HUMBOLDT-UNIVERSITÄT ZU BERLIN

Peer-to-Peer-Systeme

Teil IV: Erste DHTs – CAN und Chord

Björn Scheuermann

Humboldt-Universität zu Berlin
Wintersemester 2015/16

Suche in Gnutella

- Gnutella sucht durch Fluten im Netzwerk, also mit sehr hohem Aufwand
- Um vorhandene Daten sicher zu finden, muss das *gesamte* Netzwerk durchsucht werden

$\Rightarrow$ Sehr ineffizient, skaliert nicht!

- Grund: Die Peers haben keinerlei Information darüber, wo bzw. in welcher „Richtung“ sich ein gesuchtes Datenelement befindet

Wie können wir Overlay-Netzwerke bauen, in denen vorhandene Daten gezielt und effizient gefunden werden können?

Hashtabellen

Zur Erinnerung: Hashtabellen als Datenstruktur für das schnelle Wiederfinden von Informationen

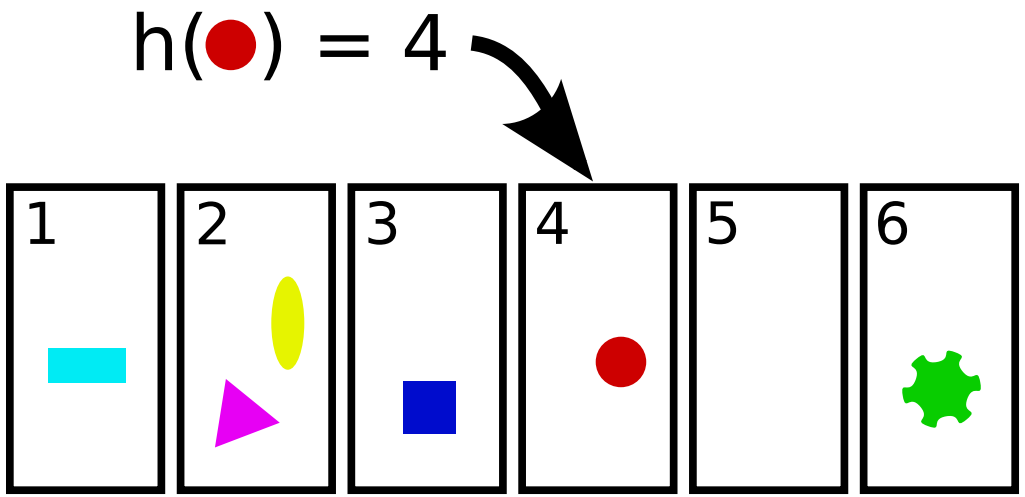
- Datenelemente haben einen *Schlüssel* (Dateiname,. . .)
- Mithilfe einer *Hashfunktion* werden Schlüssel auf Hashwerte fester Länge (i. d. R. Zahlen bzw. Bitstrings) abgebildet

$$h : K \to Q$$

- Normalerweise ist $|K| \gg |Q|$
- Idealerweise ist bei Hashfunktionen die Ausgabe so weit wie möglich unabhängig von der Eingabe
- Normalerweise wird aus Anwendungsperspektive angenommen, dass jedem Schlüssel $x \in K$ ein quasi-zufälliger Hashwert $h(x) \in Q$ zugeordnet wird

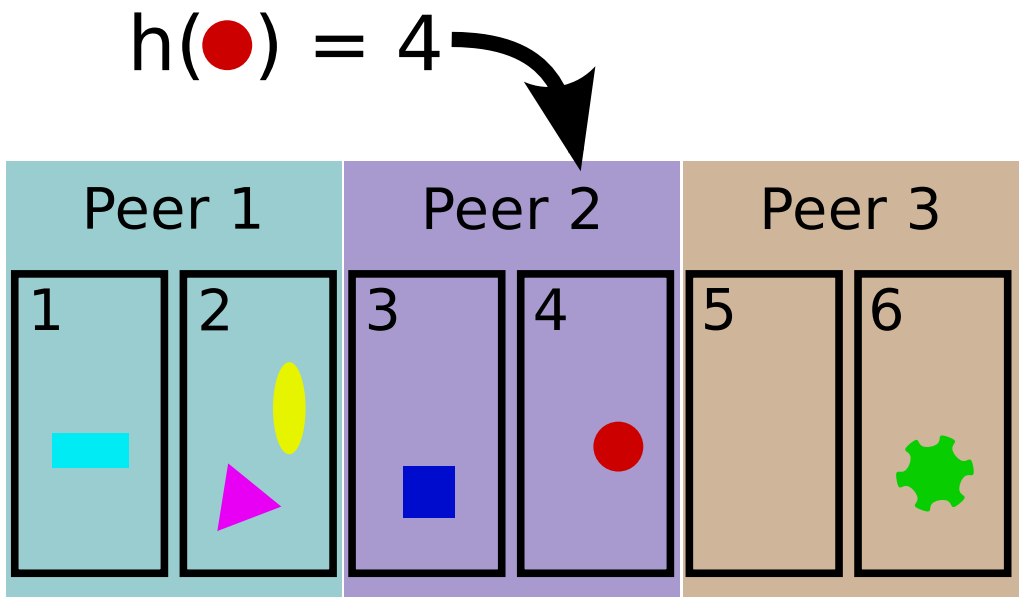
Hashtabellen

- In einer Hashtabelle werden Datenelemente in einem durch den Hashwert ihres Schlüssels festgelegten „Fach“ abgelegt
- Das richtige „Fach“ kann so in einem Schritt ($O(1)$) gefunden werden
- Bietet also ein einfaches Interface, um Informationen über Schlüssel-Wert-Paare zu verwalten



Verteilte Hashtabellen

- Eine verteilte Hashtabelle (Distributed Hash Table, DHT) wendet das Schlüssel-Wert-Prinzip an, um die Informationen systematisch auf die Peers in einem Overlay zu verteilen
- Jeder Peer ist für einen Teil des Hash-Wertebereichs zuständig



Interface einer DHT

Eine verteilte Hashtabelle bietet einer darauf aufsetzenden Anwendung folgende Funktionalität:

- Ablegen von Schlüssel-Wert-Paaren auf einem für den Schlüssel „zuständigen“ Peer im Overlay („put“)
- (Effiziente) Suche nach einem Schlüssel („get“)

Dies wird ermöglicht, indem die Peers im Overlay systematisch so verbunden werden, dass effizientes Routing zu dem für einen gegebenen Schlüssel zuständigen Peer möglich ist.

