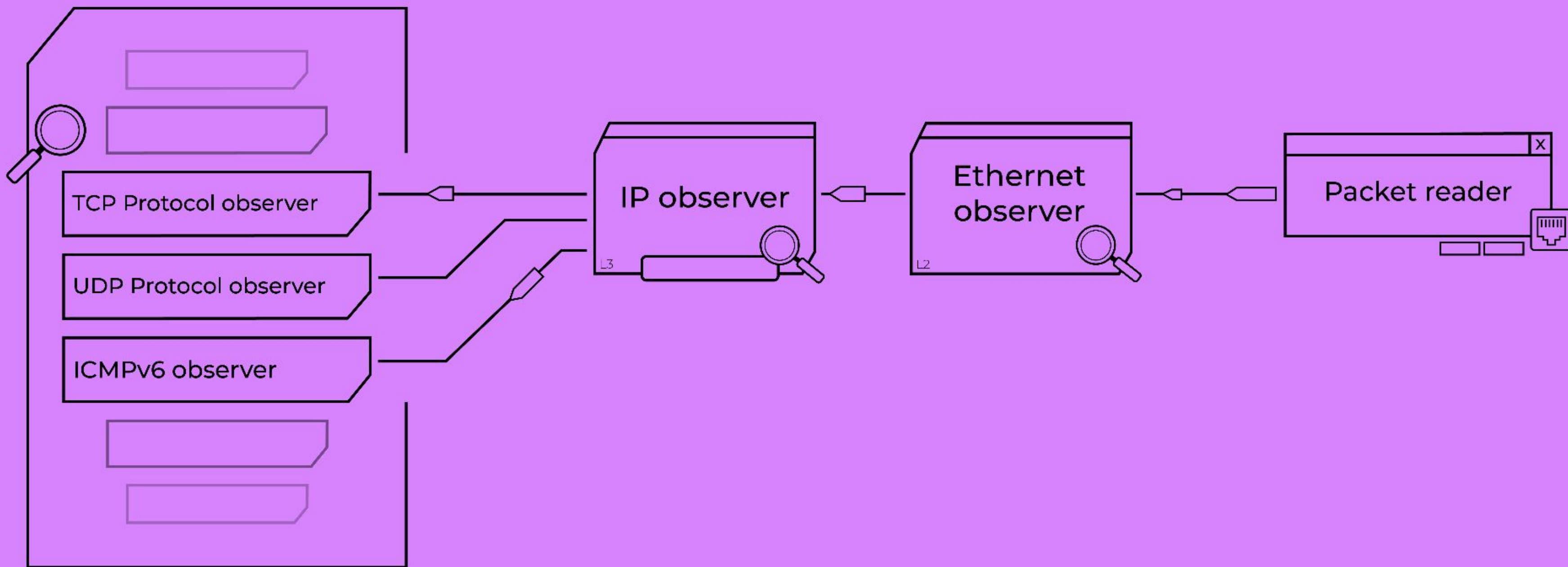


Architecture

- Split packet processing to multiple interconnected tasks called observers.
- Communication between tasks using **mpsc** (multiple senders, single consumer) channels.
- Divide observers by network layers.
- Use **async** rust and **tokio** to provide asynchronous runtime.



Observers



Packet reader

- use **luomu-libpcap** to for libpcap packet bindings.
- Sends received packets to **mpsc** channel.

```
let mut pcap = AsyncCapture::new(self.handle)?;
loop {
    tokio::select! {
        _ = &mut stop => {
            break;
        },
        next_pkt = pcap.try_next() => {
            match next_pkt {
            // write packet to channel
            }

        }
    }
}
```

Observer

- Receives packet and context from channel.
- Extracts information, observations.
- Forwards rest of the packet to output channel.

```rust
pub async fn run_udp_scan(self, mut bufs: BufReceiver, next: FrameSender) {
    while let Some((buf, frame)) = bufs.recv().await {
        let input = buf.as_input();
        // Handle input
        if let Err(err) = next.send(frame) {
            tracing::warn!("Error while sending to next channel: {}", err);
        }
    }
}
```

