024-Rust-Netzwerkanalyse

Rust- und Systemaufgabe
Anforderungsniveau: Schwer

1 Ziel der Aufgabe

Sie schreiben eine Rust-Bibliothek, die große Mitschnitte des Netzwerkverkehrs im klassischen pcap-Format streamend einliest, die Pakete zu Verbindungen zusammenfasst und diese zu Kennzahlen verdichtet. Die Dateien sind größer als der Arbeitsspeicher – gefragt ist also nicht das Zerlegen einzelner Pakete, sondern ein Entwurf, der bei einer 8-GB-Datei genauso wenig Speicher braucht wie bei einer 80-MB-Datei.

Veranschlagte Bearbeitungszeit: 6–8 Stunden

Geprüft werden:

- Rust 2021, stabile Toolchain, sauberes Fehler- und Typmodell
- Streamende Verarbeitung mit fester Speicherobergrenze
- Bidirektionale Verbindungsbildung inklusive Alterung und Verdrängung
- Verdichtung zu Kennzahlen, darunter eine speicherbegrenzte Rangliste
- Bibliotheksentwurf: schmale Schnittstelle, Ein- und Ausgabe außerhalb der Logik; Tests auf beschädigten Eingaben

2 Aufgabenbeschreibung

1. **Aufbau des Projekts**

   Eine einzige Kiste `netzblick` mit Bibliothek (`src/lib.rs`) und Programm (`src/main.rs`). Die gesamte Logik liegt in der Bibliothek, das Programm ist nur Hülle für Kommandozeile, Datei und Ausgabe. Die Bibliothek schreibt nichts nach `stdout` und beendet den Prozess nicht; alles, was schiefgehen kann, verlässt sie als `Result` mit einem eigenen Fehlertyp (`thiserror` ist erlaubt).

2. **Einlesen ohne die Datei zu laden**

   Gelesen wird ausschließlich das klassische pcap-Format, blockweise über einen Puffer fester Größe. pcapng ist nicht gefordert und wird an seiner Kennung erkannt und mit verständlicher Fehlermeldung abgelehnt. Geeignet sind `pcap-parser` (mit `PcapReaderIterator`: `next()`, `consume()`, `refill()`) und `etherparse` für die Schichten darüber; ein eigener Parser ist zulässig, aber nicht gefordert.

   Zu beherrschen ist der Linktyp `ETHERNET` (1), darüber IPv4, IPv6, TCP, UDP und ICMP. Achtung bei eigenen Testdaten: `tcpdump -i any` schreibt heute `LINUX_SLL2` (276), nehmen Sie also eine echte Schnittstelle oder `-i lo`. Jeder andere Linktyp, unbekannte Protokolle und abgeschnittene Pakete (`caplen < origlen`) sind kein Grund zum Abbruch, sondern werden je Grund gezählt und im Bericht ausgewiesen.

3. **Verbindungen bilden**

   Eine Verbindung wird über das Fünftupel gebildet und richtungsunabhängig geschlüsselt: Hin- und Rückrichtung ergeben einen Eintrag, die Zählwerte bleiben je Richtung getrennt. Bei Protokollen ohne Portbegriff setzen Sie den Port auf 0.

```
pub struct Verbindungsschluessel {
    pub proto: Protokoll,      // Tcp | Udp | Icmp | Andere(u8)
    pub a: SocketAddr,
    pub b: SocketAddr,         // a < b, damit die Richtung egal ist
}

pub struct Verbindung {
    pub schluessel: Verbindungsschluessel,
    pub beginn: Zeitpunkt,  pub ende: Zeitpunkt,
    pub pakete_hin: u64,    pub pakete_zurueck: u64,
    pub bytes_hin: u64,     pub bytes_zurueck: u64,
    pub tcp_flaggen: u16,
    pub abschluss: Abschluss, // Fin | Reset | Zeitablauf | Ende | Verdraengt
}
```

   `hin` ist die Richtung des ersten gesehenen Pakets, `tcp_flaggen` sammelt alle in der Verbindung beobachteten Flaggen über beide Richtungen.

