Technische Universität Ilmenau 30. Mai 2024

Vorlesung

Advanced Networking Technologies

Dr. Michael Roßberg

NEXTEX

Vorlesungsmitschrift von
Adrian Schollmeyer

Inhaltsverzeichnis

1. Routers and Switches

Router sind zuständig dafür, Pakete anhand der Informationen im IP-Header (i. d. R.) weiterzuleiten. Das passiert anhand der Forwarding-Tabelle (berechnet aus der Routing-Tabelle), die eindeutig einen Next Hop für eine Zieladresse festlegt. Router werden üblicherweise in *Points of Presence* (POPs) (z. B. DE-CIX) zusammengeschlossen. Das Internet ist also kein gut vermaschtes Netz, sondern die Vermaschung konzentriert sich eher auf wenige POPs.

1.1. Why we Need Faster Routers

Router werden i. d .R. für größe Bandbreiten ausgelegt, da diese sonst leich zum Bottleneck werden können. Schnelle Router sind wichtig, um Kapazitäten, Kosten, Größe und Stromverbrauch am POP zu senken. Entscheidend sind die Port-Kosten. Je weniger Ports benutzt werden sollen, umso schneller muss der Router werden. In POPs mit großen Routern können Pakete innerhalb des POPs mit deutlich weniger Interconnections weitergeleitet werden, während bei kleineren Routern deutlich mehr Verbindungen nötig sind (um alles miteinander zu verbinden).

Zur Steigerung von Übertragungsgeschwindigkeiten ist es bspw. auch möglich, die Wandlung der Daten in elektrische Signale an Routern für einzelne Farben in einem WDM auszulassen und stattdessen über ein Prisma die Farbe direkt an die passende Zielfaser weiterzuleiten. Dies spart teure Umwandlungen und erhöht den Durchsatz.

1.2. Why Making Fast Routers is Difficult

Moore's Law ist für CPUs nicht mehr gültig und wurde für Speicher nie erreicht. Weder Kapazität, Bandbreite noch Zugriffszeiten bei Speicher folgen Moore's Law, sondern steigen deutlich langsamer.

1.3. First Generationn Routers

Zu BNC-Zeiten gab es für jeden Anschluss ein Line Interface mit BNC-Anschlüssen, die an einer gemeinsamen Backplane angeschlossen waren. Durch die Bus-Architektur muss jedes Pakete zweimal über den Datenbus gesendet werden.

1.4. Second Generation Routers

Zusätzlich werden auf den Line Cards Forwarding Caches eingebaut, die ausgehende Interfaces zwischenspeichern, wodurch die meisten Pakete nur einmal über den Datenbus geschickt werden müssen. Durch das Caching können jedoch Reordering-Probleme auftreten, wenn ein zweites Paket dank Cache schon verschickt werden kann, während das erste Paket noch im Paketpuffer ist.

1.5. Third Generation Routers

In der Backplane wird eine Switchingmatrix gepflegt, um Pakete hin- und herzusenden. In den Line Cards wird jeweils eine Forwarding-Tabelle von der CPU gepflegt, sodass keine Zugriffe mehr auf die Backplane nötig sind, um das Zielinterface zu bestimmen.

1.6. Fourth Generation Routers

Router sind teilweise als Multi-Chassis-Systeme mit optischen Links zwischen den Chassis aufgebaut. Damit hat man im Prinzip schon ein Netzwerk innerhalb des Routers.

1.7. Generic Router Architecture

Bei jeden eingehenden Paket wird anhand der IP-Adresse der Zielport aus der Forwarding-Table ausgelesen. Header (TTLs) werden aktualisiert und über ein Switching Fabric an den Output Buffer des ausgehenden Interfaces geschickt. Anschließend wird das Paket am ausgehenden Link verschickt.

Für diese Vorgänge ist generell sehr wenig Zeit möglich; bspw. sind für 40 Gbps-Switching 8 ns Zeit für IP-Adress-Lookup verfügbar. Solche Lookups können jedoch nicht mit einfachen Hash-Tabellen gelöst werden, da diese Worst Case mehr Speicher nehmen als verfügbar ist. Durch Bäume lassen sich zwar die hierarchischen Strukturen, die für Lookups in CIDR nötig sind, speichern, jedoch sind bei Baumsuchen Speicherzugriffe nicht sehr cacheeffizient, was Lookups durch Cache Misses sehr teuer machen kann.

Für die Speicherung im Routern werden statt normalem RAM TCAMs benutzt. Dabei werden die Adressen für die Routen hinterlegt. Alles hinter der Subnetzmaske wird dabei auf „don't care“ gesetzt. Für eingehende Pakete werden dann einfach alle Lines im TCAM parallel durchsucht und der Eintrag mit der höchsten Priorität benutzt. So kann deterministisch und schnell eine Pattern-Suche gemacht werden. Nachteil dieser Technik sind hoher Energieverbrauch (es wird immer der ganze Speicher angesprochen) und hohe Kosten.

Für die Logik werden gerne FPGAs/ASICs eingesetzt.

1.8. Buffer Placement

Pakete müssen gelegentlich auch mal gespeichert werden. Dies kann entweder am Input Port oder am Output Port gemacht werden. Je nachdem, wo man dies tut, hat es verschiedene signifkant Eigenschaften.

1.8.1. Output queueing

Output queueing ist hinsichtlich der Latenz optimal. Worst Case kommt an jedem Input Port etwas für denselben Output Port an, die jedoch dort einfach gespeichert werden können, ohne andere Prozesse zu stören. Allerdings muss das Switching Fabric das n-fache (n ist die Anzahl Ports) der Line Rate schaffen.

1.8.2. Input queueing

Queues werden kurz vor dem Switching Fabric eingesetzt. Wenn jetzt also mehrere Ports einen Output Port ansprechen wollen, müssen die meisten warten. Für die Entscheidung, welches Paket wann geschickt wird, ist jetzt ein Scheduler nötig. Dafür muss die Switching Fabric aber nur so viel Durchsatz wie Line Rate haben, da an jeden Output Port höchstens mit Line Rate gesendet wird.

Input queueing ist anfällig für Head of Line Blocking. Wenn ein Paket in der Queue nicht verschickt werden kann, können auch nachfolgende Pakete dieses Input Ports (also auch solche, die an andere, freie Ports gerichtet sind), nicht verschickt werden. Also erhöht sich die Latenz und der maximale Durchsatz verringert sich.

1.8.3. Virtual Output Queueing

An die Input Queues werden jetzt mehrere Queues gesetzt (eine für jeden Output Port). Jetzt kann ein Scheduler entscheiden, welcher Port jetzt für welchen Output Port drankommt. Dies erlaubt es, dass Pakete andere überholen können, wenn sie an einen anderen Output Port gehen. Damit wird es wieder latenzoptimal.

1.9. The Evolution of Switching

Die gezeigten Verfahren sind zwar in der Theorie sehr gut, in der Praxis aber nur schwer umzusetzen. In der Praxis werden stattdessen schnellere Interfaces benutzt, um mehr Durchsatz zu erzielen.

2. Input-buffered Switches

2.1. Input-Queued Switch

Bei reinem Input-Queueing ist der switch nicht **work-conserving**. Wäre er das, würde ein Output Port, wann immer ein Paket für diesen Output Port irgendwo verfügbar ist, der Output Port nicht idle sein. Dies kann aber auftreten, wenn so ein Paket weiter hinten in einer Queue ist.

2.2. Simple Analysis

- Assumptions:
  - Time is slotted, trifft bspw. für ATM zu.

Die Frage ist jetzt, wie viele Zustände der Switch haben kann. Hierzu betrachtet man, wann welche Pakete in welche Input Queue liegen und an welchen Output Port diese geschickt werden sollen. Das ergibt für den 2x2-Switch 4 Zustände.

2.2.1. Balls-and-Bins Model

Der Backlog wird hier nicht betrachtet, sondern nur die Pakete an der HoL. In jedem Schritt wird von jeder Farbe eine Kugel entnommen (sprich: ein Pakete aus der Queue verschickt). Damit lässt sich für das 2x2-Modell die Anzahl Zustände auf 3 reduzieren, da es jetzt egal ist, an welchen Port welche Farbe in der HoL liegt. Dies lässt sich jetzt zu einer Markov-Kette umbauen.

2.2.2. Markov Chain

Bei Markov-Ketten werden alle Zustände und die Wahrscheinlichkeiten für Zustandswechsel aufgetragen. Zusätzlich wird zu jedem Zustand der Durchsatz aufgetragen. Die Wahrscheinlichkeiten für Zustandswechsel ergeben sich durch Überlegung, welche Pakete in einem Zustand verschickt werden können und wie viele Pakete welchen Typs mit welcher Wahrscheinlichkeit „nachrücken“.

In einem eingeschwungenen Prozess sollte die Wahrscheinlichkeit, sich in einem Zustand zu befinden mal dem Zustandswechsel in den Nachbarzustand genau so groß sein wie die Wahrscheinlichkeit, im Nachbarzustand zu sein mal die Wahrscheinlichkeit, zurückzuwechseln. Dies lässt sich mit der zweiten Hälfte der Kette fortführen. Wenn man nun noch fordert, dass alle Wahrscheinlichkeiten zusammen 1 ergeben müssen (was ja

gelten muss), lässt sich das Gleichungssystem lösen. Daraus ergibt sich am Ende ein Gesamtdurchsatz von 75 %.

Die Markov-Kette lässt sich sogar noch weiter vereinfachen, wenn als Zustände nur betrachtet, wie viele Pakete gleichzeitig verschickt werden können (die Zustände werden „kollabiert“). Dies macht auch die Analyse des 3x3-Szenarios einfacher. Bei der Berechnung der Wahrscheinlichkeiten muss trotzdem betrachtet werden, welche Zustände eigentlich hinter den kollabierten Zuständen liegen. nach Analyse ergibt sich hier ein Durchsatz von 68 %. Der Durchsatz fällt als weiter. Konkret: je mehr Ports, umso weniger Durchsatz gibt es, da es mehr Congestion gibt.

2.3. Closed Form Equations for Balls in Bins

Ein M/D/1-System ist ein Warteschlangesystem mit Markov-verteiltem Input, degeneriertem Output und einer Bedieneinheit (ein Port).

Nomenklatur:

- $E\{k\}$... Erwartungswert
- ρ... Durchsatz
- μ... Verarbeitungszeit
- σ... Standardabweichung von der Verarbeitungszeit

Da die Verarbeitungszeit im M/D/1-System immer gleich ist, ist $\sigma = 0$. Man kann jetzt $E\{k\} = 1$ setzen um zu betrachten, wie groß ρ am Port sein muss, damit das System instabil wird. Das Limit von 58 % zeigt auch, dass ab etwas mehr als 50 % Last der Delay steigt.

Zu beachten ist hierbei, dass es natürlich nur eine theoretische Annahme ist, dass das System ein M/D/1-System ist. Ethernet ist bspw. kein M/D/1-System und insbesondere gibt es durch unterschiedliche Paketgrößen auch unterschiedliche Zeiten, die ein Paket braucht, um verschickt zu werden.

2.4. Virtual Output Queues

Idee ist jetzt, an jeden Input Port für jeden Output eine Queue aufzubauen. Ein Scheduler entscheidet dann, welches Paket an welchen Output verschickt wird.

2.4.1. Basic Switch Model

An jedem Port i gibt es einen **Ankunftsprozess** $A_i(n)$, der Pakete für Port j empfängt $(A_{ij}(n))$, Warteschlangen $Q_{ij}(n)$, eine Switching-Matrix $S(n) \in \{0,1\}^{n \times n}$.

Der Ankunftsprozess wird wieder mit einer Matrix λ beschrieben, die die Ankunftswahrscheinlichkeit angibt, dass von Port i an Port j gesendet wird. Dabei muss sichergestellt werden, dass keine der Spalten- oder Zeilensummen > 1 ist, da sonst der Input überlastet wird.

2.4.2. Scheduling Algorithm

Ziel ist jetzt, $S(n)$ immer so zu wählen, dass der Switch möglichst 100 % Durchsatz hat. Das ist per Definition der Fall, wenn der Switch work conserving ist.

Für fixe Paketgrößen wird bei Work-Conservation auch der Delay minimiert. Für variable Paketgrößen hingegen kann es sinnvoller sein, kleinere Pakete schneller zu switchen als größere Pakete.

2.4.3. Common Definitions for 100 % Throughput

Die dritte Definition sagt aus, dass auch größere Bursts von Pakete mal über einem gewissen C sein dürfen, solange der Erwartungswert kleiner ist. Es ist also möglich, dass Bursts auch mal überschießen. In der Praxis reicht dies aus, damit nur wenig Paketverlust auftritt.

