

Rechnernetze & Verteilte Systeme

Ludwig-Maximilians-Universität München

Sommersemester 2019

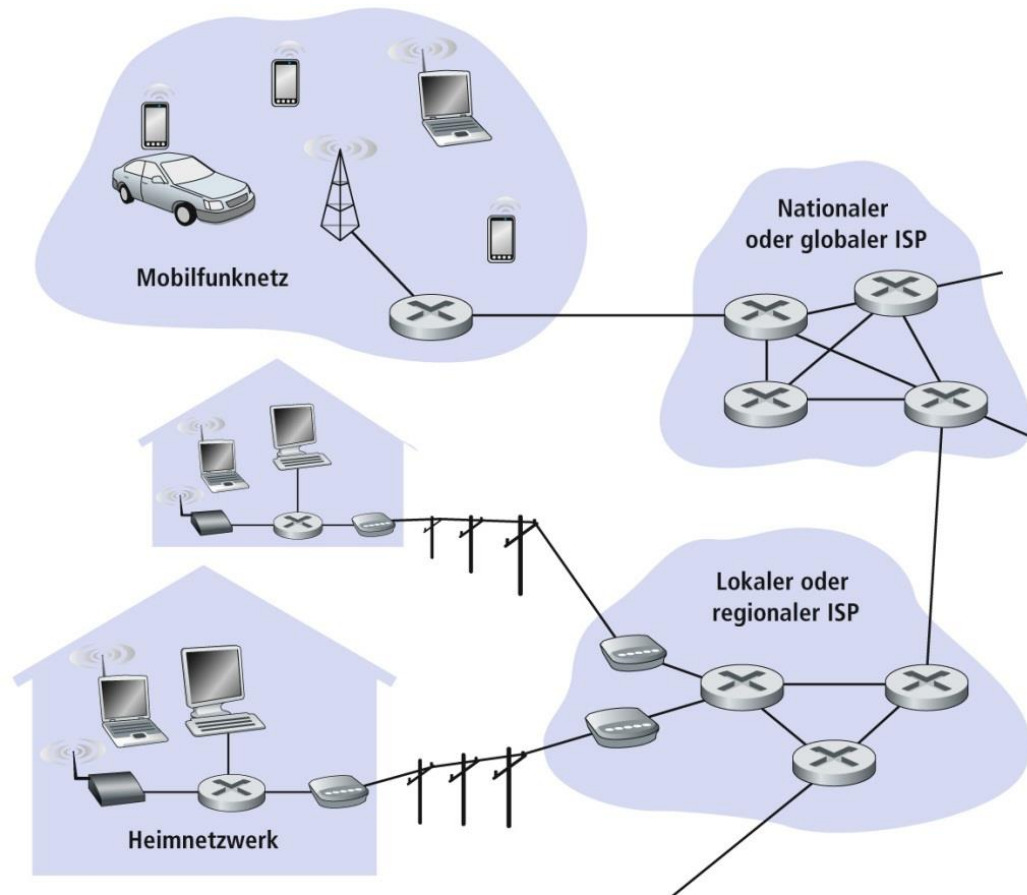
Prof. Dr. D. Kranzlmüller

<http://www.nm.ifi.lmu.de/rn>



Fehlererkennung und -Korrektur