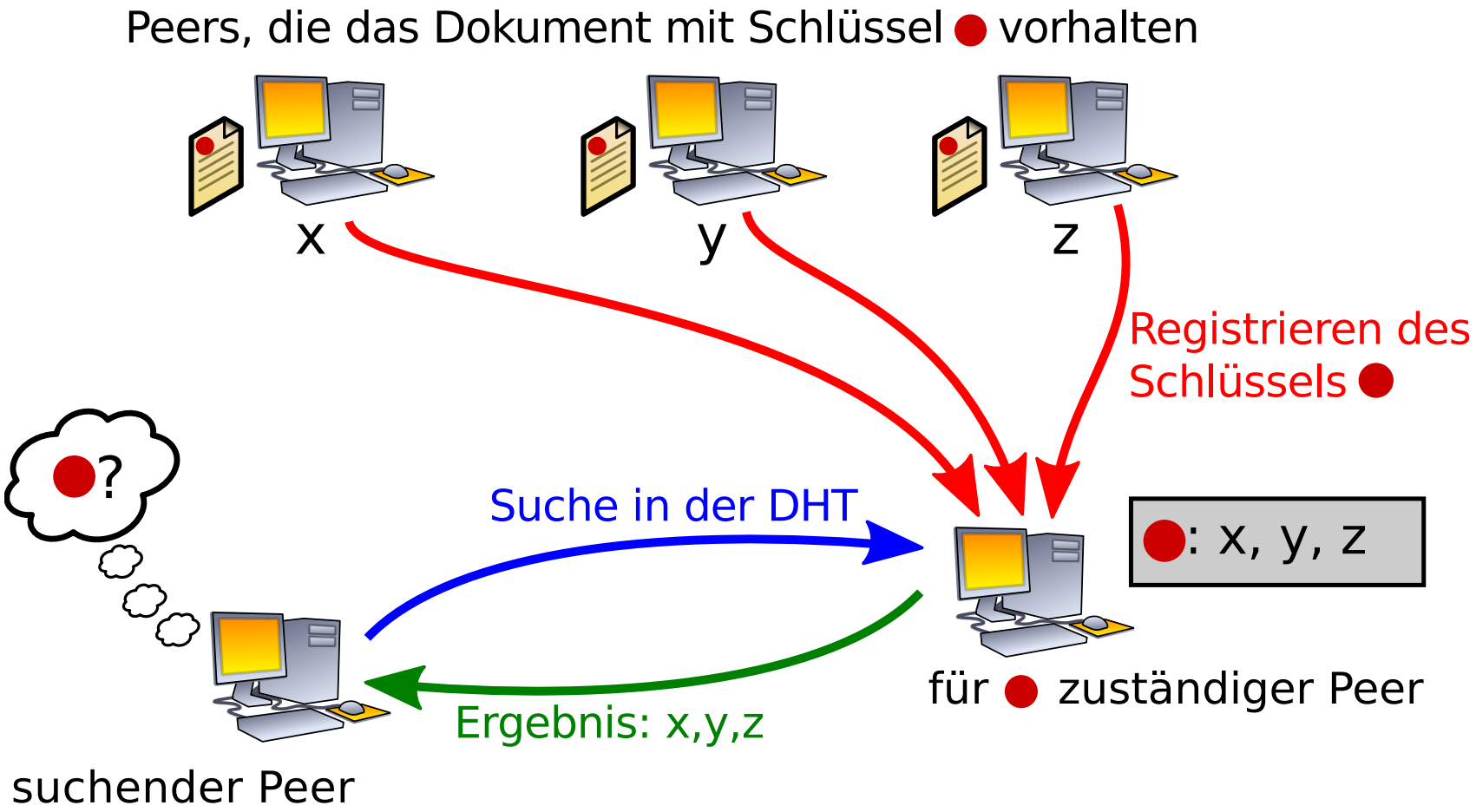
Wir betrachten nun also *strukturierte* Overlays

Welche Vor- und Nachteile einer DHT erwartet ihr im Vergleich zu einem unstrukturierten (Gnutella-artigen) Overlay?

Schlüssel-Indirektion

- Beim Beitritt zu einer DHT wird einem Peer die Zuständigkeit für einen Teil des Schlüsselraums übertragen
- Wichtig: Das bedeutet *nicht* zwangsläufig, dass alle Daten auf diesen Peer übertragen werden müssen
- Bei großen Datenelementen (z. B. Dateien beim Filesharing, Schlüssel = Dateiname, Daten = Dateiinhalt) ist es nicht sinnvoll, die Dateien auf den zuständigen Peer zu kopieren!
- Auf dem für einen Schlüssel zuständigen Peer kann stattdessen die Liste der Peers abgelegt werden, die die Daten bereithalten (das ist analog zum Suchindex in Napster!)