2.4.4. Uniform Traffic

Hier wird zunächt sehr unrealistisch angenommen, dass jeder an jeden mit der gleichen Paketrate sendet. Dann muss lediglich $\lambda < \frac{1}{N}$ sichergestellt werden, damit es nicht explodiert. Dieser Verkehr ist so einfach, dass quasi jeder Scheduling-Algorithmus 100 % Durchsatz schafft.

Uniform Cyclic Scheduling

In jedem Zyklus wird einfach die Switching-Matrix durchgeschaltet, sodass im Kreis jeder Input Port der Reihe nach auf jeden Output Port geschaltet wird. Dieser Scheduler ist fair und deterministisch, jedoch nicht latenzoptimal.

Wait Until Full

Es wird gewartet, bis auf jeder VOQ ein Paket anliegt und erst dann wird eine beliebige Permutation geschaltet. Dies ist zwar optimal für den Durchsatz, aber in Hinblick auf Delay sehr schlecht. Insbesondere führt das dazu, dass sich der Delay erst verbessert, wenn die Last *steigt*.

Uniform Random Scheduling

Abwandlung von UCS, bei der zu jedem Zeitpunkt eine zufällige Permutation gewählt wird. Dies erlaubt wieder die Analyse mit Markov-Ketten (mit unendlich vielen Zuständen), da es sich um ein M/M/1-System handelt. Durch Analyse der lokalen Gleichgewichtsbedingung für jeden Zustand lässt sich ein Erwartungswert für jeden Zustand berechnen. Damit wiederum lässt sich berechnen, wie lang ein Paket im System verbleibt (sprich: der Delay).

2.4.5. Non-Uniform Traffic with Known Traffic Matrix

Der Traffic ist zwar nicht uniform verteilt, aber die Traffic-Matrix λ ist bekannt (fix).

Bei einem uniformen Schedule wird das System sofort instabil. Es besteht jedoch die Möglichkeit, verschiedene Permutationen mit verschiedenen Wahrscheinlichkeiten zu schalten. Dies kann man bspw machen, indem eine feste Abfolge von Schedules in Reihe oder zufällig durchgeschaltet wird. Letzteres macht wieder eine Analyse mit Markov-Ketten möglich.

Die Kernfrage ist nun, ob eine solche Zerlegung sich verallgemeinern lässt. Tatsächlich lässt sich dies algorithmisch lösen.

2.4.6. Double Stochastic Matrices

Bei einem doppelt stochastischen Prozess können sich die Wahrscheinlichkeiten ändern. Dies tritt beispielsweise beim Besuch von Websites auf, wo die Wahrscheinlichkeit, dass Pakete verschickt werden, von der Wahrscheinlichkeit abhängig, wie häufig der Nutzer auf Links klickt. Es gibt also einen übergeordneten Prozess, der bestimmt, wie hoch die Wahrscheinlichkeiten sind, dass ein untergeordneter Prozess (hier: der Versand von Paketen) bestimmte Wahrscheinlichkeiten annimmt.

Eine zulässige Verkehrsmatrix ist doppelt substochastisch. Durch geschicktes Aufrunden lässt sich diese in eine doppelt stochastische Matrix überführen. Diese Matrizen wiederum können als Linearkombination endlich vieler Permutationsmatrizen dargestellt werden. Genau genommen werden „nur“ höchstens quadratisch viele benötigt.

Die Zerlegung in Permutationsmatrizen ist insofern interessant, als dass es davon nur quadratisch viele gibt, der Rechenaufwand sich also in Grenzen hält.

Mit bekannter Traffic-Matrix lässt sich nun eine Folge von Switching-Matrizen bilden, die dann in zufälliger Abfolge mit Wahrscheinlichkeit α_k durchgegangen werden. Dies führt bei bekanntem Traffic-Muster zu 100 % Durchsatz.

Example

- Prüfen, ob Matrix zulässig ist (alle Zeilen- und Spaltensummen kleiner 1[1])
- Λ' könnte bspw. auch durch lineare Optimierung bestimmt werden
  - Alle Zeilen- und Spaltensummen sind 1
  - Alle Zellen werden durch Variablen ersetzt, die mindestens so groß wie der ursprüngliche Wert sein soll
  - Maximiere die Variablen, um möglichst viel Headroom zu erzielen
- Strategie, um Permutationsmatrizen und dazugehörige α_k zu bestimmen:

[1] Wäre eine Zeilen- oder Spaltensumme 1, wäre das System äußerst instabil und kann schnell Congestion kriegen, sobald sich das Traffic-Muster minimal ändert.

- Permutationsmatrix Λ_k erzeugen, indem in jeder Zeile die größte zulässige Spalte gewählt wird (Zelle hat größtmöglichen Wert und Spalte wurde zuvor nicht gewählt).
- α_k wird zu dem kleinsten gewählt Wert gewählt.
- $\alpha_k \Lambda_k$ von Λ' abziehen, wiederholen, bis Λ' leer ist (also alles 0). Das ist spätestens nach N^2 Schritten der Fall, da in jedem Schritt mind. eine Zelle zu 0 wird.

In den Folien wird vorher ein Hilfsmatrix $\tilde{\Lambda} = 10\Lambda'$ gewählt, die einfach so hochskaliert wurde, dass alle Werte Vielfache von 1 sind.

Der Beweis, dass dies zu 100 % Durchsatz führt, lässt sich mithilfe von Lindleys Formel führen. Dabei ist zu beachten:

- $E\{A_{ij}(n)\}$ ist genau unser Traffic-Muster, also λ_{ij}
- $E\{S_{ij}\}$ ist genau die Häufigkeit, mit der wir Pakete rausschicken können (was wir eben berechnet haben).
- In Bilanz ist gemäß unserer Konstruktion $E\{A_{ij}\} \leq E\{S_{ij}\}$, also haben wir häufiger die Möglichkeit, Pakete zu verschicken, als welche ankommen.

2.4.7. Non-Uniform Traffic With Unknown Traffic Matrix

Da die Traffic-Matrix jetzt unbekannt ist, kann Birkhoff-von-Neumann nicht angewendet werden. Stattdessen wird jetzt einfach versucht, möglichst viel Traffic in jedem Schritt loszuwerden. Dazu werden größte Matchings im bipartiten Graphen zwischen Input Ports auf der einen und Output Ports auf der anderen Seite gebildet, bspw. mithilfe von Flussalgorithmen. Allerdings sind alle Flussalgorithmen sehr langsam (mind. quadratisch) und damit praktisch nicht benutzbar.

Weiterhin sind es nicht ausreichend, irgednein größtes Matching zu benutzen. Wie im Gegenbeispiel gezeigt, kann dies in manchen Fällen dazu führen, dass der Durchsatz nicht 100 % wird. Stattdessen wird versucht, ein Matching größten Gewichts zu finden, also möglichst viel Warteschlange in jedem Schritt abzuarbeiten. Statt jeder Kante im Matching Gewicht 1 zu geben, werden Gewichte anhand der Queue-Größe vergeben. Eine Frage ist jetzt natürlich, welche Strategien für die Auswahl von Gewichten genutzt werden sollen.

Longest Queue First (LQF)

Hier wird als Gewicht die Länge der Queue benutzt. Dies erzielt zwar 100 % Durchsatz, führt allerdings nicht zwangsläufig dazu, dass die längste Queue abgearbeitet wird. Insbesondere kann LQF dazu führen, dass einzelne Queues „verhungern“ (also nie gewählt werden).

Der Beweis ist ein bisschen bäh und hier nicht näher erläutert. Am Ende ist aber gezeigt, dass das System stabil ist (auch für verschiedene Arten, die Queue-Länge zu gewichten, bspw. egal ob linear oder quadratisch).

Maximal Matching

Tatsächlich zeigt sich bei Messungen, dass MWM bessere Performance (geringeren Delay) liefert als MSM. Allerdings ist die Berechnung von Matchings größten Gewichts extrem langsam (siehe Vorlesung „Effiziente Algorithmen“) und sehr schwer in Hardware implementierbar. Stattdessen wird nicht nach einer global optimalen Lösung gesucht, sondern nach einem lokalen Optimum. Dies wiederum ist mit Greedy-Algorithmen möglich und daher schnell implementierbar. Allerdings wird dadurch auch ein Fehler (Abweichung zur optimalen Lösung) eingeführt.

Eine einfache Strategie verbindet einfach irgendwie Input Ports mit Output Ports (zwischen denen tatsächlich Pakete geschickt werden sollen), bis nichts mehr verbunden werden kann. Um jetzt eine geeignete Kante zu wählen, wird auf das Kantengewicht (Anzahl Pakete in der Queue) geschaut und die Kante mit dem größten Gewicht gewählt. Damit ist der Fehler im schlimmsten Fall 50 %. Um jetzt dennoch einen Durchsatz von 100 % zu erzielen, wird die Hardware einfach doppelt so schnell getaktet.

Wave Front Arbiter

Requests werden in einem Gitter zwischen Input Ports und Output Ports angesehen. Die Idee ist jetzt, dass Punkte im Gitter entlang einer Diagonalen unabhängig voneinander sind. Bei WFA geht der Algorithmus jetzt Diagonale für Diagonale durch und aktiviert alle Verbindungen, die nicht bereits anderweitig in Konflikt stehen. Der Algorithmus zieht dabei $2n - 1$ Diagonalen, also $\mathcal{O}(n)$.

Ein Nachteil ist, dass WFA nicht mehr fair ist. Grundsätzlich wird hier nämlich immer Port 1 bevorzugt, da die „Welle“ immer dort startet. Außerdem lässt sich WFA noch weiter verbessern, indem immer zwei Diagonalen mit unabhängigen Punkten miteinander verbunden werden. Dies läuft jetzt in n Schritten, erfordert aber eine kompliziertere Schaltung. Allerdings wird der erste Port weiterhin bevorzugt, wenn man nicht an zufälligen Diagonalen anfängt.

Parallel Iterative Matching (PIM)

Alle Ports versuchen an andere Ports Signale zu schicken. Die Output Ports suchen sich einen Input Port aus, von dem sie ein Paket entgegennehmen. Bekommt ein Input Port mehrere Signale, sucht er sich einen Output Port aus. Dadurch können wieder Output Ports frei werden (Matching ist nicht maximal), weshalb der Algorithmus für die noch nicht belegten Ports wiederholt wird, bis ein möglichst großes Matching gefunden wird.

Implementiert wird die Signalsierung mittels programmierbaren Priority Encodern. Diese legen für n Eingangssignale am Ausgang die Adresse eines Eingangs an. Welche gewählt wird, ist abhängig von einer (einstellbaren) Prioritätenliste und welche Eingänge an sind. Für PIM werden diese Prioritäten einfach jedes Mal zufällig gewählt.

PIM erzielt im Worst Case maximal n Iterationen. Die Input und Output Arbiter können dafür aber unabhängig (und damit parallel) arbeiten. Die Anzahl unverbundener Ports konvergiert exponentielle gegen 0. Die Anzahl Iterationen, bis alle Ports verbunden

sind, ist ungefähr logarithmisch. Allerdings muss PIM auch wirklich mit mehreren Iterationen laufen, um große Matchings zu finden. Je mehr Ports, um so mehr Iterationen sind notwendig.

iSLIP

iSLIP ist eine Erweiterung von PIM. Statt die Prioritäten in den Priority Encodern zufällig zu wählen, werden Prioritäten jedoch zyklisch gewählt, wählen alle Output Ports in jeder Iteration unterschiedliche Input Ports. So fürt iSLIP unter hoher Last zu TDM und zu zufälligen Matchings bei niedriger Last. iSLIP konvergiert deutlich schneller als PIM und hat für hohe Lasten deutlich weniger Latenz.

3. Size and Organization of Router Buffers

3.1. Routers as Queue Servers

In diesem Kapitel nehmen wir eine Sender an, der TCP über einen Link sendet. Die Pakete kommen per Gigabit Ethernet bei einer FritzBox an, die per DSL ins Internet mit O2-Geschwindigkeit senden muss. Damit ist die FritzBox der einzige Bottleneck-Link (was auch üblich ist). Während also der Ethernet-Link nicht ausgelastet ist, ist auf dem DSL-Link Stau. Die Kernfrage ist jetzt, wie groß man die Queue machen muss. Modelliert wird der Router als M/M/1/B-System (1 Router mit Puffergröße B).