4. **Speicherobergrenze**

   Der Speicherbedarf darf nicht mit der Dateigröße wachsen; die Tabelle offener Verbindungen bekommt harte Grenzen, die der Aufrufer setzt:

```
pub struct Grenzen {
    pub max_offene_verbindungen: usize,  // Vorgabe 200_000
    pub leerlauf_tcp: Duration,          // Vorgabe 300 s
    pub leerlauf_udp: Duration,          // Vorgabe  60 s
    pub zaehler_rangliste: usize,        // Vorgabe   1_000
}
```

   Beendet ist eine Verbindung, wenn sie länger als die Leerlaufzeit ruht oder – bei TCP – beide Seiten FIN gesendet haben beziehungsweise ein RST kam; danach geht sie an die Senke und verlässt die Tabelle. Die Alterung richtet sich nach der Zeit *im Mitschnitt*, nicht nach der Uhr des Rechners, und darf nicht bei jedem Paket über die ganze Tabelle laufen. Ist die Obergrenze erreicht, wird die am längsten untätige Verbindung verdrängt und mit `Abschluss::Verdraengt` ausgegeben. Zeitstempel sind nicht immer monoton; beschreiben Sie im README, wie Sie damit umgehen, ohne abzustürzen.

5. **Kennzahlen verdichten**

   Ein Durchlauf liefert zusätzlich einen Bericht, der ebenfalls unter der Speichergrenze steht – eine vollständige Liste aller Gegenstellen scheidet aus. Für die Rangliste setzen Sie ein Verfahren mit begrenztem Zählerfeld ein (Misra–Gries beziehungsweise Space-Saving) und weisen je Eintrag die Fehlerschranke aus. Die Zahl der Zähler ist `zaehler_rangliste` aus `Grenzen` ($k = 1000$) und muss vom Aufrufer kleiner gesetzt werden können – sonst lässt sich die Fehlerschranke nicht prüfen.

```
pub struct Bericht {
    pub pakete: u64, pub bytes: u64, pub verbindungen: u64, pub verdraengt: u64,
    pub zeitraum: (Zeitpunkt, Zeitpunkt),
    pub je_protokoll: BTreeMap<Protokoll, Zaehlwerk>,
    pub top_gegenstellen: Vec<Rangplatz>, // { adresse, bytes, fehler }
    pub paketgroessen: [u64; 8],          // Klassen 0-64 ... 1519+
```

```
    pub uebersprungen: Uebersprungen,     // je Grund gezaehlt
}
```

6. **Schnittstelle der Bibliothek**

   Fertige Verbindungen gibt die Bibliothek sofort an eine Senke ab, statt Millionen Ergebnisse zu sammeln.

```
pub trait Senke {
    fn verbindung(&mut self, v: &Verbindung) -> Result<Weiter, Fehler>;
    fn fertig(&mut self, b: &Bericht) -> Result<(), Fehler> { Ok(()) }
}

impl Analyse {
    pub fn neu(grenzen: Grenzen) -> Self;
    pub fn auswerten<R: Read, S: Senke>(
        &mut self, quelle: R, senke: &mut S) -> Result<Bericht, Fehler>;
}
```

   Die Eingabe ist ein beliebiges `Read`, damit sich der Durchlauf auch aus einer Pipe speisen lässt. `Weiter` ist Ihr eigener Rückgabetyp der Senke (etwa `Weiter::Ja` und `Weiter::Abbruch`); bei Abbruch endet der Durchlauf geordnet und gibt den bis dahin gesammelten Bericht zurück. Die öffentliche Schnittstelle bleibt schmal: keine Typen aus Abhängigkeiten nach außen, jedes öffentliche Element dokumentiert, `#![deny(missing_docs)]` in der Bibliothek.

7. **Programm und Messung**

   Ein Werkzeug über `clap`, das die Bibliothek benutzt und sonst nichts tut. Die Verbindungen werden fortlaufend während des Durchlaufs als `ndjson` geschrieben; der Bericht steht erst am Ende fest und geht als JSON in eine eigene Datei. Der Fortschritt geht nach `stderr`, damit `stdout` weiterverarbeitbar bleibt.

```
netzblick --ein mitschnitt.pcap --aus verbindungen.ndjson \
          --bericht bericht.json --max-verbindungen 200000
cat mitschnitt.pcap | netzblick --ein - > verbindungen.ndjson

# Testdaten (eigener Rechner, kein fremder Netzverkehr), dann messen
sudo tcpdump -i lo -w testdaten/eigen.pcap -c 20000   # Linktyp ETHERNET
/usr/bin/time -v ./target/release/netzblick --ein gross.pcap --aus /dev/null
```

   Für die Speichermessung taugt eine mehrfach zusammengehängte Kopie derselben Datei nur bedingt: sie verlängert die Eingabe, ohne die Zahl gleichzeitig offener Verbindungen zu erhöhen – nehmen Sie einen Mitschnitt mit vielen verschiedenen Fünftupeln. Beispielmitschnitte: `https://wiki.wireshark.org/SampleCaptures`; ins Repository gehören nur kleine, selbst erzeugte Dateien.