Schlüssel-Indirektion



Herausforderungen einer DHT

- Fragen:
  - Wie werden die Schlüssel auf die Peers verteilt?
  - Mit welchen anderen Peers sollte ein Peer direkt verbunden sein?
  - Wie kann im Overlay gezielt zu einem bestimmten Schlüssel bzw. Hashwert geroutet werden?
- Besondere Herausforderungen dabei:
  - Man möchte mit wenig Zustandsinformation (insbes.: wenige Nachbarn) in jedem Peer auskommen
  - Wenn Peers kommen und gehen (oder sogar unerwartet ausfallen!) müssen die notwendigen „Umbauten“ lokal beschränkt bleiben

Die ersten DHTs

Wir betrachten in dieser Vorlesung zunächst die beiden ersten DHTs:

- CAN (Content Addressable Network)

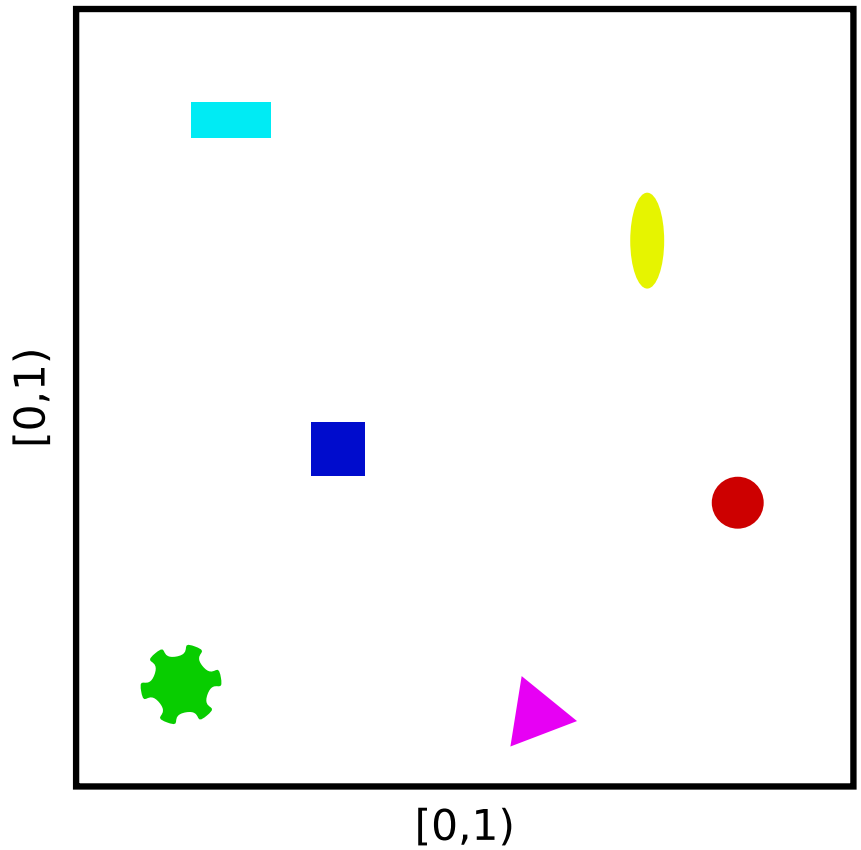
  [Ratnasamy, Francis, Handley, Karp, Shenker: A Scalable Content-Addressable Network, SIGCOMM 2001]

- Chord

  [Stoica, Morris, Liben-Nowell, Karger, Kaashoek, Dabek, Balakrishnan: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications, Transactions on Networking, 2003]

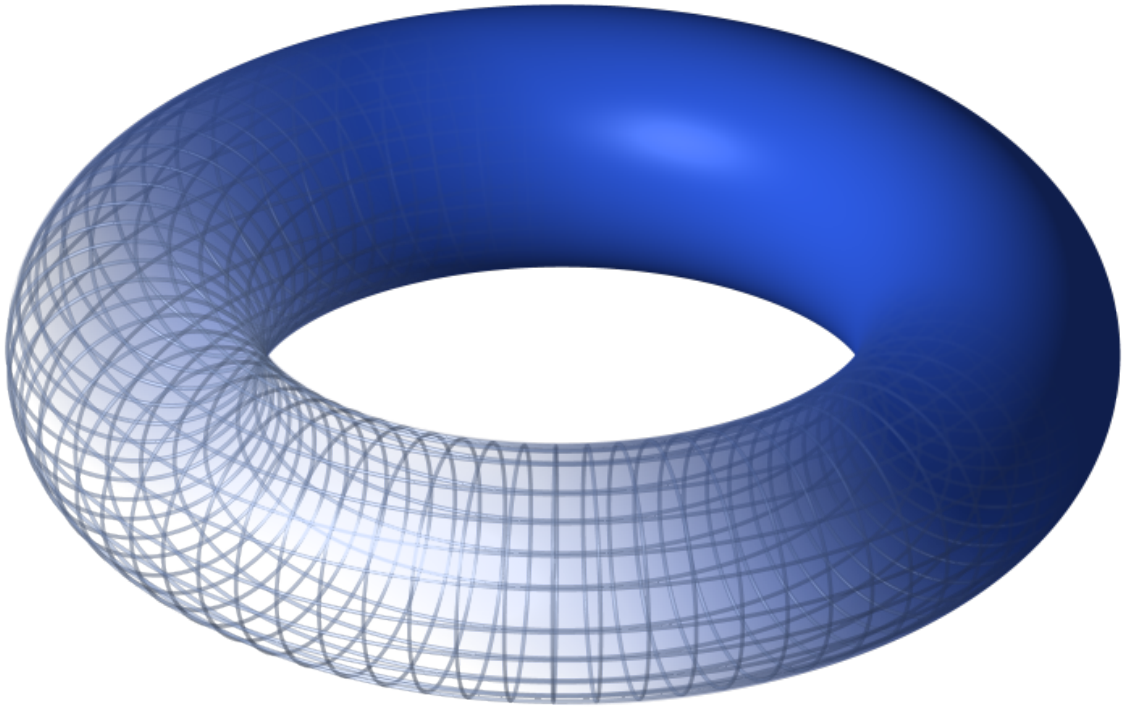
Zweidimensionaler Schlüsselraum in CAN

- In CAN werden die Schlüssel auf Punkte im Quadrat $[0, 1) \times [0, 1)$ abgebildet
- Verwende hierfür zwei Hashfunktionen $K \to [0, 1)$