Nach der Formel für die Loss-Wahrscheinlichkeit konvergiert der Paketverlust durch steigendes B. In der Theorie kann also durch einen Puffer die Loss-Wahrscheinlichkeit gering gehalten wreden. Allerdings ist das Modell nicht realistisch, da TCP Bursts generiert (also der Ankunftsprozess kein Poisson-Prozess ist). Grundsätzlich gilt aber:

- Je größer der Puffer, umso teurer (allerdings durch DRAM nicht sehr teuer)
- Je größer der Puffer, umso größer die Latenz
- Je größer der Pfufer, umso geringer die Loss-Wahrscheinlichkeit

Ziel ist jetzt, den Bottleneck-Link möglichst durchgängig auszulasten. Das ist ein Argument für große Puffer, da so Schwankungen im Ingess Traffic ausgeglichen werden können. Dagegen spricht jedoch, dass TCP ohne ECN den Puffer zunächst immer komplett füllt, bevor Congestion Control die Senderate reduziert.

3.2. TCP ACK / self-clocking

TCP sendet neue Daten immer, wenn das Congestion Window größer ist als die Menge versendeter Daten, die noch nicht geACKt wurden. ACKs dienen also nicht nur der Bestätigung, dass Pakete angekommen sind, sondern erlauben dem Sender auch, mehr Daten zu senden.

Die AIMD-Phase vom TCP führt dabei zu einem Sägezahn-Muster, welches die Menge ungeACKter Daten im Netz (das Congestion Window) additiv erhöht und multiplikativ verkleinert. In der Praxis folgt TCP jedoch nicht ganz diesem Muster. In der Initialisierungsphase wird Slow Start ausgeführt, d. h. das Congestion Window erhöht sich exponentiell, bis der intiale Congestion Threshold erreicht wurde. Dies ist das Verhalten des (alten) TCP Tahoe. TCP Reno erweitert dies um Mechanismen für Fast Recovery nach Congestion, hat aber immer noch Einbrüche im Congestion Window (und damit der Senderate). Die FritzBox muss also für eine optimale Auslastung des Links solche Einbrüche durch Puffer kompensieren können.

3.3. Deepen understanding of TCP Reno

Ziel ist, das Sägezahnmuster von TCP Reno genauer zu analysieren. Dazu werden zunächst Differentialgleichungen für die Größe des Congestion Window aufgestellt und aufgelöst:

$$W = C \cdot RTT \tag{3.1}$$
$$= C \cdot (2T_p + T_q) \tag{3.2}$$
$$= C \cdot \left(2T_p + \frac{Q(t)}{C}\right) \tag{3.3}$$
$$\dot{W} = \frac{1}{RTT} = \frac{1}{2T_p + \frac{Q(t)}{C}} \tag{3.4}$$
$$= \frac{C}{W} \tag{3.5}$$
$$W\frac{\mathrm{d}W}{\mathrm{d}t} = C \tag{3.6}$$
$$W \cdot \mathrm{d}W = C \cdot \mathrm{d}t \tag{3.7}$$
$$\int W \mathrm{d}W = \int C \mathrm{d}t \tag{3.8}$$
$$\frac{1}{2}W^2 = Ct + k \tag{3.9}$$

Wir können annehmen, dass die Warteschlange zu Beginn leer ($Q(0) = 0$) ist:

$$W(0) = C \cdot 2T_p \tag{3.10}$$
$$\frac{1}{2} \cdot (2CT_p)^2 = 0 + k \tag{3.11}$$
$$2C^2T_p^2 = k \tag{3.12}$$
$$\implies W = \sqrt{2C + 4C^2T_p^2} \tag{3.13}$$

Das Muster ist also gar kein Sägezahnmuster, sondern folgt einer konkaven Funktion! Je mehr gesendet wird, umso mehr Queueing tritt auf. Dadurch steigt das Delay, weshalb die Senderate sich zunehmend langsamer erhöht, bis tatsächlich Congestion auftritt und das Congestion Window halbiert wird.

3.4. The Single Flow Case

Zunächst wird angenommen, dass in unserem Szenario nur eine TCP-Verbindung exisitert. Diese Verbindung füllt also langsam die Puffer, bis dieser vollläuft. Da jetzt Congestion auftritt, halbiert sich die Größe des Congestion Windows von W^* auf $\frac{W^*}{2}$. Da aber noch ca. W^* unterwegs sind, muss TCP warten, bis $\frac{W^*}{2}$ Pakete geACKt wurden. In dieser Zeit, in der nichts gesendet wird, muss der Router aber die Queue leeren, um maximalen Durchsatz zu erzielen. Nimmt man an, dass die Gegenstelle jedes eingehende

Paket ACKt, dauert es $\frac{\frac{W^*}{2}}{C}$, bis der Sender wieder senden kann. Der Puffer muss also groß genug sein, um mindestens so lange senden zu können, also mindestens $\frac{W^*}{2}$. Da TCP Congestion Windows üblicherweise im Bereich mehrerer Megabyte sind, müssen die Puffer also sehr groß sein!

Betrachtet man nun die Situation nach der Wartezeit, dann ist der Puffer leer. Das aktuelle Congestion Window ist $W = \frac{W^*}{2}$. Um den Bottleneck Link voll auszulasten, muss der Sender idealerweise mit Rate C senden. Da $R = \frac{W}{RTT}$, muss $R = C = \frac{W}{RTT}$ gelten. Also $C = \frac{\frac{W^*}{2}}{2T_p}$. Durch Einsetzen und Auflösen gilt $B \geq 2\,T_p C$. Je größer also die Linkkapazität (oder die RTT), umso größer müssen also die Puffer sein. Während also in Data Centern kleine Puffer aufgrund der kleinen RTT ausreichen, kann bei großen Latenzen die benötigte Puffergröße extrem ansteigen.

Leider kann TCP auch nicht auf unterschiedlich große Puffergrößen reagieren, da TCP nicht genau erkennen kann, ob der Bottleneck Link zu große oder zu kleine Puffer hat.

3.5. Many Flows: Synchronized or Not?

Die Frage ist jetzt, wie sich das Netz verhält, wenn es viele Flows gibt. Die Frage ist zunächst, ob TCP sich synchronisieren kann, d. h. ob mehrere Verbindungen gleichzeitig das Congestion Window erhöhen. In so einem Fall verhält sich aus Sicht des Routers TCP wie ein großer TCP Flow. Das kann passieren, wenn wenige Flows ungefähr gleiche RTTs zu den Zielen haben, da sie dann ähnliches Verhalten haben, nachdem sie durch Congestion in der Queue alle einmal (etwa zeitgleich) erkannt haben, dass Congestion aufgetreten ist.

Bei vielen Flows wird das Verhalten chaotischer und Synchronisation tritt nicht mehr auf. Dann ist das Sendeverhalten gleichmäßiger. Dadurch ist der Unterschied zwischen maximaler und minimaler Puffergröße geringer, weshalb die Puffer kleiner gewählt werden können. Das Verhalten der TCP-Flows lässt sich auch mathematisch analysieren. Dazu werden die Congestion Windows der Flows zu einem Window aggregiert. Dieses bewegt sich ungefähr in der Größenordnung der Queue-Größe. Die W_i werden bei der Analyse als normales Sägezahn-Muster (ohne die konkave Form) angenommen. Das Sägezahn-Muster schwankt zwischen $\frac{2}{3}$ und $\frac{4}{3}$ um einen Erwartungswert. Unter der Annahme, dass es ein echter Sägezahn ist (also mit linearem Anstieg abseits der Sprungstelle), ist die Window-Größe dann uniform verteilt. Für jeden Flow gibt es jetzt also die Window-Größe W_i als Zufallsvariablen, welche sich aufaddieren und stochastisch analysieren lassen.

Es ergibt sich für die Standardabweichung eine obere Schranke. Diese ist umgekehrt proportional zur Anzahl der Flows! Je mehr Flows also auf dem Link sind, umso geringer die Standardabweichung. Es ist also bei Core Routern mit vielen Flows nicht nötig, große Puffer zu erhalten, um Schwankungen im Congestion Window auszugleichen.

3.6. Short Flows

Allerdings ist in modernen Szenarien nicht mehr wichtig, dass große Verbindungen mit vielen Daten möglichst viel übertragen wird. Die üblichen Szenarien moderner Webanwendungen (z. B. HTTP, Instant Messaging) werden so wenige Daten übertragen, dass TCP die Slow Start Phase ne verlässt. Für die Analyse wird jetzt eine unendliche Geschwindigkeit und auch ein Poisson-Prozess angenommen. Auch dies ist in modernen Netzwerken nicht der Fall, erleichtert aber die Analyse.

Der Router wird hier zu einem *M/G/1-System* (d. h. Markov-Ankunftsprozess mit generisch verteiltem Verarbeitungsprozess). Mit lustiger Mathe lässt sich der Erwartungswert für die Verarbeitungszeit T_v berechnen. Dabei zeigt sich, dass die Zeit für hohe Last (ρ) extrem ansteigt. Die Verteilung der Burst-Größen ist allerdings sehr meh verteilt. Da bei größeren Paketzahlen auch kleinere Bursts verschickt werden, springen die Häufigkeiten der Burstgrößen an den Sprungstellen (Zweierpotenzen) stark nach oben an, während alle anderen Häufigkeiten deutlich kleiner sind (weil sie nur für Rest-Bursts auftreten können). Insgesamt klingen die Häufigkeiten Burst-Größen exponentiell ab.

Die Durchschnittliche Queue-Länge zeigt auch Spikes in der Nähe Grenzen der Burst-Größen. Dazwischen gibt es jeweils ein lokales Minimum, was dadurch bedingt ist, dass für Restpakete im letzten Burst weniger Paket verschickt werden. Spikes ergeben sich an den Stellen, wo Bursts immer voll ausgenutzt werden.

Viele kleine Flows, die die Slow-Start-Phase nicht verlassen, akkumulieren sich nicht. Es ist für kleine Flows nicht notwendig, extrem große Puffer zu maintainen. Das Buffer-Bloat-Problem ist also rein darauf basierend, dass das Messszenario eine möglichst große Bandbreite bei großen Dateiübertragungen ist. Es ist vielleicht also doch nicht so sinnvoll, die Buffer sehr groß zu halten.

3.7. Very Small Buffers?

Mit dem bisherigen Modell ist es problematisch, Puffer richtig klein zu gestalten, da TCP ansonsten viel Paketverlust und damit geringe Performance erzielt. Um also geringe Puffer zu benutzen, muss TCP dazu gebracht werden, die Senderate zu limitieren.

Durch Anwerfen der großen Markov-Mühle ergibt sich ein schöner Term, bei dem mit $\frac{1}{\rho}$ logarithmiert wird. Also hält sich die benötigte Puffergröße hier im Rahmen. Dabei ist die benötigte Puffergröße nicht mehr von der Anzahl Flows oder der Linkkapazitäten abhängig. Solange man also nicht mehr den vollen Durchsatz erzielen können will, können auch kleine Puffer genutzt werden. Allerdings wird dafür immer noch ein *paced TCP* benötigt, was keine Bursts, sondern bei einer fixen Rate sendet.

Die Benutzung kleiner Puffergrößen ist insofern wünschenswert, als dass kleine Puffer auch gut in schneller Hardware (z. B. SRAM) statt langsamer Hardware (z. B. DRAM) implementierbar sind.

3.8. Active Queue Management

Eine gute Puffergröße ist nur schwer abzuschätzen, da dies nicht zuletzt auch von T_p abhängt. Dies ist ein Problem, da T_p nicht bekannt ist und von Umgebung zu Umgebung schwankt (z. B. extrem klein in Data Centern, aber extrem groß bei Satelliten). Stattdessen soll jetzt versucht werden, die Queue-Größe dynamisch zu verändern, indem noch vor dem tatsächlichen Volllaufen der Queues Pakete verworfen werden. Dies erlaubt es auch TCP, schneller Congestion zu erkennen und das Congestion Window zu verkleinern.

3.8.1. Random Early Drop

Die Idee von **Random Early Drop** ist, ab einer gewissen Queue-Auslastung probabilistisch Pakete zu verwerfen, obwohl noch Platz in der Queue wäre. Die Wahrscheinlichkeit dafür wird abhängig von der Queue-Größe und zwei Thresholds (min, max) gewählt. Statt Pakete zu droppen, können aber auch Pakete mit ECN (siehe Abschnitt B.3) geflaggt werden, um TCP zu signalisieren, dass Congestion auftritt. Um nicht zu stark auf Bursts zu reagieren, wird jedoch nicht die tatsächliche Queue-Länge durch einen exponentiell gewichteten gleitenden Durchschnitt geglättet, um eine durchschnittliche Queue-Größe zu erhalten.