3 Anforderungen

- **Speicher**: Der Spitzenspeicher hängt an den Grenzen, nicht an der Dateigröße, und bleibt bei Vorgabewerten auch bei mehreren Gigabyte Eingabe unter 512 MB. Kein `read_to_end`, kein Vektor mit allen Verbindungen.
- **Durchsatz**: mindestens 100 MB/s auf einem Kern durchschnittlicher Hardware, gemessen mit Ausgabe nach `/dev/null` auf einem Mitschnitt mit durchschnittlicher Paketgröße ab etwa 500 Byte; geben Sie zusätzlich Pakete je Sekunde an, weil die Zielmarke sonst von der Paketgröße

abhängt. Messwert, Hardware und Methode stehen im README; eine Messung mit `time` genügt, `criterion` ist nicht gefordert. Parallelisierung ist erlaubt, aber nicht gefordert.

- **Robustheit**: Beschädigte oder abgeschnittene Eingaben führen weder zu Absturz noch zu Endlosschleife. Kein `unwrap()`, `expect()` oder `panic!` auf Pfaden, die vom Dateiinhalt abhängen; `#![forbid(unsafe_code)]` ist gesetzt.
- **Tests**: Schlüsselbildung (beide Richtungen ergeben denselben Schlüssel); Alterung und Verdrängung (Leerlauf, rückwärts laufende Zeitstempel, Erreichen der Obergrenze); ein Vergleichstest gegen einen kleinen, beiliegenden Mitschnitt mit erwarteter Ausgabe; mindestens ein Test mit fehlerhafter Eingabe; die Rangliste gegen eine exakte Zählung, und zwar mit so klein gesetztem Zählerfeld, dass die Fehlerschranke wirklich greift – bei $k = 1000$ und zwanzig Gegenstellen prüft der Test nichts.
- **Werkzeuge**: `cargo fmt -check` und `cargo clippy -all-targets - -D warnings` laufen ohne Befund; Abhängigkeiten bleiben sparsam und werden im README je in einem Satz begründet.

4 Abgabe

- Git-Repository mit nachvollziehbarer Historie und Dokumentation über `cargo doc`.
- README mit: Startanleitung, Aufbau von Verbindungstabelle und Alterung, Ranglistenverfahren samt Fehlerschranke, Messwerten für Durchsatz und Spitzenspeicher inklusive Hardware, Umgang mit nicht monotonen Zeitstempeln und bekannten Grenzen.
- Keine echten Kundendaten, Zugangsdaten oder Kundennamen im Repository, auch nicht in Testdaten; Mitschnitte aus fremden Netzen gehören nicht hinein.
- Ein nachvollziehbarer Ablauf für den Prüfer:

```
cargo build --release && cargo test
cargo clippy --all-targets -- -D warnings
./target/release/netzblick --ein testdaten/beispiel.pcap | head -5
```

5 Bewertungskriterien

- **Speicherverhalten** – Hauptkriterium: Der Spitzenspeicher bleibt bei wachsender Eingabe flach. Eine schnelle Lösung, die die Datei in den Speicher zieht, gilt als nicht gelöst.
- **Korrektheit der Verdichtung** – Zählwerte je Richtung, Abschlussgründe und Rangliste sind gegen eine exakte Referenz belegt.
- **Bibliotheksentwurf** – Die Schnittstelle ist benutzbar, ohne den inneren Aufbau zu kennen; Ein- und Ausgabe stecken nicht in der Logik, Fehler sind eigene Typen.
- **Robustheit und Durchsatz** – Kaputte Eingaben führen zu Fehlern oder gezählten Übersprüngen, nie zu einem Absturz; die Zielmarke wird erreicht und die Messung ist nachvollziehbar beschrieben.
- **Aussagekraft der Tests** – Geprüft wird, was im Betrieb schiefgeht: Alterung, Verdrängung, beschädigte Dateien. Formatierung und Lints sind sauber, Commits lesbar.