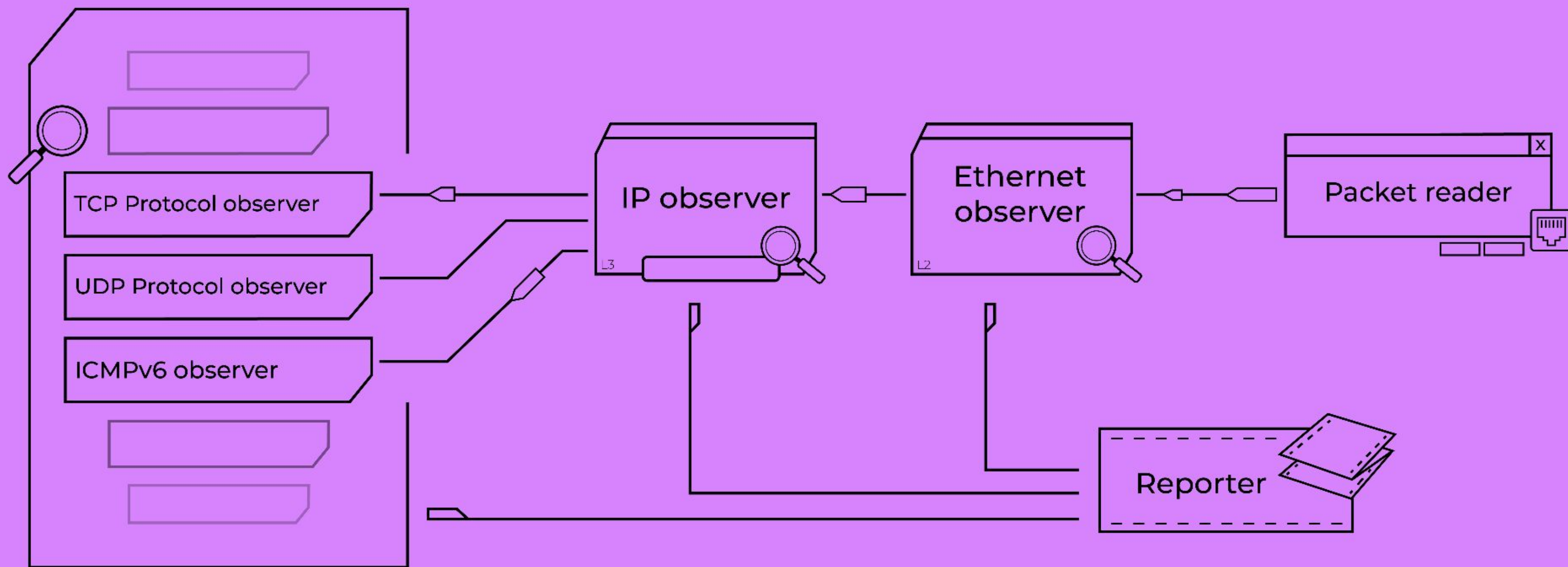
Observer state

- Observers own the packet data received from channel.
- Observers run on their own task and own their own state data.
- No need for locking between observers.

Observations

- Observers collect information from each packet they receive.
- Instead of collecting information into centralized database, have each observer send its findings using **mpsc** channel.

Observers & Reporter



Reporter

- Run on its own **async task**.
- Receives observations from channel.
- Aggregates information about devices.
- Provides periodic reports to central manager.