Ein Problem mit RED ist, dass die Parameter gut eingestellt werden müssen. Die Parameter richtig einzustellen besteht immer aus Kompromissen, die von Netz zu Netz konfiguriert werden müssten. Das macht RED in der Praxis nur schwer nutzbar.

3.8.2. Fair RED / Flow RED (FRED)

Die Idee ist, abhängig vom Flow-Typ unterschiedlich zu reagieren (bspw. nicht-adaptive Flows auch nicht beeinflussen). Statt die Queue-Länge zu betrachten, wird für Flows gezählt, wie oft Flows Penalties erhalten. Das Problem hierbei ist jedoch, dass die Router jetzt per-Flow State handlen müssen, was nicht praktikabel ist.

3.8.3. CHOKE

Statt per-Flow State zu halten, werden eingehende Pakete jetzt mit Paketen verglichen, die schon in der Queue sind. Wenn zwei Pakete die gleiche Flow ID haben, wird das Paket verworfen. Hier spart man sich zwar per-Flow State, ist aber unfair gegenüber Szenarien mit nur wenigen Flows,

3.8.4. Adaptive RED

Ziel war, Parameter von RED automatisch anzupassen, um es leichter einstellbar zu machen. Dazu werden Parameter abhängig von der Linkkapazität gewählt und max_{th} abhängig von min_{th} gewählt. w_q wird dabei mit einer Exponentialfunktion abhängig von

der Linkkapazität berechnet, damit die Geschwindigkeit unabhängig von der Geschwindigkeit ist. P_{max} wird regelmäßig aktualisiert, um möglichst groß zu sein, aber gleich zeitig möglichst selten die Queue vollaufen zu lassen.

3.8.5. Stabilized RED

3.8.6. CoDel – Controlled Delay

Die Adaptionen von RED haben gezeigt, dass RED als Ansatz nicht wirklich zielführend ist. Ein weit verbreitetes AQM-Verfahren ist **CoDel**. Anstatt sich Queue-Längen anzuschauen, wird beobachtet, wie lange Pakete brauchen, um durch die Queue zu gehen. Das funktioniert unabhängig von der Anzahl der Queues und mit variablen Link-Raten (bspw. bei Funkverbindungen).

Beobachtet wird die **Sojourn time**, also die Zeit, wie lange eine Paket vom Empfang am Input Port bis zum Versand am Output Port benötigt. Bei steigender Senderate von TCP Flows steigt die Sojourn Time. Wenn die Sojourn Time einen Threshold übersteigt, wird ein Paket gedroppt und nach einem gewissen Intervall wieder die Sojourn Time geprüft. Je länger die Sojourn Time zu groß ist, um so geringer werden die Intervalle, in denen geprüft und gedroppt wird.

Das Intervall wird als $\frac{Interval}{\sqrt{dropcount}}$ berechnet. Das soll dem TCP Flow Zeit geben, Congestion zu erkennen und darauf zu reagieren. Je länger jedoch die Sojourn Time ist, umso häufiger werden auch Pakete gedroppt und der Drop Count steigt.

Wenn Pakete gedroppt werden, passiert dies außerdem am Kopf der Queue, nicht am Ende. Dies hat den Vorteil, dass TCP schneller die Congestion erkennt und verspätete Pakete von Echtzeitanwendungen eher gedroppt werden als solche, bei denen es sich tatsächlich noch lohnt, dass sie ankommen.

4. Interfacing NICs

Dieses Kapitel beschäftigt sich mit der Funktionsweise von Netzwerkkarten.

4.1. Ingress and Egress Packet Handling in Servers

Eingehende Pakete kommen am Port an, dekodiert das Paket und prüft die MAC-Adresse, ob sie mit der MAC-Adresse der Karte übereinstimmt. Anschließend wird die CRC-Checksumme geprüft. Anschließend wird das Paket über den PCI-Bus in einen Ringpuffer im RAM geschrieben. Wurde das Paket fertig geschrieben, wird ein Interrupt ausgelöst, um die CPU über das neue Paket zu informieren.

Der Scheduler merkt sich den Eingang des Interrupts für spätere Behandlung (*Soft Interrupt*), da sonst bei zu vielen Paketen die CPU „lahmgelegt“ werden kann. Zu einem späteren Zeitpunkt behandelt der Treiber den Soft Interrupt, indem er die Pakete von der Netzwerkkarte ausliest und an den Network Layer des Kernels weitergibt. Anschließend wird das Paket an einen Socket übergeben, der die dazugehörige Anwendung über den Eingang des Paketes informiert und die Daten des Pakets bereitstellt.

Der umgekehrte Weg läuft genau umgekehrt. Die vom Socket entgegengenommenen Daten werden in IP- und Ethernet-Header verpackt, an den Treiber übergeben und in einen Ringpuffer geschrieben, der von der Netzwerkkarte für den Versand ausgelesen wird. Anschließend informiert die Karte das Betriebssystem wieder per Interrupt über den erfolgreichen Versand.

Diese Methode hat mehrere Performance-Probleme:

- Speicherzugriffe (und damit verbundene Cache-Misses, Locking, Memory Alignment, etc.)
- Hoher Speicherverbrauch für Puffer (und dadurch auch schlechtere Cache-Effizienz)
- Kopieren von Daten
- Interrupts
- Kontextwechsel

Solche Operationen sind sehr teuer und führen daher zu Performanceproblemen. Wenn Dienste virtualisiert sind, finden zusätzlich noch Kontextwechsel zum Hypervisor statt und möglicherweise kann es auch durch Sicherheitsmechanismen zu weitere Kopieroperationen kommen.

4.2. Scatter Gather Listen

Experimente, Kopien an der CPU vorbei (z. B. direkt per DMA) durchzuführen, hat sich als fehlerhaft herausgestellt. Das Ziel ist also, möglichst wenige Kopien zu erstellen, idealerweise ein Paket nie zu kopieren.

Moderne Netzwerkkarten unterstützen dafür **Scatter Gather Listen**, die Pakete nicht mehr als zusammenhängendes Segment speihern, sondern in Segmenten, über die mit einem Paketverlust zugegriffen wird.

Dieser Mechanismus erfordert dennoch große Segmente, da jeder Zugriff auf ein Segment zu zusätzlichem I/O führt. Außerdem müssen die Anwendungen Scatter Gather Listen unterstützen und statt den üblichen `recv` und `send` Calls die komplizierteren `recvmsg` und `sendmsg` Calls benutzen.

4.3. Head-, Tailroom and Header Split

Die Idee ist, bei der Speicherallokation vor und nach den Nutzdaten direkt Speicher zu allokieren, der von der Netzwerkkarte beschrieben werden kann, um Header und Trailer (z. B. für IPsec) zu schreiben. Solange der Head- und Tailroom ausreicht, können so Kopien vermieden werden. Allerdings ist nicht unbedingt bekannt, wie viel Headroom benötigt wird. Außerdem besteht so das Problem, dass die Daten nicht unbedingt aligned sind, was der Cache-Effizienz schadet. Dieses Problem lässt sich nicht wirklich lösen, da nicht bekannt ist, wie groß der Headroom sein muss.

Um das Problem (theoretisch) zu lösen, können Netzwerkkarten Pakete tiefer analysieren und die Headerdaten getrennt von den Nutzdaten in verschiedenen Segmenten einer Scatter Gather Liste abzulegen. So können die Daten immer aligned werden (da das Segment an einer Cache-Grenze allokiert wird) und der Versand an mehrere Empfänger ist auch effizient möglich, da die Headersegment alle auf die gleichen Nutzdaten zeigen können. In der Praxis hat sich dies jedoch nicht bewährt, da Spezialfälle wie Fragmentierung, etc. von der Netzwerkkarte nicht behandelt werden. Daher muss auch noch der konventionelle Datenpfad durchlaufen werden. Durch den Split sind außerdem immer mehrere DMA-Transfers notwendig, um auf das gesamte Paket zuzugreifen. Daher wird es in der Praxis auch nicht wirklich benutzt.

4.4. IP and TCP Offloading

Die Idee ist, Aufgaben von der CPU auf die Netzwerkkarte auszulagern. Dafür müssen Netzwerkkarten die Pakete besser verstehen. Dafür müssen Netzwerkkarten entsprechende Offloading-Features unterstützen, um bspw. Checksummen von IP- oder TCP-Headern zu berechnen, sodass die CPU an der Stelle einfach einen Dummy-Wert eintragen kann.

Large Receive Offload und **Large Segment Offload** erlauben es Netzwerkkarten, TCP auf der Netzwerkkarte zu interpretieren. So kann die Netzwerkkarte eingehende Bursts eines TCP Streams gemeinsam behandeln, um diese bspw. zu größeren Paketen

zusammenzufügen, bevor die Pakete an das Betriebssystem weitergegeben werden. So kann die CPU mit weniger Speicherzugriffen größere Mengen an Daten von der Netzwerkkarte empfangen. Umgekehrt kann Large Segment Offload große Pakete vom Betriebssystem auf der Netzwerkkarte in kleinere TCP-Pakete aufteilen. Insgesamt führt dies dazu, dass es zu weniger Interrupts und weniger Speicherzugriffe führt und weniger Routing Overhead auftritt.

Probleme mit dieser Aggregierung treten jedoch dann auf, wenn Pakete klein zwecks Latenz klein bleiben sollen (z. B. bei interaktiven Sitzungen, VoIP, etc.). Solche Anwendungen können also nicht optimiert werden und müssen daher wieder über den langsame(re)n Pfad bearbeitet werden. Damit werden also eher die großen Flows beschleunigt, die gar nicht so sehr darauf angewiesen sind, möglichst schnell verarbeitet zu werden.

4.5. Multi-Queue NICs

Der Zugriff auf Queues wird normalerweise über Locking geschützt, wenn mehrere CPUs die Queues auslesen sollen. Das macht Zugriffe sehr langsam. Moderne Netzwerkkarten unterstützen dagegen mehrere Queues pro Richtung, sodass jede CPU ihre eigene Queue erhält, über die Pakete empfangen werden können, ohne vorher ein Lock auf die Queue setzen zu müssen. Durch mehrere Queues können bspw. auch virtuelle Maschinen eigene Queues erhalten, um unabhängig voneinander ohne Locking Pakete empfangen zu können.

Beim Empfangen muss jetzt jedoch genauer darauf geachtet werden, welche Pakete in welche Queue abgelegt werden. Einfache Round-Robin-Verteilung von Paketen (und andere einfache Methoden) führen dazu, dass Pakete für die gleiche Anwendung auf mehreren CPUs ankommen (wodurch für die Zusammenführung wieder Locking nötig ist) bzw. Reordering auftreten kann.

Um bessere Effizienz zu erzielen, gibt es Mechanismen wie **Flow Director**, die beim Versand von Paketen eines Flows den umgekehrten Flow (Sender/Empfänger vertauscht) auf die RX-Queue der entsprechenden CPU per Flow-Regel zu mappen. Dies führt zu guter Cache-Nutzung und damit guter Performance auch auf Anwendungsseite, hat aber hohe Anforderungen an die Netzwerkkarte. Außerdem ist dies durch die Anzahl Flow-Regeln limitiert, die auf der Netzwerkkarte hinterlegt werden können.

Dazu gibt es eine etwas zustandslosere Alternative, die rein auf den Paketheadern der eingehenden Pakete arbeitet, ohne Daten speichern zu müssen. Bei **Receive Side Scaling** wird ein sog. *Toeplitz-Hash* über Headerfelder berechnet, der sich innerhalb von Paketen eines Flows nicht ändert. Der Hashwert wird dann benutzt, um den Index einer Receive Queue zu erhalten. Bei der Berechnung des Toeplitz-Hashes wird ein Key eingerechnet, der geheimzuhalten ist, da ansonsten unter Kenntnis des Schlüssels ein DoS-Angriff auf eine Anwendung potentiell dadurch geführt werden kann, dass nur Flows erstellt werden, die eine einzelne CPU lahmlegen. Außerdem wird auch das Problem nicht gelöst, dass *Elephant Flows* weiterhin nur von einer CPU bearbeitet werden können.

4.6. Increasing Speed for Virtual Machines

Das Ziel ist jetzt, auch für jede VM eine eigene Netzwerkkarte zu emulieren. Damit soll vermieden werden, dass Hypervisor Pakete behandeln müssen. Statt jetzt jedoch mehrere Netzwerkkarten im VM-Host zu verbauen, wird **SR-IOV** eingesetzt. Dabei meldet sich die Netzwerkkarte am System als mehrere virtuelle Netzwerkkarten (PCI IDs). Der Hypervisor kann dann eine solche virtuelle Netzwerkkarte direkt einer VM zuweisen, sodass Interupts, Pakete, etc. direkt an die VM statt an den Hypervisor geleitet werden. Zur besseren Steuerung von Berechtigungen meldet sich die Netzwerkkarte mit sog. „Physical Functions“ und „Virtual Functions“ an, sodass die VMs nur virtuelle Karten mit beschränkten Berechtigungen erhält. Andere Funktionen können je nach Hersteller auch bereitgestellt werden.