Komplexität begünstigt Fehler



Komplexität begünstigt Fehler

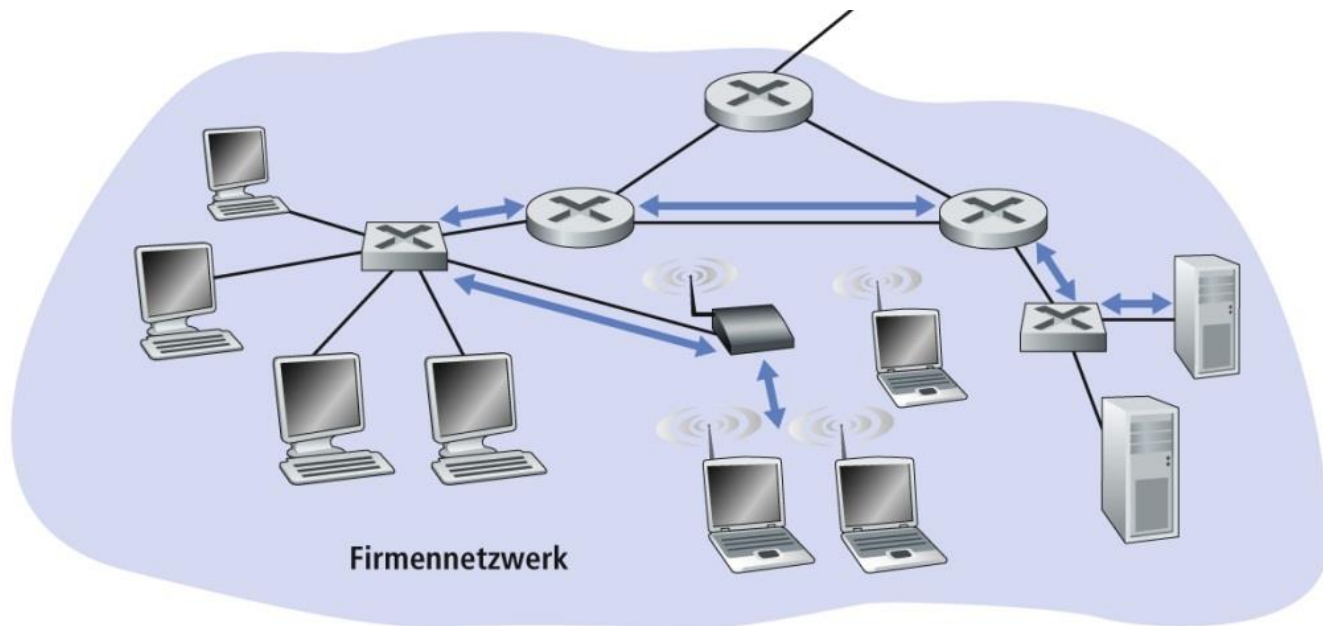
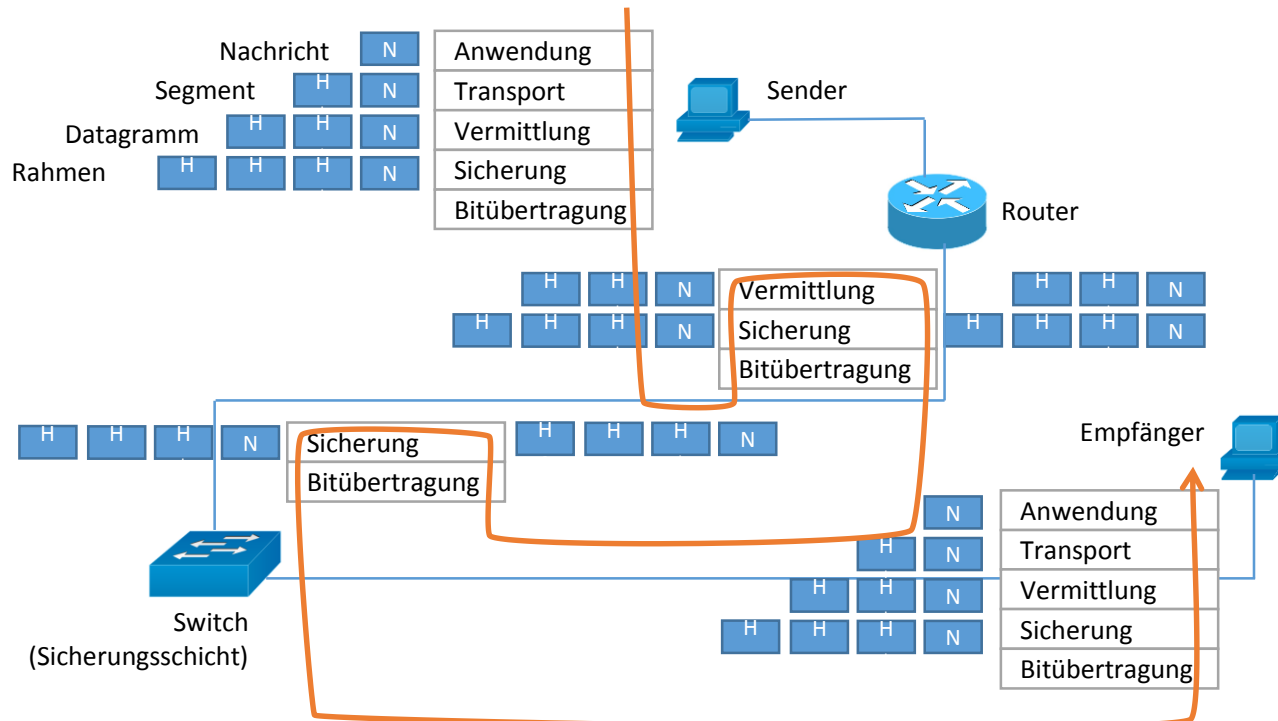


Abbildung 5.1: Sechs Hops auf der Sicherungsschicht zwischen drahtlosem Host und Server.