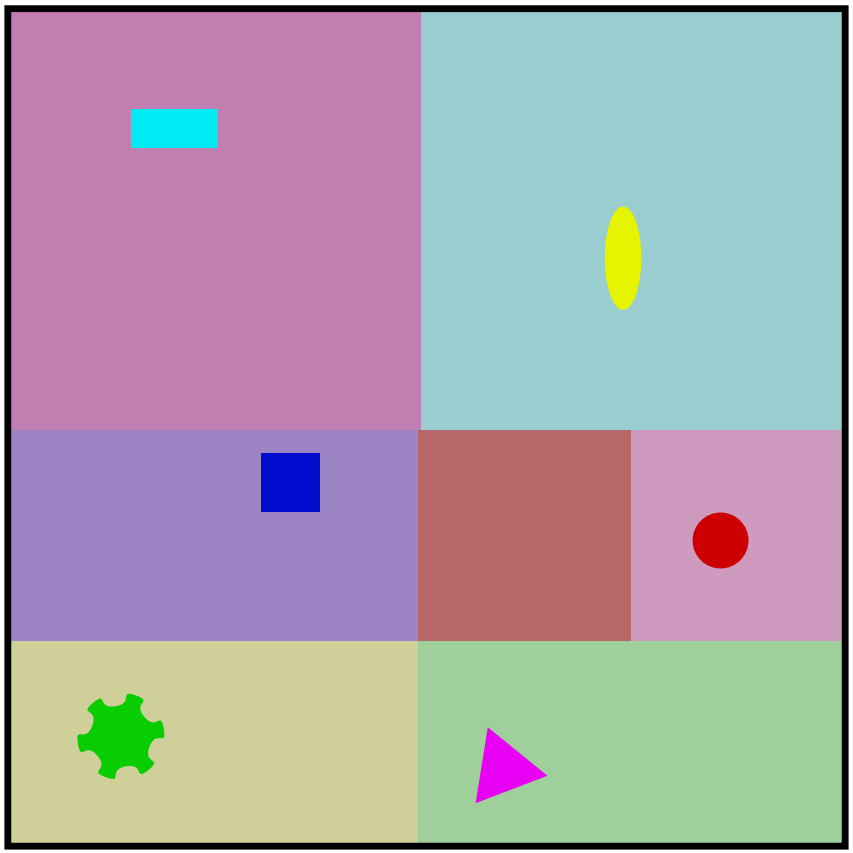
Wraparound: Torus

- Der Schlüsselraum wird als Torus aufgefasst (Wraparound)



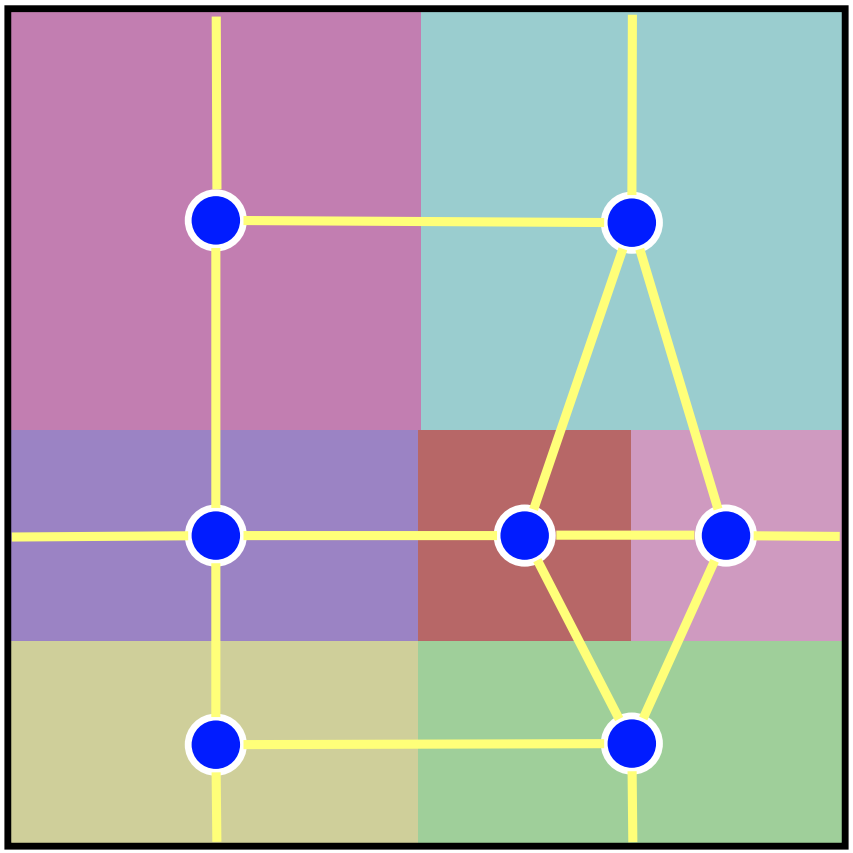
Aufteilung der Schlüssel auf die Peers

- Der Schlüsselraum wird in rechteckige Bereiche aufgeteilt
- Für jeden dieser Bereiche ist jeweils ein Peer zuständig
- Ein Peer verwaltet die Einträge der Schlüssel, die in seinen Zuständigkeitsbereich fallen