Auf der Host-Seite werden jetzt für jede VM eigene Ringe benötigt und die IOMMU muss so konfiguriert werden, dass jede VM nur auf die ihr zugewiesene virtuelle Netzwerkkarte zugreifen kann. Um Pakete an die passende virtuelle Netzwerkkarte zu leiten, ist auf der physischen Netzwerkkarte ein virtueller Switch, der bspw. anhand der virtuellen MAC-Adressen erkennt, welches Paket an welche virtuelle Netzwerkkarte geschickt werden muss. Dieser virtuelle Switch muss nicht nur eingehende Pakete an die richtige VF leiten, sondern auch zwischen VFs (Ringen der VMs) Pakete weiterleiten können.

4.7. Software Architectures for Efficient Network Appliances

4.7.1. Reducing Context Switches

Durch die geschichtete Softwarearchitektur moderner Betriebssysteme gibt es viele Kontextwechsel, die natürlich langsam sind. Das lässt sich in NICs nicht direkt umgehen. Stattdessen ist die Idee, Performance-kritische Anwendungen direkt im Kernel auszuführen, um Kontextwechsel zu sparen. Beispiele dafür sind NFS Server, Web Server (z. B. TUX), TLS. Das erlaubt nicht nur das Ersparen von Kontextwechseln, sondern ermöglicht auch eine bessere Kontrolle darüber, welche Features bspw. auf eine Netzwerkkarte ausgelagert werden können (z. B. TLS).

Hauptproblem ist hier die Sicherheit. Wenn Bugs auftreten, kann dies das komplette System betreffen. Fehler lassen sich im Kernel nur schwer debuggen, was auch die Entwicklung schwierig macht. Außerdem sind die Schnittstellen (und damit Features) im Kernel stark eingeschränkt, was die Nützlichkeit einschränkt.

Ein besserer Ansatz ist, nur einzelne Features zu outsourcen, bspw. Übertragung einzelner Dateien auf einem Socket, statt einer kompletten HTTP-Implementierung. So kann man viele Kontextwechsel, die für die Übertragung großer Daten nötig wären, ersparen. Aber auch hier eignen sich die Features nur für wenige spezielle Anwendungen. Auch dies ist also keine generelle Lösung.

Eine andere Idee ist, den Kernel als Zwischenschritt beim Networking zu eliminieren. Statt Code zur Paketverarbeitung im Kernel auszuführen, wird sämtlicher Code im Userspace ausgeführt. Dafür muss der Speicher der Netzwerkkarte direkt in den Userspace einer Anwendung gemappt werden. Das hat jedoch den Nachteil, dass immer nur eine

Anwendung auf ein NIC zugreifen kann.[1] Dafür werden aber sämtliche Kontextwechsel für die Paketverarbeitung vermieden. Außerdem kann die Anwendung Interrupts Polling benutzen, was Verzögerungen und Unregelmäßigkeiten durch Interrupts vermeidet und bei guter Programmierung auch geringe Latenzen ($<100\mu s$) erzielt. Das wiederum kann die Performance bspw. für Netzwerkspeicher deutlich verbessern. Hauptnachteil bei diesem Ansatz ist, dass die Anwendung den kompletten Netzwerkstack (von ARP bis TCP) selbst implementieren muss. Dies wird teilweise aber von Frameworks übernommen.

4.7.2. Multicore Scheduling

Um hohe Paketraten verarbeiten zu können, müssen Pakete auf mehreren CPUs parallel verarbeitet werden können. Dafür können verschiedene Modelle genutzt werden.

Im **Push to pull path** werden Pakete von einem Thread nach dem Empfang bis zu einer Transmit Queue verarbeitet. Ein anderer Thread nimmt die Pakete aus der Transmit Queue und verschickt sie auf dem NIC. Beim **Full push path** werden Pakete von dem Thread verschickt, der auch die Pakete empfangen hat.

Das Push to pull Design hat das Problem, dass Daten zwischen Threads ausgetauscht werden müssen, was sehr cache-ineffizient und damit langsam ist. Außerdem muss zwischen den Threads kommuniziert werden, welche Speicherbereiche wieder vom Input Thread benutzt werden können. Außerdem laufen die Input Threads unterschiedlich schnell zu den Output Threads, weshalb ein Scheduler eigentlich den jeweils schnelleren Thread weniger häufig ausführen müsste, damit der langsamere Threads mehr Pakete verarbeiten kann. Das Full push path Design vermeidet Kommunikation zwischen Threads weitestgehend, was sich positiv auf die Geschwindigkeit auswirkt. Da der Input Thread auch das Paket verschickt, ist ihm auch bekannt, welche Speicherbereiche wieder für den Empfang von Pakete benutzt werden können. Außerdem muss ein Scheduler (unter der Annahme, dass alle Threads ähnlich viel Input-Last haben) nur dafür sorgen, dass Threads ähnlich häufig ausgeführt werden.

Ein weiteres Problem ist die Frage, wie Daten zwischen Threads geteilt werden. Hier gibt es auch wieder die Möglichkeit, sowohl Daten zwischen Threads zu teilen (mit threadsicheren Datenstrukturen) als auch jedem Thread nur Daten zu geben, die er tatsächlich benötigt.

[1] Mit manchen Netzwerkkarten ist es auch möglich, über Flow-Regeln Queues auf Anwendungen aufzuteilen, aber das ist schwierig umzusetzen.

5. Software Defined Networking

Statt jetzt nur einzelne Netzknoten zu betrachten, ist das Ziel von *Software Defined Networking*, das Netz als ganzes so zu steuern, dass die Leistung optimiert wird.

5.1. High-Level Motivation

Bei der Entwicklung von Computern gab es einen Übergang von Computern, die nur ein Programm ausführen, hin zu Computern mit Betriebssystemen, die ein Betreibssystem haben und über eine „offene“ Schnittstelle die Ausführung mehrerer hardwareunabhängiger Programme erlaubt. Statt große spezialisierte Mainframes mit spezialisierter Hardware, spezialisiertem Betriebssystem und spezialisierten Anwendungen zu betrieben, sind heute viele General-Purpose-Rechner mit generischem Betriebssystem im Einsatz.

Bei Netzwerkhardware war lange auch der Standard, dass eine spezialisierte Hardware ein spezialisiertes Betriebssystem hat, auf dem spezialisierte Features laufen. Die Systeme sind sehr komplex, geschlossen und proprietär, ähnlich wie es auch auf früheren Mainframes üblich war. Das macht es nicht nur schwer, Konkurrenz auf dem Markt zu etablieren (was wiederum der Weiterentwicklung der Technik schadet), sondern erlaubt auch viele „dreckige“ Hacks, bei der die klare Schichtung zwischen Hardware, OS und Anwendung aufgebrochen wird, um kleine Performance-Gewinne zu erzielen.

Stattdessen wäre ein Wunsch, dass sich Router hin zu einer Schichtung ähnlich wie bei PCs entwickeln, bei der proprietäre Switching-Chips über eine offene Schnittstellen von einer Control Plane gesteuert werden, die über eine offene Schnittstelle die Ausführung beliebiger Anwendungen (z. B. für eigene Routing-Protokolle) erlaubt. Das würde es auch erlauben, die schnelle Entwicklung, die mittlerweile in Rechenzentren stattfindet, gut angehen zu können.

5.2. Archictecture

Die Kernidee von SDN ist, die eigentliche Logik, welche Pakete wie weitergeleitet werden, auf einen externen Controller auszulagern, in dem dann komplexere Entscheidungen als auf Switchen getroffen werden können. Statt in der Control Plane jedes Switches OS und Dienste laufen zu lassen, besteht die Control Plane nur noch aus einem Betriebssystem, was Daten mit einem zentralen Server (dem *Controller*) austauscht. Auf diesem laufen dann die Services, die die Switching-Entscheidungen treffen.

Der Controller kann dann den Switchen Regeln senden, die bspw. Flows oder Header matchen und ihnen dann einen Output Port zuweisen (oder auch andere Aktionen wie bspw. NAT durchzuführen). Diese können in einer Flow-Tabelle gespeichert werden, die

im Data Patch schnell ausgewertet werden kann. Es ist auch möglich, dass Switche den Controller darüber informieren, wenn Pakete eintreffen, die noch keinen passenden Eintrag in der Flow-Tabelle haben, um bspw. auf neue Flows reagieren zu können oder auch bspw. ARP-Traffic vom Controller zu beantworten statt im Netzwerk zu fluten. Lücke

5.3. CAP

TL;DR Man kann höchsten zwei von den Zielen Verfügbarkeit, Konsistenz und Partitionstoleranz erzielen.

Möglicherweise ist dies jedoch in der Realität etwas komplizierter. Es ist natürlich klar, dass man ohne Partitionierung Verfügbarkeit und Konsistenz sicherstellen kann. Und wenn man gerade eine Partition hat, kann man sich dennoch in jedem Schritt überlegen, ob man lieber weiter Verfügbarkeit oder Konsistenz sicherstellen möchte. Üblicherweise ist es eigentlich immer möglich, Lesezugriffe weiter zuzulassen (bspw. DNS), ohne Konsistenz zu verletzen. In manchen Fällen ist es auch nicht nötig, globale Konsistenz sicherzustellen (bspw. regional beschränkt bei EC-Kartenkäufen).

Eine bei Facebook übliche Konsistenz ist *Read-your-own-writes consistency*. Dabei wird das System in ein Master- und mehrere Slave-Systeme unterteilt und Clients sind mit Slave-Systemen verbunden. Statt immer den Master instantan zu aktualisieren, wird zunächst der eigene Slave aktualisiert und der Master erhält die Änderungen erst später.

5.3.1. Partition Recovery

Dennoch lassen sich nicht alle Partitionen komplett mitigieren. Daher müssen Strategien gefunden werden, Partitionen zu erkennen und zu behandeln, sodass Konsistenz später wiederhergestellt werden kann.

Es ist schon nicht immer klar zu erkennen, ob eine Partition vorliegt (bspw. wenn darunterliegende IP-Layer kaputt sind, aber die Nodes sich indirekt noch sehen können). Außerdem können die Knoten die Partition zu verschiedenen Zeitpunkten erkennen. In diesen Fällen ist es anschließend notwenig, später wieder Konsistenz herstellen zu können. Dabei können Probleme bspw. bei der Erzeugung von IDs auftreten (bspw. wenn beide Partitionen die gleichen IDs für verschiedene Objekte erzeugen). Aber bei der Softwareentwicklung wird sich darauf verlassen, dass bestimmte Invarianten in einem konsistenten System gelten, bspw. dass solche automatisch erzeugten IDs immer eindeutig sind. Im Fall einer Partition muss also Konsistenz wiederhergestellt werden, bspw. indem Code die IDs nachträglich ändert.

Eine weitere Lösung sind konfliktfreie replizierte Datentypen (*CRDT*s). Beispielsweise serialisiert Google Docs Bearbeitungen in eine Liste von Insert und Delete Operationen. Hierbei können wieder Probleme auftreten, bspw. wenn beide Partitionen jeweils andere Teile der Information ändern, wobei das gemergte Ergebnis semantisch keinen Sinn mehr ergibt.

Ein generelles Problem bei Recovery ist aber weiter, dass es aufwändig, fehleranfällig und selten getestet ist. Prakisch strebt man immer irgendwie einen Kompromiss zwischen

den Zielen an. Bspw. könnte beim Ausfall der Verbindung zwischen Bankautomat und Bank noch ein gewisses (sicheres) Limit an Geld abgehoben werden.

5.3.2. CAP in SDN Infrastructures

Während sich verteilte Systeme vor allem in höheren Layern bewegen, ist SDN ein deutlich grundlegenderes System (Layer 2/3). Da ein funktionierendes Netzwerk essentiell für den Betrieb (und auch höhere Operationen) ist, muss dass Netzwerk in der Lage sein, sich selbst wiederherstellen zu können, nachdem Fehler/Ausfälle aufgetreten sind.

Bei SDN gibt es keine globale Antwort auf das System. Die Kernfrage ist dabei, ob das Netzwerk auch noch funktioniert, wenn das System nicht vollständig online ist. Die nächste Frage ist, ob irgendwo zirkuläre Abhängigkeiten existieren, die einen automatischen Kaltstart unmöglich machen. So kann bspw. eine funktionierendes SDN erforderlich sein, wenn die Steuerdaten des SDN in-band (also über das SDN) ausgetauscht werden. Außerdem können Probleme enstehen, wenn zwei Controller unterschiedliche Entscheidungen treffen, die z. B. zu Routing Loops führen können, wenn zwei Controller für einen Flow zwei verschiedene Routen festlegen.