Protokollschichtung



Übersicht

- Klassifikation von Fehler
 - Bitflips (einzelne fehlerhafte Bits)
 - Burstfehler: Fehler treten meist gebündelt auf.
- Allgemeiner Ansatz: Beifügen von **Checkbits** zur
 - Fehlererkennung bzw.
 - Fehlerkorrektur



- Von besonderem Interesse (zur Beurteilung von Verfahren):
 - Overhead
 - Restfehlerrate (was für Fehler werden nicht erkannt?)
 - Komplexität der Berechnung (Effizienz der Implementierungen)

Einordnung

- Paritätsbits
- Checksummen
- Selbstkorrigierende Codes

→ Betrifft alle Schichten des Internet-Modells.

Paritätsbits

Paritätsprüfung

- **Einfachste Form: Paritätsbits**
- Es wird ein einzelnes Paritätsbit an die Daten angehängt.<
- Anhängen eines Bits zur Charakterisierung der binären Quersumme
 - Gerade Parität: Paritätsbit führt zur gerader Anzahl an Einsen.
 - Ungerade Parität: Ungerade Anzahl an Einsen

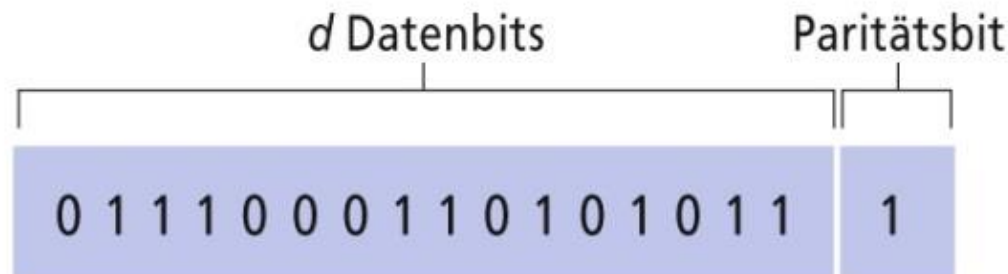


Abbildung 5.4: Gerade Ein-Bit-Parität.

Paritätsprüfung

- Paritätsbit erkennt alle Fehler bei denen eine ungerade Anzahl an Bits *zerstört* wurde. → **50% aller Fehler**
- Leicht zu berechnen, geringer Overhead
- **Schwäche:** In der Realität sind bei hohen Bitraten (hoher Durchsatz) häufig mehrere Bits betroffen (Burst Fehler).

Paritätsmatrix

	Zeilenparität →			
Spaltenparität ↓	$d_{1,1}$	...	$d_{1,j}$	$d_{1,j+1}$
	$d_{2,1}$	...	$d_{2,j}$	$d_{2,j+1}$
	...	...	...	...
	$d_{i,1}$	...	$d_{i,j}$	$d_{i,j+1}$
	$d_{i+1,1}$	...	$d_{i+1,j}$	$d_{i+1,j+1}$

Keine Fehler

1	0	1	0	1		1
1	1	1	1	0		0
0	1	1	1	0		1
0	0	1	0	1		0

Korrigierbarer Einzelbitfehler

1	0	1	0	1		1
1	0	1	1	0		0
0	1	1	1	0		1
0	0	1	0	1		0

Paritätsfehler

Paritätsfehler

Abbildung 5.5: Gerade zweidimensionale Parität.

Beurteilung Paritätsmatrix

- Overhead: Pro $n \cdot m$ Nutzdatenbits müssen $n+m+1$ Paritätsbits berechnet und mitübertragen werden.
- Gibt es in einer Zeile (oder Spalte) eine gerade Anzahl Bitfehler, so können diese in der Regel immer noch anhand der Parität der betroffenen Spalte (bzw. Zeile) erkannt werden.
- Restfehlerrate ist wesentlich geringer als bei einfacher Paritätsprüfung.

