panikzettel.htwr-aachen.de

Communications Systems Engineering

Version 2 | 21.03.2024

Jonas Schneider Adrian Groh Eric Langen

Inhaltsverzeichnis

1 Einleitung und Vorwissen

Vorwort

Dieser Panikzettel ist Open Source auf `https://git.rwth-aachen.de/jonas.max.schneider/panikzettel` . Wir freuen uns über Anmerkungen und Verbessungsvorschläge (auch von offiziellen Quellen).

Communications Systems Engineering ist auch so ein Fach wo man einfach das Wissen brauch. Es gibt kaum Schemaaufgaben. Wir haben hier nur wichtige Themen zusammengefassen, aber wie ihr seht, es ist trotzdem ein Brocken.

Vorwissen

- Datcom
- Bus

system Stellt irgendeine Funktion bereit. Z.B. ein Webserver
service Eine konkrete Implementierung einer Funktion. Z.B. ein *socket*
protocol Beschreibt die Interaktionen zwischen services (genauer service-entities) und stellt eigene Bereit. Z.B. TCP -> zuverlässige Kommunikation.

2 Communication Systems Engineering

2.1 Protokolle

Ein Protokoll muss diese Sachen definieren:
- Regeln (*rules*), wie Daten verarbeitet und ausgetauscht werden
- Format (*syntax*), was valide Nachrichten sind
- Meaning von Nachrichten (*semantik*)

Mit Hilfe von Protokollen können Services implementiert werden.

Oft können Protokolle mit Hilfe von *state machines* abstrahiert werden.

2.2 Communication Systems

Communication Systems sind oft sehr komplex, deshalb ist es sinnvoll, sie in mehrere getrennte Komponenten, oft in *layers*, aufzuteilen, wie es bei ISO/OSI und TCP/IP der Fall ist.

Um in der Praxis den overhead zu minimieren, werden im TCP/IP-Modell einige Schichten des ISO/OSI Modells (z.B. Application, Presentation, Session layer) zusammengefasst.

2.3 Protocol Design

Beim design eines Protokolls gibt es einige Sachen zu beachten. Das Protokoll sollte effizient laufen, es sollten also z.B. *copy* Operationen vermieden werden. Außerdem sollte das Protokoll komplett und korrekt sein, also auch für seltene und spezielle Fälle definiertes Verhalten haben, um die zuverlässigkeit zu gewährleisten.

Clayton Tunnel
Aufgrund eines unvollständigen Protokolls und einer limitierten Auswahl an Nachrichten im Protokoll ist es zu einem Unfall im Clayton Tunnel gekommen. (Genauer Ablauf siehe Folien)

Demnach sollten Protokolldevelopment nach dem üblichen Muster: Analyse → Design → Impl. → Testing → Evaluation → Analyse laufen.

3 Network Programming

3.1 Sockets

Kernel seitig werden *sockets* benutzt, um network requests zu machen. Hier gibt es verscheidene Typen:

Definition: Sockettypen

- `SOCK_RAW`: Bloß IP.
- `SOCK_STREAM`: Bidirektionaler zuverlässiger (auch mit Reihenfolge) Stream. Z.B. meistens TCP
- `SOCK_DGRAM`: unzuverlässiger ohne Reihenfolge Kommunikation. Z.B. meistens UDP
- `SOCK_RDM`: Wie `DGRAM` aber zuverlässig
- `SOCK_SEQPACKET`: Wie `STREAM` aber fixed size packet.

Diese Sockets können dann verschiedene Protokol Families (PF) benutzen, u.a. IP, Bluetooth, ATM,...

Ein Protokoll besteht also aus einer Protocol Family (PF) und einem Socket Type.

Sockets können entweder als Server oder Client funktionieren, der Unterschied ist, dass Server passiv auf Verbindungen warten (`listen`) und Verbindungen akzeptieren kann (*accept*), während sich Clients aktiv mit einem entfernten Host verbinden.

Sie werden dann an einem bestimmte Adresse/Port via `bind` gebunden.

Die tatsächliche Kommunikation läuft dann über `send/write`, `recv/read`, bzw. `sendto()`, `recvfrom()` für verbindungslose Protokolle.

Die tatsächliche Implementierung/Funktionsweise der Protokolle bleibt dabei transparent für den Benutzer. Z.B. übergeben wir einem `SOCK_STREAM` ganze Daten, obwohl es *packetized* wird.

Am Ende wird der Socket dann via `shutdown` & `close` geschlossen.

Konfigurationen
Die meisten Befehle lassen sich mittels Argumenten konfigurieren (z.B. bei `shutdown`, ob keine weiteren receives/sends oder beides geschehen sollen).

Aber auch die Sockets selber haben Konfigurationen, die mit `getsocketopt/setsocketopt` gelesen/geschrieben werden können. Hierbei lässt sich das Level auswählen (z.B. TCP configs oder Socket API configs) und die Konfiguration. Dann wird entweder in den übergeben Pointer geschrieben oder von ihm gelesen.

Best Practices
- Beim Senden und Empfangen immer die „length fields“ überprüfen
- Gehe nicht davon aus, dass der Kernel die Pakete auf eine bestimmte Weise „packetized“
- Gehe immer davon aus, dass alles schief gelaufen ist, um von Fehlern recovern zu können

3.2 Non-blocking I/O

Standardmäßig sind alle network calls *blocking*, was auf Servern sehr ineffizient ist. Mit non-blocking I/O wird beim Lesen oder Schreiben nicht geblocked, sondern ein Fehler geworfen, wenn die Operation fehlschlägt.