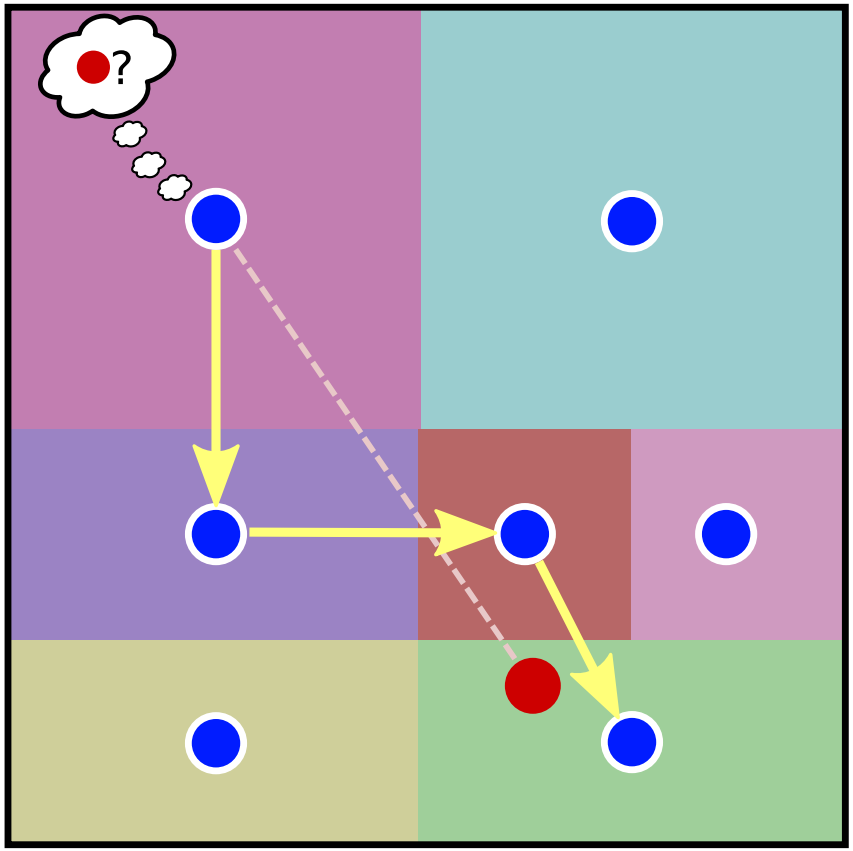
Verbindungen im CAN-Overlay

- Ein Peer hält Verbindungen zu allen Peers mit direkt angrenzendem Zuständigkeitsbereich (gemeinsame Kante)
- Beachte Wraparound-Kanten (Torus!)



Suche im CAN-Overlay

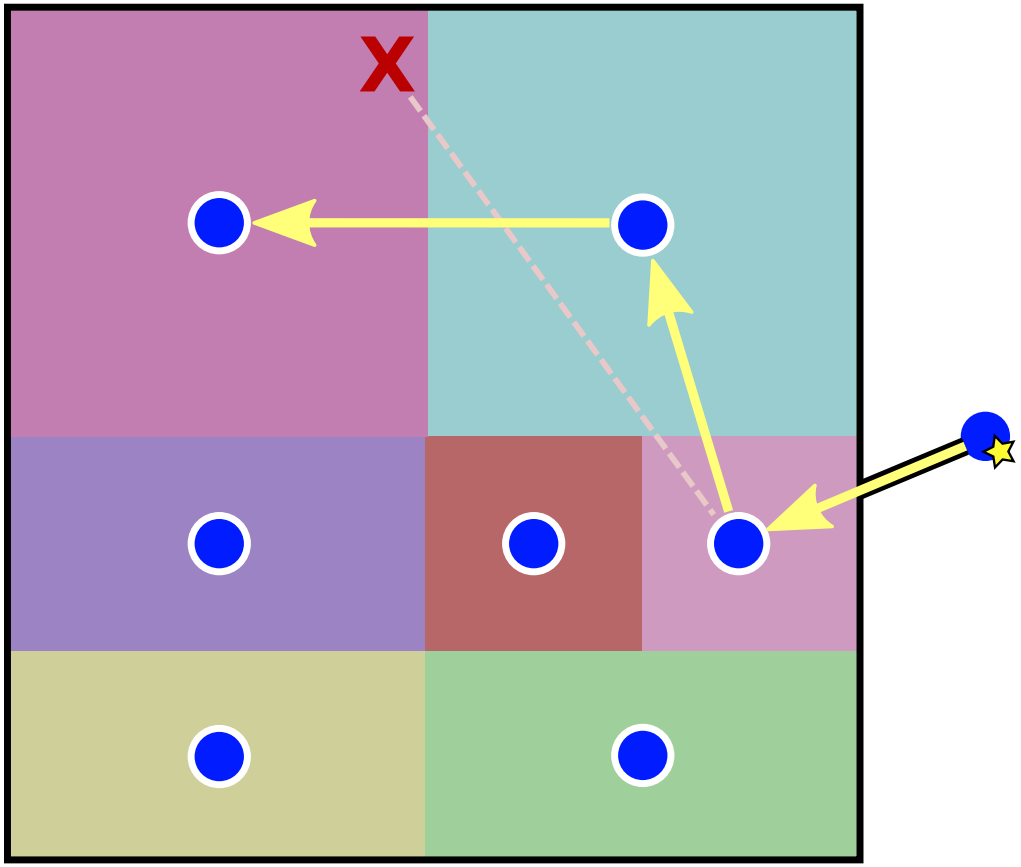
- Zur Suche nach einem Schlüssel wird die Anfrage in Richtung seiner Koordinaten weitergereicht



- Zum Routing genügt also für alle Nachbarn ein Tupel (IP-Adresse, Port, Zuständigkeitsbereich)