5.4. OpenFlow

5.4.1. OpenFlow Components

Die Kernidee von **OpenFlow** ist ein Switch mit Input Queue, Output Queue und Forwarding-Tabellen. Statt jetzt jedoch die Forwarding-Tabelle mit Backward Learning zu managen, wird über die sog. *Southbound API Logic* mit einem OpenFlow-Controller gesprochen, der die Forwarding-Tabellen managed. In der Realität gibt es natürlich mehrere SDN-Controller, die die Forwarding-Tabellen auf den Switchen aktualisieren können. Der Management-Traffic zwischen Controllern und Switches läuft üblicherweise in einem Out-of-Band-Netz.

Statt eine Flow-Tabelle pro Switch zu führen, werden mehrere Flow-Tabellen kaskadiert. Dies ermöglicht bessere Flexibilität bei der Steuerung des Switches. Zusätzlich können Ports in Group Tables zusammengefasst werden, um Weiterleitungen auf Gruppen von Ports zu ermöglichen (bspw. für Multicast, Failover oder Load Balancing). Außerdem wird noch zwischen physischen Ports (die tatsächlich and der Hardware existieren) und logischen Ports (Repräsentatoren bspw. für VLANs, Tunnelprotokolle, LAG) unterschieden. So können Regeln erstellt werden, die Pakete an logische Ports schicken, um Traffic bspw. regelbasiert zu tunneln. Weiterhin gibt es noch weitere reservierte Ports, die Abstraktionen für bestimmte Aktionen im Switch (bspw. Senden an alle Egress Ports, Start der OF-Pipeline, Behandlung nach herkömmlicher Switching-Logik) repräsentieren.

Häufig haben Openflow-Switche auch eine traditionelle Switching-Logik, um bspw. bestimmte Traffic-Klassen „normal“ zu behandeln (z. B. Management-Traffic).

5.4.2. OpenFlow Pipeline Processing

Eingehende Pakete werden im Switch zunächst von Flow-Tabelle 0 bearbeitet werden, in den Regeln kann jetzt das Paket an einen oder mehrere Output Ports geleitet, das Paket manipuliert (z. B. Headerfelder geändert) oder auch die Auswertung weiterer Regeln durchgeführt werden. Nach Ausführung aller Aktionen geht das Paket möglicherweise noch an eine Gruppentabelle und wird an den Output Port geschickt, wo wiederum Flow Tabellen Regeln ausführen können (sofern in Hardware verfügbar), um zielspezifische Aktionen durchzuführen (z. B. NAT).[1]

Beim Matchen der Regeln werden immer alle Regeln einer Tabelle gematcht. Die Regeln können verschiedene Prioritäten haben, sodass die Regel mit höherer Priorität nach der Auswertung Vorrang bekommt. Es ist nicht klar spezifiziert, was passiert, wenn mehrere Regeln mit gleicher Priorität matchen. Für den Fall, dass es kein Match gibt, lassen sich Policies festlegen. Gematcht werden kann auf:

- Port (Ingress/Egress)
- Ethernet-Adressen
- EtherType
- IP-Protokoll
- IP-Adressen
- TCP/UDP Ports

Regeln können haben:

- Prioritäten
- Counter (werden beim Match hochgezählt, bspw. für statistische Erfassung)
- Instruktionen
- Timeouts (wann die Regel gelöscht werden soll)
- Cookies (Kennungen, bspw. um zu sehen, von wem die Regel ist)
- Flags (bspw. Benachrichtigung an den Controller, wenn eine Regel gelöscht wird)

Aktionen können sofort ausgeführt oder in einem Action Set für spätere Ausführung vorgemerkt werden. Letzteres ermöglicht es, bspw. einen Abbruch oder eine Weiterleitung an den Controller durchzuführen, wenn die zuvor ausgeführten Aktionen zu einem sinnlosen Ergebnis geführt haben. Aktionen sind:

- Pakete an bestimmte Ports schicken

[1] Die Anforderung für OpenFlow-Switche ist nur das Vorhandensein *einer* Flow-Tabelle im Ingress Processing. Es gibt nicht notwenigerweise mehrere Flow-Tabellen und auch nicht notwendigerweise Regelauswertung im Egress Processing.

- Pakete durch eine Gruppe verarbeiten
- Paket in eine Queue leiten
- Messen (z. B. für Traffic Shaping)
- Zusätzliche Tags (z. B. VLAN Header, MPLS-Label) hinzufügen oder entfernen
- Felder im Paket oder Metadaten setzen
- Felder kopieren
- TTLs ändern
- Pakete klonen

Counter können geführt werden für:

- Flow-Tabellen
- Flow-Einträge
- Ports
- Queues
- Gruppen
- Group Buckets
- Meter
- Meter Band

Gruppentabellen haben wieder einen Identifier und Regeln, wie Pakete verarbeitet werden sollen. So können Pakete an jeden Bucket in der Gruppe, Round Robin oder auch an den ersten verfügbaren Port geschickt werden, um bspw. Broadcast/Multicast, Fast Failover oder auch Load Balancing zu realisieren.

Meter haben einen Identifier in einer Meter Table und führen neben dem Counter auch Bands. Diese sind ein Typ, haben eine Rate, bestimmte Counter (z. B. für Thresholds) und typspezifische Argumente.

5.4.3. Controller and Control Channel

Zwischen Controller und Switch werden über einen Control Channel Nachrichten ausgetauscht. Dieser Channel läuft auf TCP-Basis und kann optional mit TLS abgesichert werden. Letzteres kann zwar Authentisierung und Vertraulichkeit sicherstellen, führt aber zu Verfügbarkeitsproblemen, da bspw. jetzt auch die Abhängigkeit zu funktionierenden RTCs und gültigen Zertifikaten besteht, die wiederum notwendig sein könnte, um abgelaufene Zertifikate aktualisieren zu können.

Über diesen Control Channel werden Nachrichten ausgetauscht, um Features der Switche zu ermitteln, Konfigurationen zu lesen oder zu aktualisieren, den aktuellen Zustand (Zählerwerte) auszulesen oder auch Flow-Tabellen zu modifizieren. Außerdem können Controller veranlassen, dass Switche Pakete zu erzeugen bzw. ein vom Controller generiertes Paket zu senden. So können bspw. DHCP-Anfragen zentral ohne Broadcasting beantwortet oder LLDP-Nachrichten erzeugt werden.

OpenFlow ist ein nachrichtenorientes Protokoll auf TCP-Basis. Daher kommen Nachrichten an Switchen in der richtigen Reihenfolge an. Allerdings gibt es keine Garantien, dass Befehle zu bestimmten Zeitpunkten schon fertig ausgeführt sind. Wenn jedoch ein nachfolgender Befehl die Ausführung eines vorigen Befehls bedingt, muss eine sogenannte *Barrier*-Nachricht, bei der der Switch jetzt wartet, bis alle Anfragen bearbeitet wurden. Weiterhin können über den Channel Nachrichten zwischen Controllern ausgetauscht werden.

Switche können Pakete an Controller weiterleiten, Controller über Änderungen an der Flow-Tabelle informieren, Änderungen am Port Status anzeigen, Controller über die Anmeldung eines neuen Master Controllers informieren und auch den Zustand eines Flows an Controller weiterleiten.

Hinzu kommen synchrone Statusnachrichten für Pings, Fehlermeldungen und andere Bookkeeping-Aufgaben.

5.4.4. OpenFlow Switches

Es gibt verschiedene Arten von Switches, die OpenFlow unterstützen. Dazu gibt es zunächst die typischen Commodity Hardware Switche, die von Herstellern angeboten werden und OpenFlow unterstützen. Ein Problem bei diesen Switchen ist Performance, da komplizierte Regeln auch viel Leistung zur Auswertung benötigen. So kann es sein, dass die Switche größer dimensioniert werden müssen als das Netz eigentlich verarbeiten muss. Für experientelle Aufbauten gibt es außerdem NetFPGA-basierte Hardware-Switches, die sich umprogrammieren lassen.

Außerdem gibt es auch Software-Switches mit OpenFlow unterstützung. Ein Beispiel hier für ist Open vSwitch. Dies ermöglicht es, VMs auf einem Server miteinander über virtuelle NICs miteinander zusammenzuschalten. Statt im Linux Kernel viele Bridges zu verwalten, wird ein Kernelmodul geladen, was nur einfache Forwarding-Regeln für Flows anwendet, ohne Pakete in den Userspace übertragen zu müssen. Pakete ohne zutreffende Regel hingegen werden im Userspace-Daemon weitergeleitet. Für die OpenFlow-Unterstützung, läuft auf einem Open vSwitch ein Server, der die Konfiguration hält und über ein Management-Protokoll den Open vSwitch konfiguriert.

5.5. SDN Controllers

Das Ökosystem um SDN-Controller ist sehr divers. Grundlegend haben diese jedoch immer als Schnittstellen ein Southbound Interface, um Switche zu konfigurieren (z. B. mit OpenFlow) und ein Northbound Interface, über welches Netzwerkanwendungen am Controller angebunden sind und über die Dienste im System gesteuert werden können. Für

die Kommunikation zwischen Controllern können Controller außerdem East-/Westbound Interfaces benutzt werden.

Im Controller laufen viele Module, die verschiedene Logik ausführen, um bspw. Flow-Regeln zu managen, Statistiken regelmäßig auszulesen, Sicherheitsmechanismen zu etablieren, etc. Die genaue Liste an Modulen und Features ist jedoch immer davon abhängig, welcher Controller benutzt wird.

5.6. Scalability and Resilience in SDN

Ein großes Probelem an SDN-Controllern ist, dass diese erst einmal zentral sind. Hauptargument ist, dass die Controller nur logisch zentralisiert sind, aber nicht physisch zentralisiert gebaut sein müssen. Eine Idee ist, verteilte Controller zu bauen. Das führt jedoch wieder zu weiteren Problemen. Hauptproblem ist zunächst die Kommunikation zwischen Controllern. Versuche, ein standardisiertes Protokoll für die East/Westbound-Kommunikation zu etablieren, waren nicht erfolgreich, da die Anwendungen im SDN Controller sehr divers sind.

Schaltet man mehrere SDN-Controller zusammen, ist es möglicherweise auch wünschenswert, dass die SDN Controller von einem anderen Controller zentral konfiguriert werden.

Zwischen autonomen System ist es auch möglich, statt die Router BGP sprechen zu lassen, könnten auch SDN-Controller der autonomen Systeme direkt miteinander verschaltet sein, um BGP direkt auszutauschen. So kann BGP als (stark eingeschränktes) East-/Westbound-Protokoll eingesetzt werden. Die BGP-Verbindung wird dabei über die Router getunnelt.

Erste Ansätze, SDN im Netzwerk einzusetzen, beruhten auf Ad hoc Entscheidungen, bei denen Verbindungen erst reaktiv aufgebaut wurden. Das hatte zu 30 000 Anfragen pro Sekunde geführt, was schnell zu Lastgrenzen geführt hat. Moderne Implementierungen schaffen mittlerweile deutlich höhere Raten. Dazu werden verschiedene Verfahren eingesetzt.

Ein erster Ansatz war, proaktiv Regeln für Flows einzutragen, noch bevor Flows auftreten. Weiter ist es möglich, Controller nah an Switches zu stellen, um Latenzen zu minimieren. Da Switch-CPUs bei SDN auch ein Bottleneck darstellen, werden manche Switches mittlerweile auch mit leistungsfähigen CPUs ausgestattet und die Pfade zum Controller direkt geswitcht statt über die CPU zu laufen.

Im Falle von Linkausfällen in einem echten (Layer 2) Netz muss das Netzwerk weiterhin funktionieren. In klassischen Netzen werden dazu Spanning-Tree-Protokolle eingesetzt, die im Fall eines Linkausfalls einen neuen Spannbaum über das (noch funktionierende) Netz berechnet. Das ist recht langsam. So lange jedoch in einem SDN-Netz die Verbindung zum Controller funktioniert, kann der Controller über den Linkausfall informiert werden, der dann die Flow-Regeln auf allen Switchen entsprechend anpasst. Eine (schnellere) Alternative dazu ist aber auch, ein Underlay-Netzwerk zu betreiben, in dem alle Pakete geroutet werden und in dem ein Routing-Protokoll auf Linkausfälle reagiert.