Hierfür ist `select/poll` eine wesentliche Funktion. Sie ermöglicht es, auf mehrere Sockets gleichzeitig zu warten und zu reagieren, wenn einer von ihnen lesbar/schreibbar wird oder ein Fehler auftritt. Allerdings wächst die Komplexität dieser Funktionen linear mit der Anzahl der Sockets. Genauer bei vielen Sockets dauert das iterative durchsuchen der *wait queues* zu lange.

`Epoll` soll dieses Problem lösen. Die syntax ist aber anders:
- *Level triggered*: Ähnlich wie `select`. Fragt ab, ob Daten zu lesen sind.
- *Edge triggered*: Fragt ab, ob **neue** Daten zu lesen sind.

Würde ein vorheriger `recv` call nicht alle Daten des Buffers lesen, würde *edge triggered* epoll keine weiteren Daten anzeigen.

3.3 TCP Optionen

Packetizing

Am Anfang von TCP hat jede `write` Operation ein Packet erzeugt, das führte jedoch zu dem *Small Packet Problem* (Nagle). Da jedes TCP Paket noch 40 Byte an header (IP + TCP Header) Daten beinhaltet, führen viele kleine Pakete zu einem großen Overhead.

Nagle's Alogithmus behebt dieses Problem wiefolgt:

Algorithmus: Nagel's Algoritmus

- Volle Segmente werden sofort gesendet
- Kleine Segmente werden zurückgehalten
  - Wenn genug Segmente für ein volles Segment vorhanden sind oder die vorher gesendeten Daten ACKed wurden, wird das neue Segment gesendet (1 RTT Wartezeit)
  - Wenn vorher keine Daten gesendet wurden, werden auch kleine Segmente sofort gesendet

Es befindet sich also immer nur ein nicht volles Segment gleichzeitig auf der Leitung.

Nagle's Algorithmus kann aber zu Problemen führen, wenn ein Paket größer als die MSS gesendet wird, da dann das letzte Fragment des Pakets erst bis zu 1RTT später gesendet wird (oder länger bei *delayed ACKs*).

Delayed ACKs

TCP benutzt *delayed ACKs*, weil beim Empfang von Daten, die zugehörige Anwendung meistens kurz darauf eine Antwort sendet. Indem das ACK kurz verzögert wird, kann es so häufig mit dieser Antwort mitgesendet werden. Dabei gibt es einen Timeout von 0.5s. Dadurch wird wie bei Nagle der Overhead reduziert.

Bei einer Sequenz von vollen Segmenten sollte mindestens jedes zweite ACKed werden.

Die Kombination von Nagle und delayed ACKs kann zu großen Wartezeiten (0.5s + RTT) führen, wenn die Anzahl der vollen Fragmente ungerade ist.

Beide Features können bei Bedarf mit Socket Optionen deaktiviert werden.

TCP Fast Open

Normalerweise ist in TCP ein 3-Way Handshake vorgesehen (e.g. `SYN` → `SYN, ACK` → `ACK, DATA`). Die führt zu einem RTT (Round Trip Time) Verzögerung. Da wir für HTTP soweiso meist nur eine Anfrage stellen und dann die Antwort wollen, wollen während des `SYN,ACK` auch noch Daten übertragen.

Probleme die jedoch dabei auftreten:

- Doppelte SYN: Was die `SYN` anfrage dupliziert wird? Wir können nicht wissen, dass es sich nicht um eine neue Anfrage handelt.
- DoS attacks (Resource exhaustion attack -> & Reflection Attack): `SYN` flood attacks. Schicke einfach `SYN`s, die dann aber wegen den Fast Open Daten an den HTTP server weitergereicht werden (mehr Arbeit). Reflektion ist einfach das nur mit spoofed IP Adressen, die dann die HTTP Daten von der Request bekommen. (Somit auch eine Art *amplification*)

Das zweite Problem kann mit Cookies (!TCP cookies nicht HTTP) behoben werden. Wir müssen einmal einen 3-WHS machen und bekommen dann einen Nachweis über das halten der betreffenden IP-Addresse. Das müssen wir bei Fast Open mitschicken. Der Cookie muss natürlich irgendwie sicher übertragen werden. Ebenfalls hängt er von der IP Addresse ab, bei einer geänderten IP Adresse gilt er deswegen nichtmehr.

Auch TCP Fast Open wird bei Bedarf über Socket Optionen (de)aktiviert.

Multipath (MPTCP)

Standard TCP nutzt nur ein Interface um eine Verbindung zu erstellen. Besonders für mobile Endgeräte ist ein wechseln zwischen Interfaces (z.B. LTE und WIFI) häufig, führt aber zu Verbungsabbrüchen.

Mittels Multipath TCP kann eine TCP Verbindung über mehrere Wege erreicht werden.

Hierfür erklären wir mit der `MP_CAPABLE` Option unsere Kompatibilität, die der Server ebenfalls aktiviert haben muss. Ab dann können mehrere `SUBFLOWS` existieren. Um einen neuen `SUBFLOW` zu der Verbindung hinzuzufügen muss eine neue TCP Verbindung zum gleichen Server aufgebaut werden, erneut mit der `MP_CAPABLE` **und** `JOIN` Optionen. Die vorhandene Verbindung wird dann durch die Server (oder Client Adresse jenachdem wer die neue Verbindung aufbaut) identifiziert.

Das gesamte läuft erneut transparent für den Benutzer der Sockets ab, heißt er bekommt davon nichts mit, denn für ihn sieht alles nach einer einzigen Verbindung aus.

Middleboxes

Middleboxes können fast alle Felder eines IP Pakets im Netz unvorhersehbar modifizieren, was die Implementierung neuer Protokolle, wie auch MPTCP erschwert. Dazu können sie Pakete mit unbekannten TCP Optionen oft einfach droppen. Somit können neue Funktionen entweder erst gar nicht aktiviert oder behindert werden.