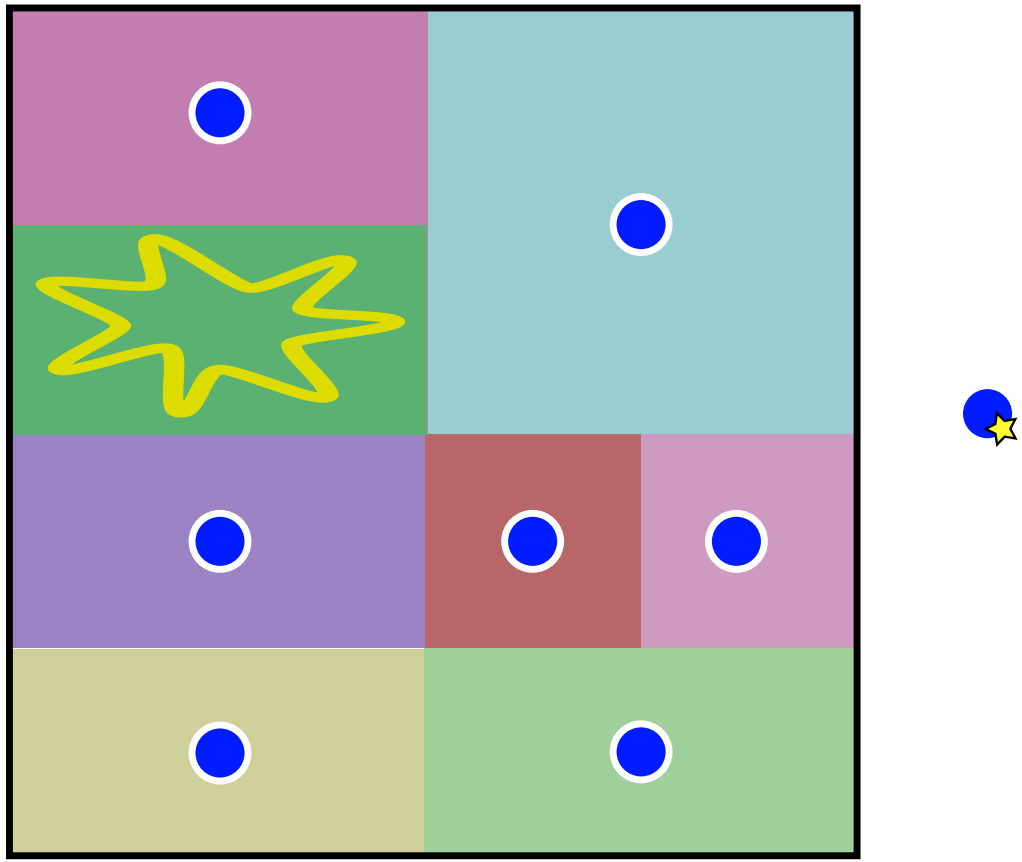
Einfügen eines Peers

1. Neuer Peer kontaktiert einen existierenden Peer
2. Wählt dann einen zufälligen Punkt X aus $[0, 1) \times [0, 1)$
3. Suche nach dem für X zuständigen Peer



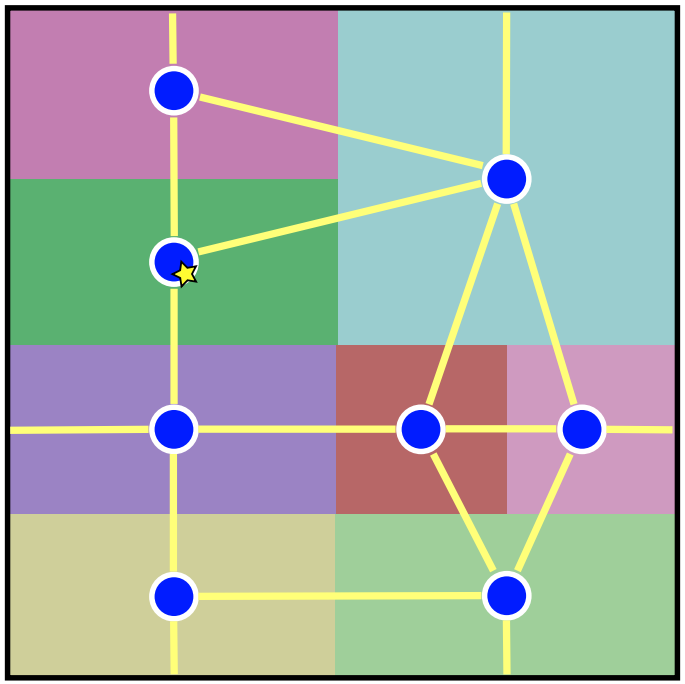
Einfügen eines Peers

4. Zone dieses Peers wird in der Mitte geteilt
5. Für das Teilen gibt es eine festgelegte Reihenfolge der Dimensionen (z. B.: zuerst wird entlang der x-Achse geteilt, dann entlang der y-Achse)



Einfügen eines Peers

6. Einträge aus der übergebenen Hälfte des Zuständigkeitsbereiches werden übertragen
7. Informationen über Nachbarn werden übertragen
8. Den Nachbarn werden die Veränderungen mitgeteilt



Der Peer ist nun vollständig eingebunden, Änderungen im Overlay traten nur lokal auf.

Fairness der Aufteilung

Betrachte die Wahrscheinlichkeit, dass ein Rechteck der Fläche A nach dem Einfügen von n weiteren Peers noch nicht weiter unterteilt wurde.

Diese Wahrscheinlichkeit ist $(1-A)^n \leq e^{-nA}$, also exponentiell klein

Beweis: Beim Einfügen eines Peers ist die Wahrscheinlichkeit, dass ein *anderes* Rechteck gewählt wird, $1-A$. Dass n Peers *alle* ein anderes Rechteck wählen, geschieht mit Wahrscheinlichkeit $(1-A)^n$. Da

$$\forall m > 0 : \left(1 - \frac{1}{m}\right)^m \leq \frac{1}{e},$$

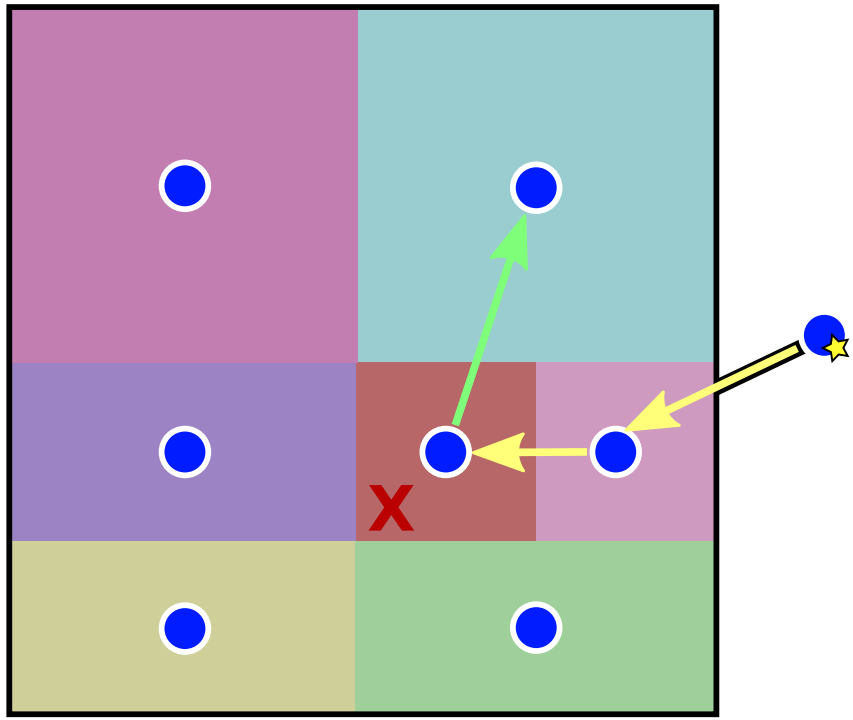
gilt:

$$(1-A)^n = \left((1-A)^{\frac{1}{A}}\right)^{nA} \leq e^{-nA}.$$

Unverhältnismäßig große Gebiete sind also unwahrscheinlich

Volume Balancing

- Die Fairness der Aufteilung kann durch *Volume Balancing* noch weiter verbessert werden
- Beim Einfügen wird der neue Peer wird an Nachbarn mit einer größeren Zone „weitergereicht“, bis ein (lokales) Optimum gefunden wurde



Größe der Zustandsinformation und Routenlänge

- CAN wurde hier im Zweidimensionalen erklärt, lässt sich aber auf einen d-dimensionalen Raum völlig analog erweitern
- Menge der Zustandsinformation für das Routing in jedem Peer (= Anzahl der Nachbarn im Overlay) im Durchschnitt:

$$O(d)$$

- Durchschnittliche Pfadlänge:

$$O(dn^{1/d})$$

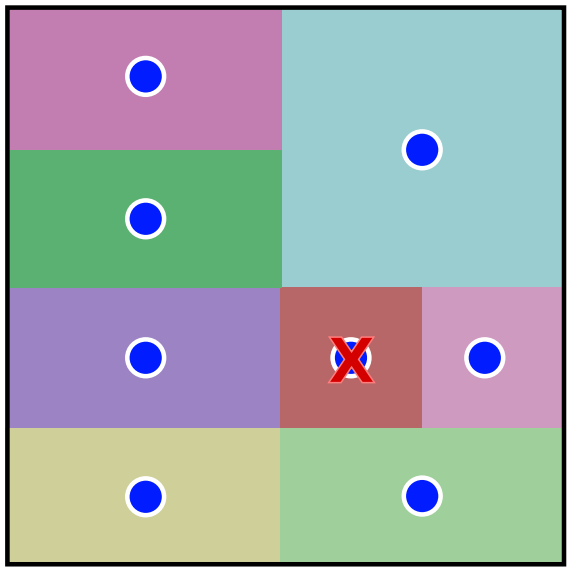
Begründung der durchschnittlichen Zahl von Nachbarn

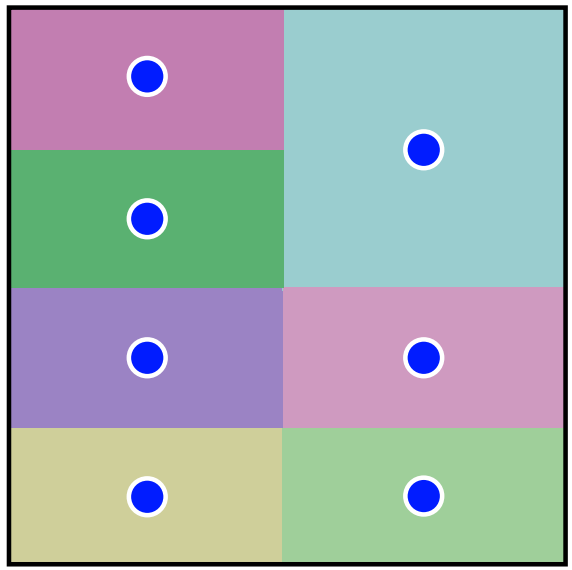
- Dass die durchschnittliche Zahl von Nachbarn in $O(d)$ liegt, lässt sich folgendermaßen begründen:
  - Denken wir die Kanten zwischen benachbarten Zonen als *gerichtete* Kanten (obwohl sie das natürlich nicht wirklich sind)
  - Zwischen unterschiedlich großen benachbarten Rechtecken zeigen die Kanten von der kleineren zur größeren Fläche, sonst ist die Richtung zufällig
  - Nun gehen von jedem Rechteck maximal $2d$ Kanten aus: zu (größeren oder gleich großen) linken und rechten Nachbarn entlang jeder Dimension
  - Bei n Rechtecken gibt es also insgesamt höchstens $2dn$ Kanten
  - Jede Kante verbindet zwei Nachbarn, insgesamt hat ein durchschnittliches Rechteck also höchstens $2 \cdot 2dn/n = 4d$ Nachbarn

Entfernen eines Peers

- Wenn ein Peer das Overlay verlässt, gibt es zwei Möglichkeiten:

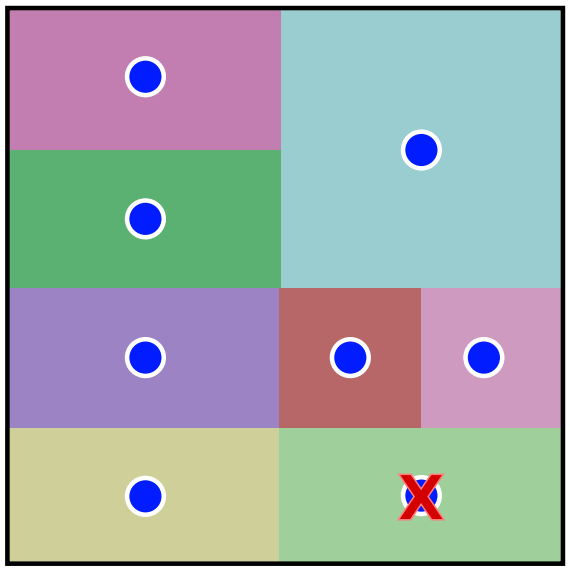
  1. Wenn möglich, sollte er seine Zone mit der passenden anderen Hälfte verschmelzen