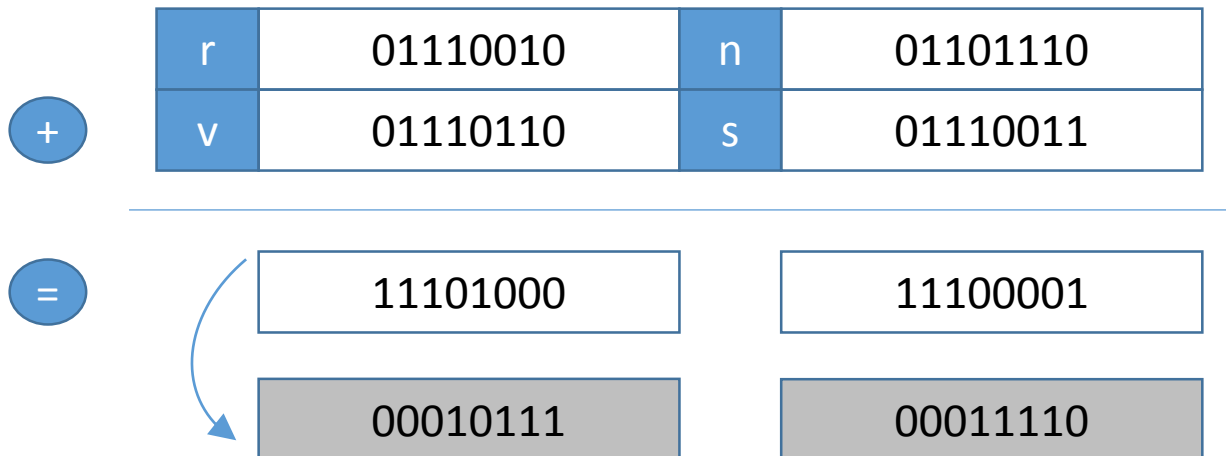
Checksummen

Internet-Checksumme

- Wird in UDP zur Fehlererkennung in Datagrammen verwendet (RFC 1071)
- Berechnungsvorschrift (Sender)
 1. Summiere alle k-bit Codewörter
 2. Berechne 1er Komplement (Invertieren aller Bits)
- Prüfungsvorschrift (Empfänger)
 1. Summiere alle k-bit Codewörter + Checksumme
 2. Ergebnis darf ausschließlich Einsen beinhalten

Internet Checksum: Beispiel

- Übertrage Nachricht „rnvs“
- UDP-Felder bestehen immer aus 16-Bit Wörter (2 Bytes)



Beurteilung Internet-Checksumme

- Einfach zu berechnen
- Geringer Overhead: Nur 16 Bit
- Schwäche: Allgemein dennoch schwächerer Schutz als andere Checksummen Varianten.

Zyklische Redundanzprüfung (CRC)

- Betrachte d Datenbits als Koeffizienten für ein Polynom mit d Terme.
- Sender und Empfänger vereinbaren ein $r+1$ bit-pattern, was als Generatorpolynom G aufgefasst wird.
- Der Sender berechnet r zusätzliche Bits (**R**) als Prüfsumme, sodass die Kombination aus d Datenbits und r teilbar durch G ist.
- Empfänger teilt empfangene Nachricht ebenfalls durch G .
- **Wenn Rest = 0, ist die Nachricht fehlerfrei übertragen.**

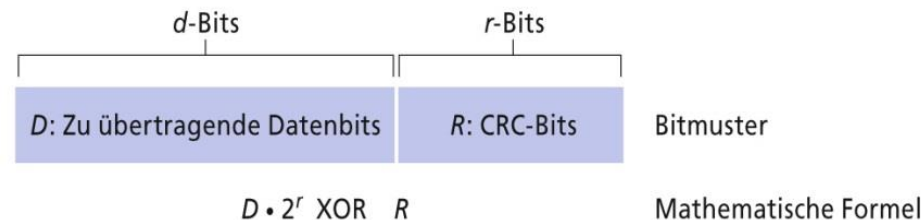


Abbildung 5.6: CRC-Codes.

Berechnung der CRC Quersumme

- Sei $G(x)$ ein Generator-Polynom mit Grad r
 - *Beispiel:* $x^5 + x^4 + x^0$ (Grad 5)
- Füge r Null-Bits an die Nachricht M (d Datenbits)
- Wähle R , sodass $D * 2^r \text{ XOR } R = nG$

$$\rightarrow R = \text{remainder} \frac{D * 2^r}{G}$$

Beispiel: CRC

- Generator-Polynom $G = x^3 + 1$
→ $G = 1001, r = 3$

- Daten: 101110
→ Anhängen von r Nullen.