Zudem könnten Middleboxes die nur einen Teil der MPTCP Verbindung sehen die Verbindung als gescheitert ansehen und daraufhin auflösen/blockieren, da sie einen Teil der Nachrichten erhalten.

Unteranderem um dieses Problem zu lösen verwendet MPTCP die Sequence Nummern von TCP jeweils nur für einen *Subflow*. Um aber über *Subflows* hinweg die Reihenfolge zu beizubehalten werden globale `DATA ACK`s und `DATA Seq. Num.` eingeführt und in den TCP Optionen mitübertragen.

3.4 Design Principles

Da sich Anforderungen an Protokolle mit der Zeit verändern können, sollten sie so dynamisch und erweiterbar wie möglich designed werden. Das lässt sich in die folgenden Stichpunkte zusammenfassen:
- Simplicity
- Modularity (teile das Protokoll in viele kleine Teile die jeweils eine Funktion erfüllen)
- Well-formedness (tut das was es soll, nicht mehr oder weniger, beachtet verfügbare Resourcen)
- Robustness (Fehlerfälle sind abgedeckt)
- Consistency (keine deadlocks)

Oft ist ein Schichtenmodell sinnvoll.

10 Rules of Design

1. Make sure that the problem is well designed
2. Define the service first
3. Design external functionality before the internal one
4. Keep it simple
5. Do not connect what is independent
6. Do not impose irrelevant restrictions (extendability)
7. Build a high-level prototype first and validate it
8. Implement the design, evaluate and optimize it
9. Check the equivalence of prototype and implementation
10. Don't skip rules 1-7

Internet Design Goals

Das primäre Ziel beim Internet Design ist es, die existierenden Netzwerke optimal auszunutzen. Dabei werden Gateways benötigt, um Netzwerke verschiedener Art zu verbinden.

Dabei gibt es einige sekundäre Ziele, die beachtet werden müssen:
- Internet Kommunikation muss bei Ausfällen von Netzwerken oder Gateways bestehen bleiben (survivability)
- Das Internet muss mehrere Typen von Communication Services unterstützen (z.B. Email (muss zuverlässig sein), file transfer (sollte schnell sein)), realisiert durch die Transportschicht
- Das Internet sollte mit möglichst vielen Arten von Netzwerken kompatibel sein
- Das Betreiben des Internets sollte möglichst Kostengünstig sein

Fehler beim Design des Internets:
- Staukontrolle musste im Nachhinein hinzugefügt werden
- Zu geringe Anzahl an IP Adressen (NAT erforderlich)
- Sicherheit musste durch Firewalls hinzugefügt werden, die das end-to-end Prinzip verletzen

Circuit Switching vs. Packet Switching

Bei Circuit Switching bekommt jede Sitzung einen eigenen (virtuellen) Kanal mit reservierten Resourcen. Dabei wird der Zustand der Verbindungen auf jedem Knoten entlang des Pfades gespeichert.

Positives (+)	Negatives (-)
• Garantien und Performance sind einfach • Distributed State (kann auch ein negativ Punkt sein)	• Session Setup muss gemacht werden • Ungenutzte Kapazitäten werden verschwendet

Im Gegensatz dazu werden bei Packet Switching Daten in Packets durch das Netzwerk gesendet, die sich die Resourcen des Netzwerks teilen. Der Zustand wird hier nur auf den Endhosts gespeichert.

Positives (+)	Negatives (-)
• Resourcen werden besser ausgenutzt. • Fehler in den Middleboxen und Routern einfacher zu beheben (einfach einen anderen Weg wählen) • Keine Sessions notwendig • Durch den Endhost Zustand sind Router einfach (fast) Plug and Play.	• Es kann zu Stau kommen (benötigt Congestioncontroll) • Performancegarantien sind schwieriger (aber machbar auf verschiedenen ISO/OSI Leveln)

Im Internet hat sich das Prinzip „dummes Netzwerk, Intelligenz an den Endsystemen“ durchgesetzt. Das beinhaltet die verwendung von Packet Switching und die Verwendung von zwei Transportprotokollen. Dabei wird ein Schichtenmodell verwendet, durch das der Rechenaufwand erhöht wird.

3.5 Design a Protcol

- Adressen: Entweder per Client (Device Adresse) oder Flow Adresse à la RTP. Der Adressraum sollte groß genug sein für die Zukunft
- Sequence Control: Der Sequencenumberraum sollte groß genug sein, so dass keine zwei gleichen seq. numbers simultan unterweges sind
- Flow Control: end-to-end (z.B. stop-and-wait oder sliding window) oder hop-by-hop (z.B. resource reservation). RTT sollte berücksichtigt werden
- Congestion Control: nicht vergessen!

Congestion Control

TCP hat ursprünglich fehlende ACK Nachrichten genutzt, um Informationen über die Auslastung einer Leitung zu sammeln. Ein modernerer Ansatz ist es, Explicit Congestion Notifications (ECNs) zu verwenden, mit denen das Netzwerk dem Sender Feedback zu der Stausituation sendet. Je nach Anwendung kann die Staukontrolle auch von der Anwendungsschicht übernommen werden (z.B. bei Video Streaming mit der Bitrate).

Error Control

Fehler die auftreten können:

- **Corruption**: Mittels checksums detectable (z.B. CRC)
- **Lost Packets**: Mittels sequence numbering oder timeouts detectable

Wir können bei Fehlern entweder automatisch das Packet retransmitten (ARQ) oder immer schon redundante Packete mitschicken und mittels CRC Korrekturen vornehmen.