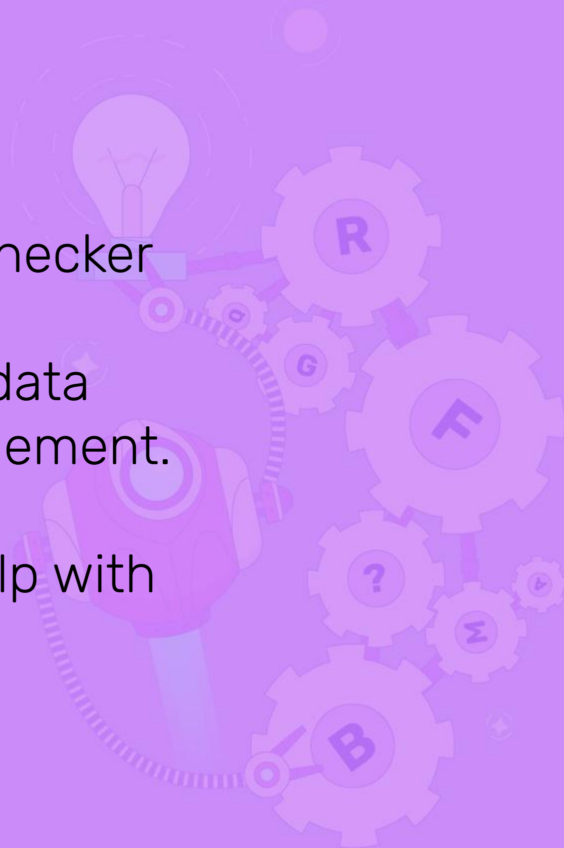
```rust
pub async fn report_producer(
    mut observations: ObservationReceiver,
    socket: UnboundedSender<String>,
    report_interval: Duration,
) {
    let mut assets: HashMap<MacAddr, AssetInfo> = HashMap::new();
    let mut interval = time::interval(report_interval);

    loop {
        tokio::select! {
            re = observations.recv() => {
                // handle observation, break on error
            }
            _ = interval.tick() => {
                produce_reports(&mut assets, &socket).await;
            }
        };
    }
}
```

Design goals

- Simple
  - Separation of components.
  - No locking needed.
- Expandable
  - New observers can be easily added.
- **Tokio** provides concurrent and parallel execution of observers.

Ownership & message passing

- Rust’s ownership model and borrow checker ensure data ownership.
- Ownership guides you to think about data
  - Simple solutions are easier to implement.
  - “what would rustc say about this”.
- While not unique to Rust, channels help with concurrency.



Rate limiting with channels

- Channels with capacity help with rate limiting.
- Either block on **send()** if channel is full.
- .. or use **try_send()** and drop packet if channel is full.

```rust
pub struct DroppingSender<T>(mpsc::Sender<T>);

impl<T> DroppingSender<T> {
    pub fn send(&self, data: T) -> Result<bool> {
        match self.0.try_send(data).map_err(Error::from) {
            Ok(()) => Ok(true),
            Err(Error::Dropped) => {
                tracing::warn!("channel dropping data");
                Ok(false)
            }
            Err(err) => Err(err),
        }
    }
}
```

Channel types for different purposes

- Using type aliases help with refactoring or if channel behavior needs to be changed.
- Different channel types according to channel purpose.