Zu Übertragende Nachricht:
101110011

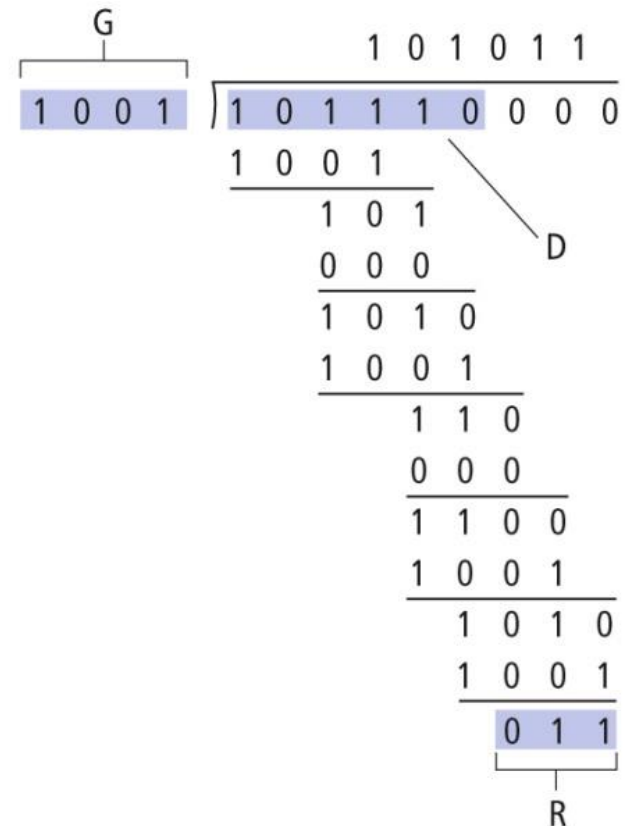


Abbildung 5.7: Beispiel einer CRC-Berechnung.

Anforderungen an Generatorpolynome

- Alle 1-Bit Fehler, falls $G(X)$ mindestens zwei Terme ungleich 0 hat.
- Alle 2-Bit Fehler, falls $G(X)$ einen Faktor mit drei oder mehr Summanden hat
- Jede ungerade Anzahl an Fehler, solange $G(X)$ einen Faktor $(X+1)$ hat
- Jeden zusammenhängenden Fehler (Fehlerburst), dessen Länge nicht größer als r ist.
- Einen Teil der Bursts mit $r+1$ Bits
- Einen Teil der Bursts mit mehr als $r+1$ Bits

CRC Standards

- Wähle G so, dass potenzielle Fehler möglichst kein Vielfaches von G sind.
- Eine Auswahl an standardisierten Gs

CRC-16	$X^{16} + X^{15} + X^2 + X^1$
CRC-CCITT	$X^{16} + X^{12} + X^5 + X^1$
CRC-32	$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$
Bluetooth	$X^8 + X^2 + X + 1$

- CRC-32
 - Einsatz in Ethernet seit 1980
 - Erkennt alle Fehler mit ungerade Anzahl an Bits
 - Erkennt alle Bursts der Länge ≤ 32

Selbstkorrigierende Codes

Selbstkorrigierende Codes

- Idee: Codetabelle wird so dünn besetzt, dass Verfälschung zu unzulässiger Codierung führt. Die Fehlerlokalisierung ermöglicht Korrektur.
- Hammingabstand **h** von Codewörtern: Anzahl der unterschiedlichen Bits zwischen zwei Codewörter
→ Logisches „exclusiv order“ (XOR)
- Hammingabstand **h** von Codes: Minimum der Distanzen aller Paare von Codewörtern.

Beispiel: Code-Matrix

	00000	00111	11001	11110
00000	0	3	3	4
00111	3	0	4	3
11001	3	4	0	3
11110	4	3	3	0

- Hamming Distanz $h = 3$
 - Korrigiert alle 1-Bit Fehler.
 - Erkennt alle 2-Bit Fehler.

Relevante Parameter

- Hammingabstand eines Code: h
- Erkennung von h Fehler: $h+1$ Code
 - h Fehler (Bitflips) ergeben kein anderes valides Codewort
- Korrektur von h Fehler: $2h+1$ Code
 - Bei h Fehler ist das ursprüngliche Codewort immer noch näher am Fehler als jedes andere Codewort.

Erkennung von 1 Bit Fehler

- Betrachte einen Code mit $d + r$ Bits (d = Daten, r = Redundanz)
- Minimaler Hamming-Abstand $h = 2$ erforderlich
- es gibt 2^d Nutzzeichen sowie 2^n Bitmuster
- jedes Nutzzeichen hat n Nachbarn mit Distanz 1
→ benötigt daher $n+1$ Bitmuster
- Daher gilt

$$(n + 1)2^d \leq 2^n$$

$$(d + r + 1) \leq 2^r$$

Beispiel: ASCII

- für ASCII gilt $d = 7 \rightarrow r = 4$
- Es sind 11 Bits erforderlich, um das Nutzzeichen + Redundanz zu repräsentieren
- Daher: 36% Overhead nötig um 1-Bit Fehler zu erkennen.
 - Vergleich Paritätsbit: 12,5% Overhead.