1. Bei ARQ ist es egal, ob Packete falsch oder nicht ankommen, wir schicken sie immer einfach erneut. Hierbei gibt es auch Strategien:
   - **stop-and-wait**: Nur bei geringer RTT und hoher Fehlerrate
   - **Go-back-n**: Nur bei kleinem Buffer
   - **Selective repeat**: Sonst
2. Forward Error Correction kann z.B. so laufen: Für alle n Pakete erstelle ein XOR Paket von allen n Paketen $1 \oplus 2 \oplus ... \oplus n$. Dann kann eins der $n+1$ Pakete verloren gehen oder corrupted werden. Wir müssen allerdings auf alle $n+1$ Pakete warten um es wiederherzustellen.

Hamming Codes

Ist der minimale Code für die Korrektur von 1-bit Fehlern.

Algorithmus: Hamming Beispiel

Nachricht:

```
H 1101000
T 1110100
W 1110111
R 1110010
```

wird dann zu

```
1  2  3  4  5  6  7  8  9  10  11
---------------------------------
      1     1  0  1     0  0   0
      1     1  1  0     1  0   0
      1     1  1  0     1  1   1
      1     1  1  0     0  1   0
```

Nun müssen wir die Parity Bits berechnen (hier für das erste Byte):

$$c_1 = p_1 = p_3 \oplus p_5 \oplus p_7 \oplus p_9 \oplus p_{11} = 1 \oplus 1 \oplus 1 = 1$$

Die Positionen kommen durch die Binärschreibweise zustande: $3 = 1 + 2, 5 = 1 + 4, 7 = 1 + 2 + 4$,...

$$c_2 = p_2 = p_3 \oplus p_6 \oplus p_7 \oplus p_{10} \oplus p_{11} = 1 \oplus 1 = 0$$
$$c_3 = p_4 = p_5 \oplus p_6 \oplus p_7 = 1 \oplus 1 = 0$$
$$c_4 = p_8 = p_9 \oplus p_{10} \oplus p_{11} = 0$$

Insgesammt ergibt sich:

```
1  2  3  4  5  6  7  8  9  10  11
---------------------------------
1  0  1  0  1  0  1  0  0  0   0
1  0  1  0  1  1  0  1  1  0   0
0  0  1  0  1  1  0  1  1  1   1
0  1  1  0  1  1  0  1  0  1   0
```

Hamming Code korrigiert immer nur 1-bit Fehler. Bei mehrern Fehlern kann er teilweise falsche Korrekturen vornehmen.

BCH

Bose, Chaudhuri, Hocquenghem (BCH) funktioniert mit Blocks und existiert für jeden Integer mit $m \geq 3$ und $t < 2^{m-1}$ mit Eigenschaften:

- Blocklänge: $n = 2^m - 1$
- Paritätsbits: $n - k \leq m * t$
- min distance: $d_{\min} \geq 2 * t + 1$
- **Korrigiert Fehler**: t Fehler in einem Block der größe $2^m - 1$ bits.

Funktionieren tut dies mit generator polynomen und dem GF(p) Endlichen Körper.

Reed-Solomon Code

Reed-Solomon Codes sind nicht binäre BCH-codes. Wir nutzen hier ein (n,k) code: k-dimensionalen Untervektorraum aller n-Tupel in $\mathrm{GF}(q)$, wobei $q = p^n$ für irgendein $p \in \mathbb{P}, n \in \mathbb{N}$.

Error Tolerance

Generell werden beschädigte Pakete vom Link Layer verworfen. Da bei z.B. Audio Streaming die Payload im Vergleich zum Header klein ist, ist es nicht unwahrscheinlich, dass bei Übertragungsfehlern nur der Header beschädigt wurde.

Das Konzept „Refector“ ermöglicht es, dass auch fehlerhafte Pakete an die Anwendung weitergegeben werden (opt-in). Dies wird zum Beispiel in Voice over IP (VoIP) benutzt (evtl. mit FEC auf dem Payload).

Das Problem dabei ist, dass korrumpierte Header dazu führen können, dass nicht klar ist, zu welcher Anwendung das Paket gehört. Refector löst dies, indem es die Pakete der Anwendung zuordnet, zu der sie am wahrscheinlichsten gehören (mit BCH code).

3.6 Encoding

Videos und Bilder können wir natürlich lossy komprimieren, andere Daten nicht umbedingt. Für lossless compression gibt es zwei Kategorien:

- Redundancy removal: Ausnutzen von Wiederholungen.
- Statistische Analyse: Minimale Länge eine bestimmten Sequenz.

Sidenote: Üb das und Hamming

Das sind einer der weniger Schemaaufgaben die es gibt, sie kommen also… Wir gehen hier durch alle als Übung durch

Run-length Encoding (RLE)

Nehme ein nicht benutztes Symbol (Flag) und gebe die Anzahl der Wiederholten Symbole an. Als optimierung kann ein offset z.B. 4 genutzt werden, da bei Längen unter 4 keine komprimierung erfolgt.

Algorithmus: Run-length Encoding

HTWWWWWWWRRRRRRAACHEEENNNN

$\Longrightarrow$

$\#W = 7 - 4 = 3$
$\#R = 6 - 4 = 2$
$\#E = 4 - 4 = 0$

HTW!3R!2AACHEEEN!0

Lempel-Ziv-Welch Coding (dictionary)

Die Idee bei *Differential Encoding* ist, dass Zeichen in einer Zeichenkette von den vorherigen abgängen. Anstatt also die Werte selbst zu speichern, werden sie als Differenz zu einem früher vorgekommenen Wert gespeichert.

Beim LZW Coding wird ein dictionary von der Sequenz aufgebaut, sodass wenn Werte aus diesem dictionary vorkommen, sie durch einfache keys ersetzt werden können.