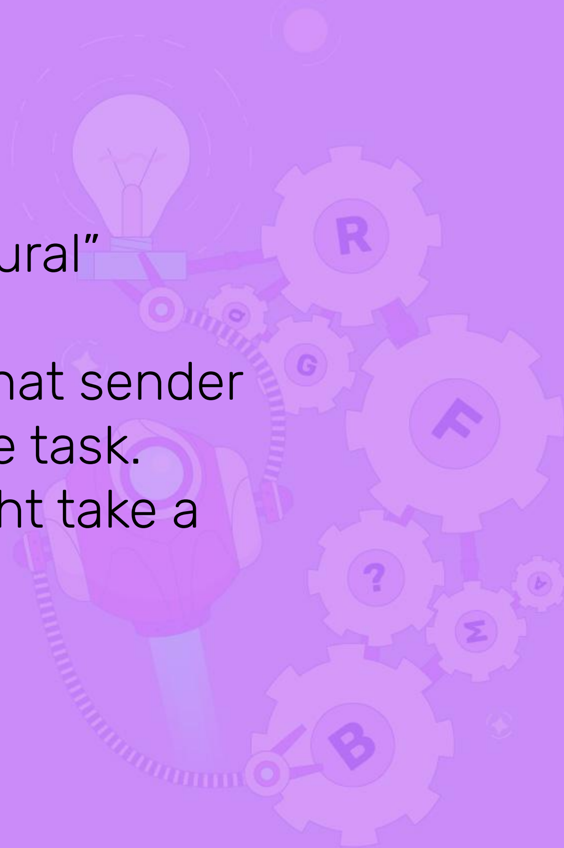
```rust
pub type BufSender = DroppingSender<(Buf, Frame)>;
pub type BufReceiver = mpsc::Receiver<(Buf, Frame)>;

pub fn buf_channel() -> (BufSender, BufReceiver) {
    let (tx, rx) = mpsc::channel(BUF_CHANNEL_SIZE);
    (DroppingSender::create(tx), rx)
}
```

Fan-out

- Observer model allows running multiple instances of same observer.
- “Previous” observer or packet reader needs to schedule packets to multiple channels.
- Note the internal state if running multiple instances
  - Use **mutex** to protect the state.

Program termination

- Tasks-and-channels model allow “natural” termination of observer tasks.
- Once the **recv()** operation indicates that sender has stopped, it is time to terminate the task.
- If using channels with capacity, it might take a while as channels are emptied



Graceful shutdown

- When user signals it is time to stop:
  - Terminate packet reader
  - Wait for all observers and reporter to terminate.

```rust
let Ok(signals) = Signals::new([SIGINT, SIGTERM]) else {
    panic!("Could not install sighandler");
};
let sig_handle = signals.handle();
let sig_handler = tokio::spawn(signal_handler(signals, stop_tx));
// ...
let packet_reader = tokio::spawn(capt.read_packets(packet_tx, stop_rx));
if let Err(error) = packet_reader.await {
    tracing::warn!("Packet Reader exited with error: {}", error)
}
tracing::info!("packet reader stopped");
// stop the signal handler
sig_handle.close();
// wait for all other tasks to terminate
futures::future::join_all(handles).await;
// and finally wait for signal_handler to terminate
if let Err(e) = sig_handler.await {
    tracing::warn!("Signal handler terminated with error: {}", e);
}
```

Unit testing

- Observer -model allows easy splitting for unit and system tests.
- Unit test each observer
  - Provide input context, expect observations.
- Focus on testing observer functionality
  - Packet parsing should be tested separately.

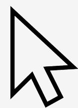
System tests

- Use system tests to test combinations of multiple workers.
- Provide packet input, expect observations as output

```
async fn test_passthrough_tcp4() {
    let pkt = "\
        f09fc266b02b1c57dc5bb65508004500003400004000400063567c0a801e60344\
        3f8bd7f001bb2cadaf7c4674a62a801007ff14ac00000101080a5e1160089109\
        631d";
    let raw = parse_hex_string(pkt).unwrap();
    let mut ctx = Context::create();
    let expected_obs = [
        Observation::Asset(AssetObservation {
          /// ...
        }),
        Observation::Asset(AssetObservation {
           // ...
        }),
    ];
    ctx.send_packet(raw);
    ctx.expect_frame(expected_flow).await;
    for obs in expected_obs {
        ctx.expect_observation(obs).await;
    }
    ctx.teardown().await;
}
```

Parsing packets

01110010
01110101
01110011
0111100



- Specification defines layout of data within a packet
- When parsing, data is interpreted according to specification
- Interpret data as values
  - Byte ordering
  - Constants
  - Substructures

Length

- One of the key things is to figure out how much data to read.
- While Rust protects you from reading outside of allocated buffer, length checks still need to be made.
  - Panics occur when trying to read data past the buffer.