5.6.1. Kandoo

Bei verteilen SDNs mit mehrere Controllern wächst die Komplexität stark an. Bei Ausfällen müssen Controller sich einig werden, wie Entscheidungen getroffen werden. Dafür gibt es *Kandoo*, eine Architektur mit zwei Hiearchieebenen von Controllern. Switches reden mit lokalen Controllern und treffen möglichst viele Entscheidungen lokal. Nur „größere“ Entscheidungen werden vom Root Controller getroffen, auf dem Nicht-lokale Applikationen laufen und Entscheidungen treffen. Für Entwickler steigt hier wiederum die Komplexität dadurch, dass entschieden werden muss, welche Anwendungen wo laufen sollen. In manchen Einsatzfällen funktioniert dieser Ansatz, aber insbesondere größere Netzwerkoptimierungen sind damit nicht möglich. Beim Deployment solcher Setups stellt sich immer die Frage, ob die Anzahl Probleme, die lokal gelöst werden können, signifikant ist. Außerdem fragt sich, ob die Rechenleistung für die lokalen Controller überhaupt an einem geeigneten Standort untergebracht werden kann. Bei diesem Ansatz wurde allerdings auch nicht geklärt, wie Konsistenz erzielt werden soll. Außerdem leidet die Verfügbarkeit, da der Ausfall des Root Controllers wieder dazu führt, dass das SDN-Netz nicht richtig funktioniert.

5.6.2. DevoFlow

Im *DevoFlow*-Paper wurde gemessen, was physische Hardware kann und es wurde festgestellt, dass eigentlich vor allem die Top Talker interessant waren. Ziel war, OpenFlow so zuzubauen, dass solche Flows dies besser behandeln kann.

OpenFlow wurde im Paper um weitere Regeln erweitert. Regeln können jetzt Wildcards enthalten und ein CLONE Flag besitzen, welches dazu führt, dass beim Match eine neue Regel erzeugt, die besser matcht und mit der dann Counter geführt werden können. So kann anhand der Regeln erzeugt werden, welche Flows die meisten Pakete senden. Für Sampling wurden weiterhin Trigger eingebaut, sodass mit einer gewissen Wahrscheinlichkeit ein Paket eines Flows an den Controller geschickt wird. Damit sollten Elephant-Flows am Controller erkannt und behandelt werden.

Durch Erkennung von Elephant Flows können diese dann auf spezielle Pfaden im Netzwerk umgeleitet werden.

Mittels Netflow und IPFIX können jedoch auch Statistiken über Flows geführt werden. Mit sFlow werden auch bereits Sample-basiert Paketdaten an zentrale Stellen geschickt, um Elephant-Erkennung zu ermöglichen. Diese Protokolle werden bereits heute von allen Switchen unterstützt, was DevoFlow redundant erscheinen lässt.

Die Behandlung von Elephant Flows kann aber weitere Probleme erzeugen. So können diese möglicherweise auf anderen Pfaden Flows verdrängen oder Flapping verursachen. Das kann zu chaotischem Verhalten im Netzwerk führen, wodurch das Netz sehr instabil wird.

A. Markov-Prozesse [5]

Die folgende Einführung in Markov-Prozesse basiert vorrangig aus dem Material der Vorlesung „Telematik 2 / Leistungsbewertung“ [5].

A.1. Einführung

Markov-Prozesse sind stochastische Prozesse und ein sehr mächtiges Mittel in der Leistungsbewertung und Modellierung von Netzwerken. Innerhalb von Netzwerken gibt es dabei viele Knoten, die ein Warteschlangensystem beinhalten. Das Verhalten der Warteschlangen wird dabei maßgeblich davon beeinflusst, mit welcher Rate Pakete in die Warteschlange gelegt und wie schnell die Pakete in der Warteschlange abgearbeitet werden. Zur Analyse dieses Verhaltens werden solche Warteschlangensysteme als sog. *Markov-Ketten* modelliert. Dies erlaubt die Berechnung von Wahrscheinlichkeiten bestimmter Warteschlangenzustände (d. h. Füllstände der Warteschlangen) und damit auch die Ableitung von Leistungsmerkmalen der Systeme.

Ein stochastischer Prozess wird als **Markov-Prozess** bezeichnet, wenn er die **Markov-Eigenschaft** besitzt. Dies besagt, dass Ereignisse unabhängig sind, also das zukünftige Verhalten des Systems nach Zeitpunkt t nur vom Zustand $X(t)$ abhängt und *keinen* vorigen Zuständen $(X(\tau), \tau < t)$.

Dies hat zwangsläufig zur Konsequenz, dass die Inter-Arrival- und Servicezeiten exponentiell verteilt und die Ankunfts- und Serviceereignisse Poisson-verteilt sind.

A.2. Markov-Ketten

Markov-Ketten und Markov-Prozesse besitzen einen diskreten Zustandsraum (z. B. Anzahl Pakete in einer Queue) und einen stetigen Zeitraum (es gibt keinen festen Takt, in dem Pakete ankommen können, sondern dies kann jederzeit passieren). Das zukünftige Verhalten des Markov-Prozesses ist dabei vollständig durch den aktuellen Zustand beschreibbar.

Markov-Ketten werden gerne genutzt, um Zustandswahrscheinlichkeiten nach Erreichen einer *Steady Phase* zu analysieren. Es wird also angenommen, dass der Prozess ausreichend lang lief, sodass die Wahrscheinlichkeit, dass sich das System in einem bestimmten Zustand befindet, weder vom Ausgangszustand noch von der aktuellen Zeit abhängt. Ein solcher stationärer Zustand muss existieren, wenn Markov-Ketten

- **positiv rekurrent** sind (d. h. jeder Zustand ist in endlich viel Zeit erreichbar),

- **irreduzibel** sind (d. h. jeder Zustand ist von jedem anderen Zustand aus erreichbar) und
- **endlich** sind.

Solche Ketten und Zustände werden dann als **ergodisch** bezeichnet und sie erreichen nach einer transienten Phase ($t \to \infty$) einen Steady State.

A.3. Analyse einer Markov-Kette

Betrachte das einfache Bespiel einer Markov-Kette in Abbildung A.1. Dies könnte beispielsweise ein System aus zwei Queues mit zwei Paketen sein, die zwischen diesen Systemen hin- und hergeschoben werden. Es können also zu jedem Zeitpunkt diese zwei Pakete in verschiedenen Queues liegen (also entweder beide in der ersten Queue, beide in der zweiten Queue oder ein Paket in einer und eines in einer anderen Queue). Die Zustände repräsentieren hierbei alle Kombinationen von Füllständen der Queues, während μ_1 und μ_2 die Wahrscheinlichkeiten angeben, dass ein Zustandswechsel stattfinden. μ_1 ist dabei die Wahrscheinlichkeit, dass der erste Knoten ein Paket verschickt. Dies führt dazu, dass der zweite Knoten ein Paket mehr in der Queue hat, also einen Zustand weiter nach rechts gegangen wird. μ_2 ist umgekehrt die Wahrscheinlichkeit, dass der zweite Knoten ein Paket verschickt und in den Zustand weiter links gegangen wird. Zusätzlich gibt es auch für jeden Zustand die Wahrscheinlichkeit, dass sich der Zustand nicht ändert. Diese lässt sich durch Differenzbildung berechnen und ist nicht weiter aufgeführt.

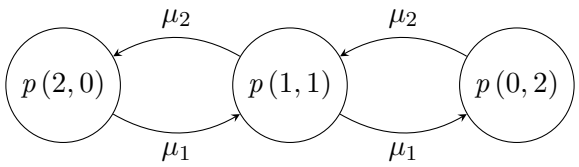

Abbildung A.1.: Einfaches Zustandsdiagramm eines Markov-Prozesses

Zur Analyse solcher Markov-Ketten können nun Kolmogorovs Gleichungssystem aufgestellt werden. Genutzt wird dabei, dass die Wahrscheinlichkeit, den Zustand zu betreten, gleich der Wahrscheinlichkeit ist, den Zustand zu verlassen. Dazu wird für jeden Zustand eine Gleichung aufgestellt, bei der von den Wahrscheinlichkeiten, von einem (anderen) Zustand in diesen Zustand zu wechseln, die Wahrscheinlichkeiten, diesen Zustand zu verlassen, subtrahiert werden. Dabei wird auch jeweils die Wahrscheinlichkeit, in einem entsprechenden Zustand zu sein, multipliziert:

$$\text{Zustand } (2,\,0): \quad p_{(1,1)} \cdot \mu_2 - p_{(2,0)} \cdot \mu_1 = 0 \tag{A.1}$$

$$\text{Zustand } (1,\,1): \quad p_{(2,0)} \cdot \mu_1 + p_{(0,2)} \cdot \mu_2 p_{(1,1)} \cdot (\mu_1 + \mu_2) = 0 \tag{A.2}$$

$$\text{Zustand } (0,\,2): \quad p_{(1,1)} \cdot \mu_1 - p_{(0,2)} \cdot \mu_2 = 0 \tag{A.3}$$

Zum Schluss wird noch ausgenutzt, dass sich das System jederzeit in einem der Zustände befinden muss, d. h. dass die Summe der Wahrscheinlichkeiten, in einem der Zustände zu sein, immer 1 sein muss:

$$p_{(2,0)} + p_{(1,1)} + p_{(0,2)} = 1 \tag{A.4}$$

Dieses lineare Gleichungssystem lässt sich lösen und ergibt die Wahrscheinlichkeiten, sich in einem der Zustände zu befinden. Diese Wahrscheinlichkeiten können dann bspw. genutzt werden, um zu ermitteln, wie ausgelastet die Knoten im Netzwerk sind (d. h. wie viel Prozent der Zeit sie arbeiten).

A.4. Eindimensionale Geburts- und Sterbeprozesse

Ein Markov-Prozess ist ein eindimensionaler Geburts- und Sterbeprozess, wenn nur Zustandsübergänge zwischen „benachbarten“ Zuständen möglich sind (z. B. wenn Zustände die möglichen Füllstände einer Warteschlange sind). Die zwei Wahrscheinlichkeiten eines solchen Prozesses sind:

Geburtswahrscheinlichkeit λ_k, die die Wahrscheinlichkeit angibt, von Zustand k nach $k+1$ überzugehen.

Sterberate μ_k, die die Wahrscheinlichkeit angibt, vom Zustand $k{+}1$ nach k überzugehen.

Hierfür gibt es dann die angepasste Chapman-Kolmogorov-Gleichung:

$$\frac{\mathrm{d}}{\mathrm{d}t} p_k(t) = \underbrace{p_{k+1}(t)\,\mu_{k+1} + p_{k-1}(t)\,\lambda_{k-1}}_{\text{Transitions to state } k} - \underbrace{p_k(t)\cdot(\lambda_k + \mu_k)}_{\text{Transitions from state } k} \qquad \text{Für } k > 0 \tag{A.5}$$

$$\frac{\mathrm{d}}{\mathrm{d}t} p_0(t) = p_1(t)\,\mu_1 - p_0(t)\,\lambda_0 \qquad \text{Für } k = 0 \tag{A.6}$$

Normalisierungs-Bedingung:

$$\sum_{k \in Z} p_k(t) = 1 \tag{A.7}$$

In einem solchen eindimensionalen Geburts- und Sterbeprozess existiert globales Gleichgewicht für Zustand k, wenn Übergänge von und zu benachbarten Zuständen im Gleichgewicht sind:

$$0 = p_{k+1}\mu_{k+1} + p_{k-1}\lambda_{k-1} - p_k(\lambda_k + \mu_k) \qquad k > 0 \tag{A.8}$$

$$0 = p_1\mu_1 - p_0\lambda_0 \qquad k = 0 \tag{A.9}$$

$$\sum_{k \in Z} p_k = 0 \tag{A.10}$$

Im Steady State ergibt sich bei globalem Gleichgewicht die Wahrscheinlichkeit, dass sich das System im Zustand k befindet, als:

$$p_k = p_0 \cdot \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \qquad k > 0 \tag{A.11}$$

A.5. Das M/M/1-System

Das M/M/1-Queueing-System modelliert einen einzelnen Knoten in einem Netzwerk. Dabei wird eine unendlich große Queue angenommen (also auch unendlich viele Zustände). Die ist zwar in der Praxis nicht der Fall, macht jedoch die Analyse einfacher. In der Praxis würde eine zu volle Queue jedoch zu Paketverlust führen. Somit kann eine in der Theorie sehr große Queue ein Indikator dafür sein, dass sich ein entsprechendes System in der Praxis sehr instabil verhalten würde.