Algorithmus: Zempel-Ziv-Welch

- Konstruiere ein dictionary mit allen Symbolen, die im Input vorkommen
- Von der aktuellen Position im Input String aus
  - Encode die darauffolgenden Symbole mit dem längsten match der dictionary Einträge
  - Erstelle einen neuen dictionary Eintrag aus dem ausgewählten Eintrag zusammen mit dem nächsten Symbol
  - Gehe zum nächsten noch nicht verarbeiteten Symbol und wiederhole die Schritte

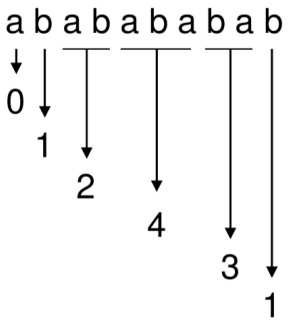
Beispiel

- **Given:**
  - Symbols *a*, *b*
  - Input *a b a b a b a b a b*

Dictionary:

0	a
1	b
2	ab
3	ba
4	aba
5	abab
6	bab

Encoding:



Information & Entropie

$$I = \log_2\left(\frac{1}{p}\right) = -\log_2(p)$$

$$H = -\sum_{i=1}^{N} p_i \log_2(p_i)$$

Dabei ist N die Anzahl an Symbolen und p_i die Häufigkeit des Symbols i.

Entropie kann genutzt werden, um die minimale durchschnittliche Anzahl an Bits pro Symbol um eine Symbolsequenz zu encoden zu ermitteln.

Die Idee dabei ist, häufige Symbole mit kürzeren Bitfolgen zu encoden (Huffman Code).

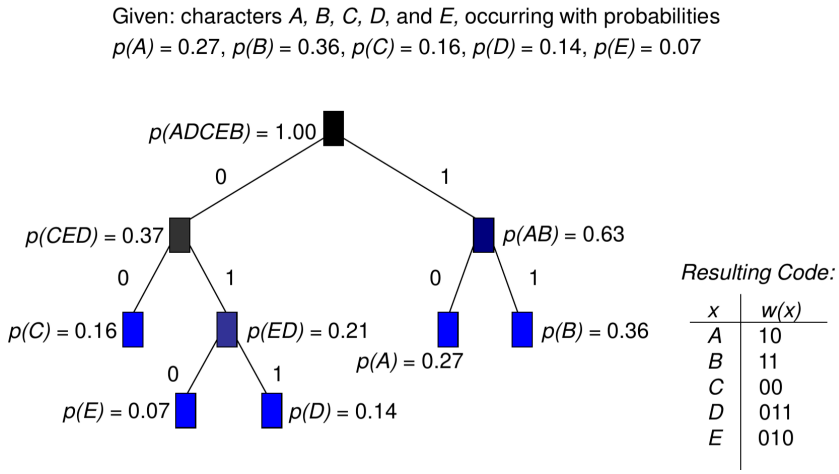
Huffman Code

Konstruiere eine Binärbaum, mit Symbolen als Blätter. Die Häufigkeit der Blätter muss dabei bekannt sein.

Algorithmus: Huffman Code

- Zähle die Häufigkeit für jedes Symbol und füge sie in eine Menge
- Entferne die zwei Elemente mit der niedrigsten Häufigkeit
- Füge diese beiden Elemente in einen Baum ein (links niedrigere Häufigkeit, richts höhere Häufigkeit)
- Füge den Baum als neues Element mit der Menge hinzu (Häufigkeit = Häufigkeit links + Häufigkeit rechts)
- Wiederhole die Schritte, bis nur noch ein Element (der Baum) übrig ist
- Markiere Kanten nach links mit „0“ und Kanten nach rechts mit „1“

Beispiel



Alternativ kann Arithmetic Encoding verwendet werden, was in manchen Fällen kürzere Codes als Huffman erzeugt, wo der Datenstream allerdings nur als Ganzes decoded werden kann.

3.7 Design Aspekte

Eine Frage ist: **Verbindung oder keine Verbindung?**

Wir könnten auch die Status Informationen, Konfigurationen und die Datenübertragung über unterschiedliche Protokolle/Verbindungen machen.

- *in-band* signaling: Control & Data über die gleiche Verbindung (z.B. TCP, HTTP).
- *out-of-band* signaling: Unterschiedliche Verbindungen (z.B. FTP, SIP + RTP).

Vorteile des *in-band* signaling

Positives (+)	Negatives (-)
• Einheitliches und vereinfachtes Management von Verbindungen	• Kann Latez erhöhen. • Erschwert QoS garantien und eventuell niedrigere Durchsatzrate

Auch beim Zustand und bei der Kontrolle gibt es optionen:

- *Hard-State*: Zustand wird explizit verändert (z.B. TCP via `FIN` Nachrichten)
- *Soft-State*: Zustand wird regelmäßig erneuert/refreshed.
- *Centralized Control*: Client-Server
- *Decentralized Control*: Z.B. P2P.

3.8 Sonstiges

- Randomisierung kann hilfreich. Z.B. Initalisierung der Seq. Nummern oder wait time bei Kollisionen.
- Indirections: Nicht per se in Protokollen, aber häufig in networked systems. Wir referenzieren eine Art Middleman/Router die uns dann den richtigen Content gibt. Wird zum Beispiel für load balancing verwendet.