Parsing with Rust

- We use **untrusted** crate for reading untrusted data.
- **untrustended** crate adds extensions for reading common values.
- Use **Result** type and **?** to fail fast

```
pub fn parse_header(input: &mut Reader) -> Result<TcpHeader, Error> {
    let src_port = input.read_u16be()?;
    let dest_port = input.read_u16be()?;
    let seq = input.read_u32be()?;
    let ack = input.read_u32be()?;

    let offset = input.read_byte()?;
    let data_offset = (offset >> 4) * 4;
    let flags = input.read_byte()?.into();
    if !(20..=60).contains(&data_offset) {
        tracing::warn!("Unexpected length {} for TCP header", data_offset);
        return Err(Error::ParseError);
    }
    // skip window, checksum, urgent pointer and options
    // we have already consumed 14 octets from the header.
    input.skip(usize::from(data_offset) - 14)?;

    Ok(TcpHeader {
        ports: PortPair::new(src_port.into(), dest_port.into()),
        flags,
        seq,
        ack,
    })
}
```

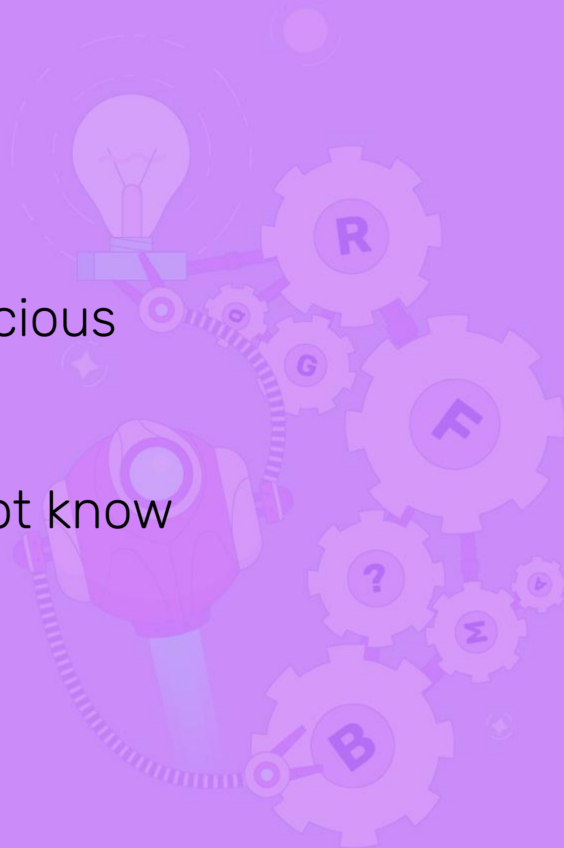
Parsing with rust

- Use **enums** and/or newtype pattern to add type safety and checks for validity
- Borrow checker will help with copying/passing reference

```rust
fn parse_dhcp_option<'a>(input: &mut Reader<'a>) -> Result<Option<DhcpOption<'a>>, Error> {
    let mut code = DhcpOptionCode::Pad;
    while code == DhcpOptionCode::Pad {
        code = input.read_byte()?.try_into()?;
    }
    if code == DhcpOptionCode::End {
        // End of options
        return Ok(None);
    }
    let option_length = input.read_byte()?;
    Ok(Some(DhcpOption {
        code,
        value: input.read_bytes(usize::from(option_length))?,
    }))
}
```

Testing

- Parsers should be tested extensively
  - If it comes from internet, it is malicious
- Unit tests
  - Don’t forget negative test cases.
- Fuzzing helps to find issues you did not know existed. Apply liberally



Conclusions

- Connecting tasks with channels allows composable architecture.
- Rusts ownership model <3
- When writing protocol parsers, check your lengths and test, test and test.

Thank you!
Questions?