Das M/M/1-System ist ein eindimensionaler Geburts- und Sterbeprozess, bei dem $\lambda = \lambda_0 = \lambda_1 = \ldots$ und $\mu = \mu_0 = \mu_1 = \ldots$, es also vom Zustand unabhängige, unveränderliche Geburts- und Sterberaten gibt. Die Geburtsrate wird hier als „Ankunftsrate“ bezeichnet, die in der Praxis der Ankunft eines Paketes in der Queue entspricht. Die Sterberate ist die Servicerrate/Bedienrate und gibt an, wie schnell das System Pakete aus der Queue verschicken kann.

Die Last des Systems wird als $\rho := \frac{\lambda}{\mu}$ definiert und $\rho < 1$ muss gelten, damit das System einen Steady State erzielen kann. Für die Wahrscheinlichkeiten p_k gilt dann:

$$p_0 = 1 - \frac{\lambda}{\mu} = 1 - \rho \tag{A.12}$$

$$p_k = p_0 \cdot \left(\frac{\lambda}{\mu}\right)^k \qquad k > 0 \tag{A.13}$$

$$= (1 - \rho)\,\rho^k \tag{A.14}$$

Dies ist die Wahrscheinlichkeit, dass k Pakete im System sind. Da $\rho < 1$, ist diese für steigende k monoton fallend.

Weiter lässt sich die durchschnittliche Anzahl Pakete k im System ermitteln:

$$k = \frac{\rho}{1 - \rho} \tag{A.15}$$

Die durchschnittliche Anzahl Pakete L in der Warteschlange ist:

$$L = \frac{\rho^2}{1 - \rho} \tag{A.16}$$

Beide Werte steigen für wachsendes ρ stark an. Eine hohe Last führt also zu vollen Warteschlangen und irgendwann auch zu Überlast (sprich: Paketverlust). Für die durchschnittliche Antwortzeit T_v und Wartezeit T_w gilt:

$$T_v = \frac{1}{\mu - \lambda} \tag{A.17}$$

$$T_w = \frac{1}{\lambda} \cdot \frac{\rho^2}{1 - \rho} \tag{A.18}$$

Auch hier führt eine hohe Last dazu, dass T_v und T_w stark ansteigen.

B. Buzzword Of The Day

In den Vorlesungen wird ein „Buzzword“ bzw. Begriff aus der Welt der Netzwerktechnik genannt. Dieser Begriff ist von den Hörenden zu recherchieren und wird in der anschließenden Vorlesung diskutiert. Hier sind einige Begriffe samt Diskussionen und zusammengetragen.

B.1. Cut-Through Switching [1, 6]

Cut-Through Switching vermeidet Latenz, indem ein Paket bereits an den ausgenden Port verschickt wird, sobald die Zieladresse und der ausgehende Port bestimmt werden konnten. Das Paket kann also weitergeleitet werden, noch bevor es vollständig empfangen wurde.

Cut-Through Switching erfordert, dass die Linkgeschwindigkeit des ausgehenden Ports mindestens so groß ist wie die des eingehenden Ports, da sonst das Paket wieder gepuffert werden muss. Ein weiteres Problem ergibt sich dadurch, dass die Checksumme des Ethernet Frames nicht geprüft werden kann, bevor das Paket verschickt wird (einerseits, da die Payload zu diesem Zeitpunkt noch nicht ganz bekannt ist, andererseits, da die Checksumme erst am Ende des Ethernet Frames steht). Daher leitet ein Cut-Through Switch auch beschädigte Frames weiter, die ein Store-and-Forward Switch verwerfen würde.

Cut-Through Switching kann auch in einer adaptiven Variante betrieben werden. In diesem Fall wird es nur benutzt, solange Fehler am Link nur selten passieren. Steigt hingegen die Fehlerrate am Port, wird auf Store and Forward umgeschaltet, um unnötiges Weiterleiten von Paketen zu vermeiden.

Eine andere Erweiterung ist Fragment-Free-Cut-Through, bei der nur die ersten 64 Bytes eines Ethernet Frames geprüft werden. Dies beruht auf der Erfahrung, dass Framefehler meistens nur innerhalb der ersten 64 Byte auftreten.

In einem RZ-Setup ist üblicherweise für jedes Rack ein Top-Of-Rack-Switch verbaut, an dem die Server des Racks angeschlossen sind und der das Rack mit dem Backbone verbindet. Das Problem dabei ist, dass Pakete fast immer im Top-Of-Rack-Switch gepuffert werden müssen, weil sie von den Servern zeitgleich Pakete empfangen. In so einem Fall bringt Cut-Through also nichts. Änlich bringen die Latengewinne im Nanosekundenbereich selbst für latenzkritische Anwendungen (wie bspw. PTP) nicht viel, da die Latenzen an den Endsystemen meist deutlich größer sind und somit die Gewinne durch Cut-Through-Switching zunichte machen.

B.2. Hairpin Turn

Es ist teilweise notwendig (z. B. bei VLAN-Routing, Routing im IX), dass ein Router ein Paket an den gleichen Port zurückzuschicken, an dem das Paket ankam. Bei Ethernet ist dies nicht möglich, da sonst durch Backward Learning das Paket im Kreis geschickt wird (wird zwischen zwei Switchen direkt hin und hergeschickt). Schaltet man jedoch Backward Learning aus, kann ein eingehendes Paket auch bei Ethernet direkt zurückgeschickt werden. Dies kann beispielsweise im Cloud-Umfeld genutzt werden, um Traffic-Policies, ACLs, u. Ä. auf Switch-Ebene auch für Traffic durchzusetzen, der zwischen zwei VMs auf dem gleichen VM-Host läuft.

Eine klare Lektion hierbei ist, dass ein Switch nicht nur von einem Port an andere (vom Input Port ungleiche) Ports sendet, sondern auch wieder zurück an den Port selbst.

B.3. Explicit Congestion Notification [3]

Explicit Congestion Notification (ECN) ermöglicht es Routern und Switchen, den Endpunkten einer Transportverbindung zu signalisieren, wenn eine Queue Congestion erfährt. Dazu setzen Endpunkte ein Bit in den zwei für ECN vorgesehenen Bits im IP-Header. Erfährt nun ein Router/Switch Congestion, werden beide Bits gesetzt, um Congestion anzuzeigen. Die Gegenseite erkennt diese Benachrichtigung und gibt sie in ihrer Antwort an den Sender (z. B. im TCP ACK als Flag) wider. Der Sender kann daraufhin Maßnahmen ergreifen, um Congestion vorzubeugen.

ECN erfordert Unterstützung sowohl an beiden Endpunkten der Verbindung als auch bei den Geräten auf dem Pfad zwischen beiden Endpunkten. Zwar ist es prinzipiell möglich, dass innere Knoten des Pfades kein ECN unterstützen, jedoch dürfen diese Pakete mit gesetzen ECN-Bits nicht verwerfen und auch ihre Routing-Entscheidungen dadurch nicht verändern (was möglich ist, wenn diese Bits anders interpretiert werden). Außerdem funktioniert ECN nur, wenn die inneren Knoten des Pfades auch tatsächlich bei bevorstehender Congestion (welche i. d. R. durch einen AQM-Algorithmus erkannt werden muss) die beiden ECN-Bits setzen. Passiert dies nicht, muss Congestion weiterhin über herkömmliche Methoden wie Packet Loss erkannt werden, was je nach Situation zu größeren Verzögerungen bei der Erkennung von Congestion führt (z. B. bei Tail Drop am Ende eines Bursts) signifikante Verschlechterungen der Verbindungsqualität zur Folge hat (z. B. bei unzuverlässiger Videoübertragung über RTP, bei denen es keine Retransmissions gibt).

Indem ECN den Sender über Congestion informiert, bevor Pakete gedroppt werden müssen, kann das Sendeverhalten (die Senderate) angepasst werden, ohne Daten zu verlieren. Gerade bei unzuverlässigen Übertragungen wie z. B. Echtzeit-Video kann dies die Verbindungsqualität verbessern.

Ein Problem bei der Einführung von ECN ist Abwärtskompatibilität. Die für ECN genutzten Bits des IP-Headers waren zwar zuvor reservierte Bits, wurden jedoch auch schon für andere Zwecke (z. B. Routing-Entscheidungen) genutzt. Dies bedeutet, dass ein Router gesetzte ECN-Bits fehlinterpretieren kann und bspw. Pakete mit gesetzten

ECN-Bits anders routet. Dies wiederum kann zu Reordering führen, was die Performance der Verbindung einschränkt. ECN ist außerdem nur dann effektiv, wenn die Knoten im Pfad, die Congestion erfahren, auch ECN-fähig sind.

Das andere Problem bei der Einführung von ECN ist, dass AQM quasi Pflicht für Knoten ist. Ohne AQM wird auch keine Entscheidung getroffen, wann der Knoten Congestion erfährt und wann er eine ECN versenden soll. Somit kann insbesondere ältere Hardware oder Hardware, die normales FIFO-Queueing mit Tail-Drop benutzt, keine ECN versenden.

Zuletzt können Probleme entstehen, wenn einigen Knoten ECN-fähig sind und deren AQM-Algorithmus die ECN-Bits setzt, statt Pakete zu droppen, ein nachfolgender Knoten aber (aufgrund irgendwelcher Policies zur Bereinigung von Headern) die ECN-Bits auf 0 setzt (sprich, ECN ausschaltet). Dann würde der Sender sein Sendeverhalten nicht anpassen, was möglicherweise zu Problemen im AQM-Algorithmus führen kann.

B.4. L4S RFC 9330

B.5. Datenblatt erklären

B.6. Intent Based Networking

Bei SDN führt die Hardware einfache Kommandos aus, die von einem Controller konfiguriert werden. Bei Intent Based Networking soll der Programmierer gegenüber dem IBN-System nur seine Absichten (z. B. „Computer A soll möglichst schnell mit Computer B reden“) und das Netzwerk konfiguriert sich selbst automatisch so, dass dies umgesetzt werden. Diese Technik klappt allerdings nur so semi-gut.

B.7. Zero Trust [8, 7, 2]

Zero Trust ist ein Architekturprinzip, bei dem die Grundannahme ist, dass kein Teil des Netzes als vertrauenswürdig angesehen wird. So wird versucht, dass jeder Kommunikationsvorgang authentifiziert und verschlüsselt ist, um Vertraulichkeit und Integrität sicherzustellen. Dies hat eine bessere Angriffsprävention zum Ziel, da Angreifern sehr viele und hohe Hürden gelegt werden sollen, die einen Angriff unwahrscheinlicher machen.

Weitere Implikationen aus dem Zero-Trust-Prinzip erfordern, dass alle Geräte, die im Netzwerk mitwirken, zunächst klassifiziert und ihre Schlüssel freigeschaltet werden müssen. Für jedes Gerät muss festgelegt werden, welche (minimalen) Berechtigungen das Gerät erhält, um gerade die Dinge im Netzwerk durchführen zu können, die es gemäß Klassifikation des Administrators können soll.

B.8. SD-WAN

ob das auch was mit security zu tun hat, und was?

Stichwortverzeichnis

Literatur

[1] *Cut-through switching*. In: *Wikipedia*. Page Version ID: 1205281012. 9. Feb. 2024. URL: https://en.wikipedia.org/w/index.php?title=Cut-through_switching&oldid=1205281012 (besucht am 08. 04. 2024).

[2] Dr Datenschutz. *Zero Trust – Das Sicherheitsmodell einfach erklärt*. Dr. Datenschutz. URL: https://www.dr-datenschutz.de/zero-trust-das-sicherheitsmodell-einfach-erklaert/ (besucht am 30. 05. 2024).

[3] Gorry Fairhurst und Michael Welzl. *RFC 8087: The Benefits of Using Explicit Congestion Notification (ECN)*. IETF Datatracker. Section: IETF - Internet Engineering Task Force. 2017. URL: https://datatracker.ietf.org/doc/html/rfc8087 (besucht am 28. 04. 2024).

[4] Michael Roßberg. "Advanced Networking Technologies". Vorlesung. Vorlesung. 2024.

[5] Michael Roßberg. "Telematik 2 / Leistungsbewertung". 2023.

[6] *Switching*. Elektronik-Kompendium. URL: https://www.elektronik-kompendium.de/sites/net/0907141.htm (besucht am 08. 04. 2024).

[7] *Zero Trust*. Bundesamt für Sicherheit in der Informationstechnik. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Zero-Trust/zero-trust.html?nn=1091870 (besucht am 30. 05. 2024).

[8] *Zero Trust Security*. In: *Wikipedia*. Page Version ID: 242568663. 25. Feb. 2024. URL: https://de.wikipedia.org/w/index.php?title=Zero_Trust_Security&oldid=242568663 (besucht am 30. 05. 2024).