3.9 Wissenswertes zu den Bespielen

Versionsrecap HTTP:

1. Version 0.9: Nur einzelne ASCII Anfragen und ASCII **HTML** antworten. Auch nur ein Stream das mit CRLF endet… :/
2. Version 1.0: Einführung der HTTP Methoden, Status Codes `GET`,`POST`, etc…, Einführung von Multiline Nachrichten und andere Antworten außer HTML. Einführung der HTTP Header
3. Version 1.1: Persistent connection: Wie wäre es nicht jedes mal eine TCP Verbindung aufbauen zu müssen. Viele weitere Funktionen: Encodings, Cookies, etc. Optimierungen für HTTP 1.1 Client kann mehrere TCP Verbindungen nutzten um gleichzeitig Anfragen zu machen (das ist immer noch nicht so parallel).
4. Version 2.0: Vom ASCII protocol zu *binary*. Einführung von Connection Streams, Messages und Frames.
   - Ein *Frame* ist die kleinste Form der Kommunikation in HTTP/2
   - Eine *Message* ist eine Sequenz von *Frames*.
   - Ein *Stream* ist ein bidirektionaler Nachrichtenverkehr
   - Eine *Connection* kann mehrere Streams beinhalten. Aber **nur eine** *Connection* zwischen Client und Server.

   Eine Nachricht kann zum Beispiel dann aus einem Header *Frame* und einem DATA *Frame* bestehen. Die Stream ID identifiziert den Stream. Server initialisierte Streams **gerade**, Client initialisierte **ungerade**.

   Es gibt auch Flow Control aufbauend auf dem TCP flow control. Hiermit kann aber besseres fine-tuning und Applikationsspezifisches flow controlling erreicht werden. Streams können unterschiedlich gewichtet sein und *dependencies* haben (somit können beim scrollen die wichtigen Sachen priorisiert werden). Der Server kann auch *Server push* ohne explizite Anfrage des Clients, aber mit Vorwarnung.

   Header Frames können mit `HPACK` comprimiert werden. Im Hintergrund nutzt es Huffman codes und vordefinierten statischen (oft verwendeten Begriffe) und dynamischen Tabellen.

   Insgesamt ein ziemlich gutes Protokoll.
5. Version 3.0: Praktisch HTTP/2 over QUIC statt TCP. Kein HPACK sondern QPACK um HoL blocking zu vermeiden.

Sidenote: HTWR

Ich gebe mein Bestes und will alle tollen neuen Protokolle nutzen. https://htwr-aachen.de unterstützt IPv6 und HTTP/3

Tolles QUIC. Es baut auf vielen Extensions zu TCP auf:
- QUIC nutzt Fast Open
- QUIC nutzt 0-RTT TLS 1.3 (TLS ist in QUIC drin).

- QUIC connections sind nicht an IPs gebunden um multipath/mobilität zu erlauben.
- Praktisch alle insights und developments in QUIC implemtiert oder weiter verbessert (z.B. TCP NewReno *congestion control*).
- Selective ACKs, *Retransmission* können von regulären Nachrichten unterschieden werden.

Explicit Correction for Delayed ACKS, Forward Error Correction (aktuell nicht aktiviert).

4 Kernel stuff

4.1 101

Grundlagen:
- User Space: Außerhalb des Kernels. Interaktionen über system calls, Speicherzugriffe sind *restricted*.
- Kernel Space: Du willst alles überschreiben? ok

Hardware Sachen werden meistens mit Interrupts oder Polling gemacht.

Hardware Mechanismen:
- Timer interrupts: Kernel bekommt garantiert regelmäßig Kontrolle zurück
- Memory protection: Segmentation und Paging (Virtual Adress Spaces etc..)
- Dual Mode: Kernel Prozesse können auch *non-previliged* sein, User Prozesse aber nicht *previliged*.

4.2 Kernel Networking

Situation: ein Packet kommt am Network Interface Controller (NIC) an.

1. Packet wird per Direct Memory Access (DMA) in den RAM kopiert.
2. Issue Interrupt Request (IRQ). Informiere das System über das neue Packet

Interrupts

Interrupts werden aufgeteilt in zwei Hälften/Teile:
1. *Top Half*: Der wirkliche Interrupt. möglichst wenig hier tun, da das OS und Prozesse interrupted sind. Während des Top Halfs sind meistens Interrupts deaktiviert. Um das tatsächliche Handling zu betreiben rufe einen Handler außerhalb des Interrupts auf.
2. Bottom Half: Mache tatsächlich was.

Es gibt mehrere möglichkeiten den Bottom-Half zu schedulen:
1. SoftIRQs (max. 32): Cannot be interrupted but can run on multiple CPUs.
2. Tasklets: Cannot be interrupted. Built on top of SoftIRQs. (easier to use)
3. Work Queues (Kernel Threads): Can create queues and or threads. Can sleep.

	SoftIRQ	Tasklets	Work Queues
Execution Context	runs in interrupt context	runs in interrupt context	runs in process context
Reentrancy	Can run simultaneously on multiple CPUs	Cannot run simultaneously. Different CPUs can run different Tasklets	Can run simultaneously on multiple CPUs
Sleep/Preemption	Cannot	Cannot	Can
Ease of use	Requires locking and deadlock prevention	Possible easier to use	Possibly easier to use

4.3 New API (*NAPI*)

Wie funktioniert das jetzt wirklich? Um die Nachteile des Pollings zu umgehen, aktivieren wir polling immer erst dann, wenn die Anzahl der eintreffenden Pakete einen bestimmten Schwellenwert überschreitet. Wir deaktivieren es entweder nach einer kurzen Zeit (ein paar ms) oder wenn keine zu verarbeitenden Pakete mehr da sind.

4.4 Arbeiten mit Netzwerkdaten

Das wichtigeste hier ist der *Socket Buffer* (mbuffs). Das ist die Datenstruktur auf der gearbeitet wird.

Teilkomponenten sind ein *header* (Metainformationen) und ein Speicherort für das Paket.
- `len`: total bytes im Paket
- `data_len`: bytes der tatsächlichen Daten
- ...

Wir wenden das zero-copy System an. Hierfür haben wir Pointer die auf die einzelnen Regionen zeigen:
- `head`: Zeigt zum Anfang des Speichers
- `data`: Anfang des Pakets
- `tail`: Ende des Pakets
- `end`: Zeigt zum Ende des Speichers

Locking
- *semaphores*: Falls eine kritischer Abschnitt *gelocked* ist, wird der wartende Prozess bei *semaphores* Schlafen gelegt.
- *spin_lock*: Falls eine kritischer Abschnitt *gelocked* ist, prüft der wartende Prozess bei *spin_locks* durchgehend den Lock. Er verbraucht also CPU Zeit. Dafür können *spin_locks* auch von (Prozess)-Strukturen genutzt werden, die nicht Schlafen können (z.B. SoftIRQs).

5 Testing

- Defensive Programmierung
- Verifikation (GIESL *hust*)
- Regression (Regression Testing bedeuted testen von funktionen die bereits implementiert sind. Gucken das Sie weiterhin funktionieren)
- Reactive Testing (versuche bekannte Fehler zu beheben, Input der zum Fehler führt ist bekannt)
  - Debugging: `gdb`, `valgrind` can also find memory leaks and has many sub-tools

  `memcheck`, `helgrind`, `cachegrind`... Usages:

  ```
  # gdb
  > gdb ./binary

  # valgrind
  > valgrind --tool=memcheck (--leak-check=full) ./binary
  ```
- Proactive Testing (versuche bisher unbekannte Fehler zu finden)
  - Assertions:
    - Assertions: Läuft via Compiler Direktiven. Fail wenn eine expression fehl schlägt
    - Contracts: Spezifiziere das genaue input schema via assertions.
    - Mit static assertions können auch bei compiletime Überprüfungen gemacht werden
    - Unit Testing: Schreibe extra Funktionen mit assertions, die eine „unit“ testen
  - *box Testing, Fuzzing: Unit-Testing, Black-Box Testing (Only test API), Whitebox, Coverage-guided Fuzzing

- Symbolic Execution, KLEE: Erkundet alle Pfade eines Programs. Folge Branches unabhängig von einander. Bei Dynamic Symbolic Execution: forke bei branch.
  Concolic Execution bei zu vielen Zuständen. Bearbeite nur einen Pfad nacheinander.
  Static Symbolic Execution: Untersuche das Programm als logische Formel.

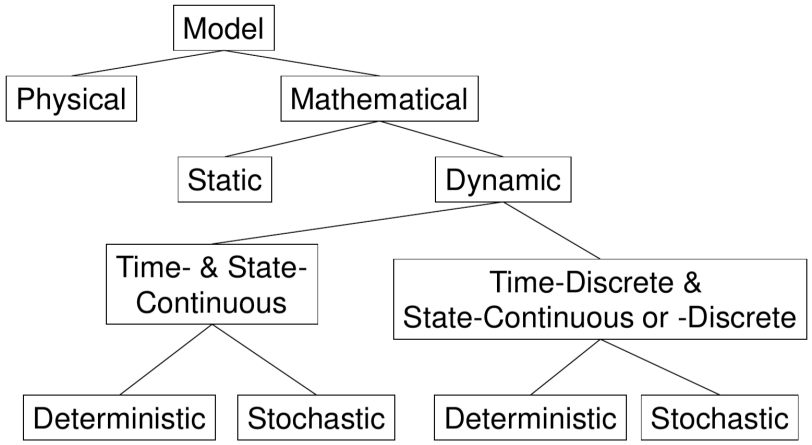
6 Simulation

Nachdem man ein Protokoll designed hat, sollte man es auf jeden Fall ausführlich testen, um die Performance beurteilen zu können. Da es schwierig ist, an große Netzwerke mit vielen Hosts zu kommen, um in der Praxis zu testen, ist die Simulation solcher Netze eine gute Alternative.

6.1 Allgemeines

Ein System besteht aus Komponenten die miteinander interagieren. Dabei hat das System einen Zustand der sich meistens mit der Zeit verändert, was entweder *kontinuierlich* oder *diskret* passieren kann.

6.2 Arten von Modellen



6.3 Discrete Event Simulation (DES)

- ist zeitdiskret und stochastisch (unten rechts)
- hat viele Anwendungsfälle
  - Netzwerke
  - Schaltkreise
  - Schlachtfelder
  - ...

Stochastisch bedeutet, dass die Simulation nicht deterministisch ist, also die gleichen Startbedingungen zu verschiedenen Abläufen führen können, da z.B. ping oder packet loss zufällig vorkommen.

Von der zu simulierenden Situation müssen einige Sachen festgestellt werden:
- die Komponenten, die Interaktionen durchführen (components)
- den Zustand des Systems, der aus den Zuständen der Teilsysteme besteht (state)
- Parameter, die definieren, wie sich der Zustand mit der Zeit ändert (parameters)

Bei DES wird davon ausgeganen, dass es diskrete Events gibt, bei denen sich der Zustand des Systems ändert und dass es zwischen diesen Events keine Zustandsveränderungen gibt. Dadurch kann man in der Simulation immer direkt zum Zeitpunkt des nächsten Events springen (next-event algorithms).

Für die Randomness brauch man noch einen Random Number Generator (RNG), z.B. Linear Congruential Generator (LCG):

$$x_n = (ax_{n-1} + c) \bmod m$$

Die Zeit innerhalb der Simulation ist unabhängig von der außerhalb und kann mit beliebigen Einheiten notiert werden (typischerweise mit Sekunden nach Simulationsstart). Mögliche Bezeichnungen für die verschiedenen Zeiten sind z.B. „simulated time“ (innerhalb) und „wall-clock time“ (außerhalb).

Generell sollte man bei der Simulation das *Framework* vom *Modell* getrennt halten.

Framework Das, worauf das Modell nachher läuft, hat eine API für Modelle und kann optional noch Features wie Statistiken oder ein GUI zur Verfügung stellen.

Modell Implemenetiert die zu simulierenden Prozesse und läuft dann auf einem Framework.

6.4 Simulation Frameworks

Zwei weit verbreitete Simulation Frameworks sind *OMNeT++* und *ns-3*.

OMNeT++
Objective Modular Network Testbed in C++ kann für jede beliebige Discrete Event Simulation verwendet werden, z.B. Modelierung von Telefonnetzen, Protokollen oder Harware Systemen. Dabei sind Framework und Modell sowie die Definition der Topologie und die eigentliche Implementation klar getrennt. Allerdings nicht so gut geeignet für Netzwerksimulationen, da die Umgebung gegenüber echter Hardware starke Abstraktionen hat.

ns-3
Die Implementation von ns-3 folgt dem ISO/OSI Modell und der Netzwerkarchitektur von Linux. Dadurch ist es auch möglich, dass die Simualtion mit echter Hardware oder VM Hosts außerhalb der Simulation interagiert. Ist also sehr gut geeignet, um ein neues Protokoll in großen Netzwerken zu testen.

6.5 Parallel Discrete Event Simulation

Um DES zu beschleunigen, ist es sinnvoll, die Simulation zu parallelisieren. Das geht entweder, indem die Event Handler mit z.B. OpenMP oder CUDA parallelisiert werden oder mehrere Event Handler parallel auf mehreren Threads oder mehreren CPUs ausgeführt werden. Dabei ist es wichtig zu beachten, welche Events voneinander abhängig sind, da das Ergebnis der Simulation sonst verfälscht werden könnte. Eine Möglichkeit dieser Synchronisation ist mit *Space-parallel Simulation*.

Space-parallel Simulation
Es gibt logische Prozesse (LP), die jeweils einen Teil der Simulation ausführen. Jeder LP hat seine Eigene Clock und event list. Die LPs tauschen untereinander Nachrichten zur Synchronisierung aus, wodurch eine korrekte Ausführung sichergestellt wird. Die LPs sollten so gewählt werden, dass die nötige Kommunikation zwischen ihnen minimal ist.

Bei Space-parallel Simulation gibt es zwei Arten von Algorithmen:
- Konservative Algorithmen, die streng auf die richtige Reihenfolge und Abhängigkeit von Events achtet
- Optimistische Algorithmen, die Events einfach immer ausführt. Die dadurch resultierenden Kausalitätsfehler werden dann erkannt und beseitigt. Dies führt of insgesamt zu einer besseren Performance.

6.6 Performance Evaluation

1. Problem formulation and system definition

2. Choice of metrics, factors and levels
3. Data collection and modeling
4. Choice of simulation environment & model implementation
5. Verification
6. Validation and sensitivity analysis
7. Experimentation, analysis and presentation

Die gesammelten Daten können dann z.B. mit arithmetischem Mittel oder empirischer Varianz aufbereitet werden.

7 Measurements

Messungen sind nützlich für die Validierung in Simulationen. Außerdem können Simulationen ganz vermieden werden, wenn wir existierende Systeme vermessen können.

In komplexen Systemen wie dem Internet ist das nicht so leicht.

Mögliche Gründe für Messungen sind unter Anderem:
- Wie entwickelt sich das Internet?
- Wie werden bestimmte Protokolle adaptiert?
- Wie gut ist die Performance von meinem Protokoll in einem bestimmten Netzwerk, bzw. wie verhält es sich unter bestimmten Bedingungen

(ähnlich zu Simulationen, aber jetzt bare metal)

7.1 Active Measurements

Sende selbst Traffic zum Messen los. Das ist zum Beispiel sinnvoll, um Situationen zu untersuchen, die sonst nur selten/nie auftreten. Der Nachteil ist, dass das untersuchte System dabei beeinflusst wird.

Ein einfaches Beispiel von Active Measurements ist das Tool `ping`, mit dem man die RTT zu einem Host messen kann.

7.2 Passive Measurements

Sammle existierenden Traffic passiv. Dadurch können nützliche Einblicke in das Netzwerk gewonnen werden. Allerdings ist dieser Zugriff in den meisten Netzwerken sehr schwer zu erhalten.

7.3 Tools

Für beide Arten von Measurements bieten sich die folgenden Tools an
- `Ipsumdump`, um dump files zusammenzufassen
- `Tcpdump`, um einzelne traces zu erfassen
- `Wireshark`, kann auch traces erfassen und diese visualisieren
- `Tshark`, ähnlich wie `Wireshark`, aber im Terminal
- `Bro`, Protokollanalysen etc.

Beim Erfassen von Paketen sollten auch immer Metadaten, wie das genutzte Tool und dessen Version gespeichert werden.

7.4 Practical Challenges
- Wenn man den gesamten Traffic eines Netzwerkes speichert, generiert man sehr schnell sehr große Datenmengen, die man speichern und nachher verarbeiten muss
- An viele Daten kommt man nur schwer dran, gerade an die interessanten
- Wenn man nur einen Teil der Daten hat, ist es schwierig, gute Aussagen zu treffen

- Filtering/Sampling ist meistens notwendig, damit die generierten Datenmengen im Rahmen bleiben (z.B. nur bestimmte IP Addressen, Protokolle, Ports, etc. oder nur Teile von Paketen)
- Realer Internet Traffic ist voll von Anomalien, unerwartetem Verhalten und Hosts, die sich nicht an Standards halten → Daten müssen vor der Verarbeitung aufbereitet werden