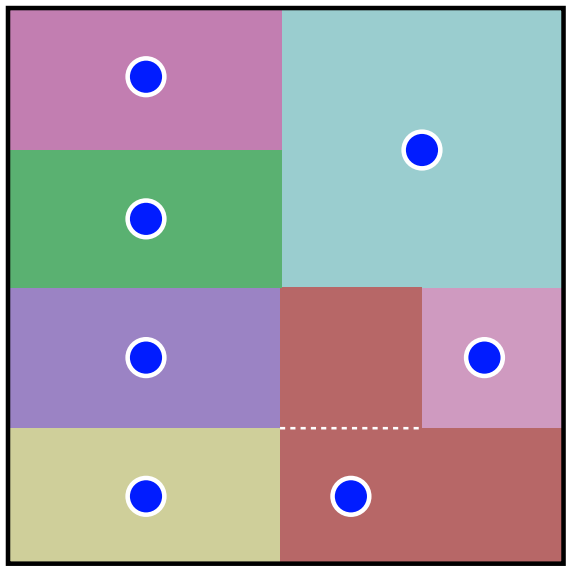




Entfernen eines Peers

- Wenn ein Peer das Overlay verlässt, gibt es zwei Möglichkeiten:
  2. Andernfalls wird das Gebiet an den Nachbarn mit dem kleinsten Zuständigkeitsbereich vergeben, dieser verwaltet dann vorübergehend zwei Zonen





Update-Nachrichten

- Peers senden periodisch *Update*-Nachrichten mit Informationen über ihre Zuständigkeitsbereiche an alle ihre Nachbarn
- Dadurch wird das Wissen der Nachbarn „aufgefrischt“ (Soft-State)
- Ein Hauptzweck ist die proaktive Reparatur von Inkonsistenzen im Overlay, beispielsweise aufgrund verloren gegangener Nachrichten
- Der andere zentrale Zweck ist das Erkennen und Reparieren von Peer-Ausfällen
- Update enthält auch die Liste aller *eigenen* Nachbarn

Ausfall eines Peers

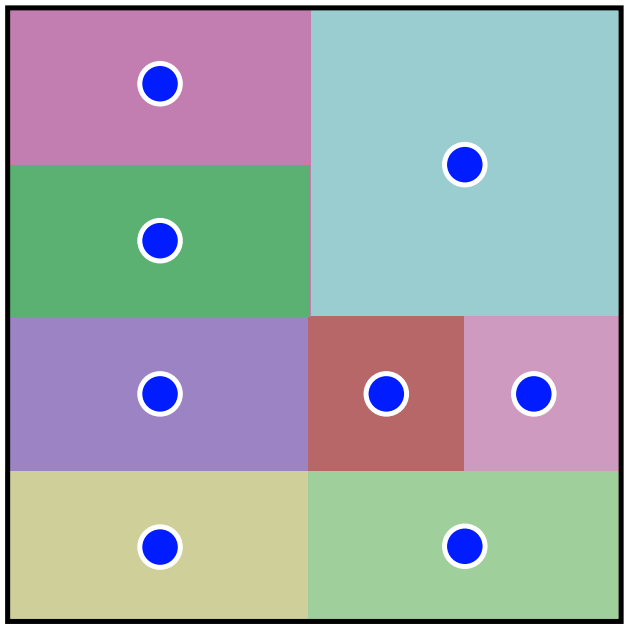
- Längeres Ausbleiben der Update-Nachrichten von einem Nachbarn signalisiert einen Ausfall
- Stellt ein Peer dies fest, sendet er eine *Takeover*-Nachricht an alle Nachbarn des *ausgefallenen* Peers
- Dabei wird sichergestellt, dass der Nachbar mit der kleinsten eigenen Zone das Gebiet des ausgefallenen Peers übernimmt:
  - Der Timeout ist länger bei Peers mit größerem eigenem Gebiet, deswegen werden in der Regel Peers mit kleiner verwalteter Zone schneller reagieren
  - Bei Empfang einer Takeover-Nachricht überprüft ein Peer außerdem, ob sein eigenes verwaltetes Gebiet kleiner ist
  - Falls ja, reagiert er seinerseits mit einer Takeover-Nachricht
- Kann auch wieder dazu führen, dass Peers mehrere Zonen verwalten ($\Rightarrow$ Defragmentierung wird notwendig)

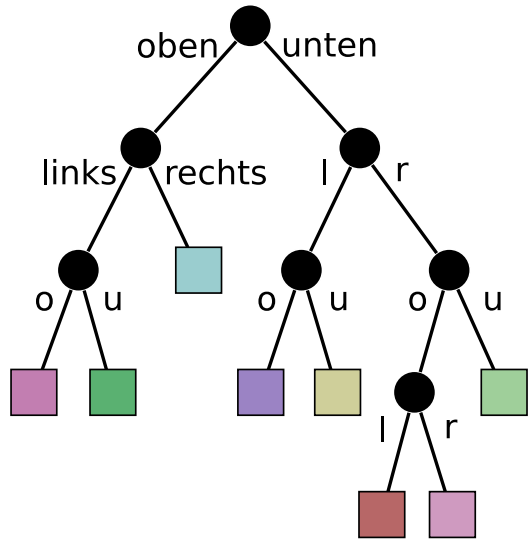
Datenreparatur

- Beim Ausfall eines Peers können Informationen zu Schlüsseln verloren gehen
- Vorschlag der CAN-Autoren: Das Einstellen eines Schlüssel-Wert-Paares soll periodisch wiederholt werden (wieder Soft-State)
- Wenn nicht aufgefrischte Einträge automatisch nach einer gewissen Zeit entfernt werden, hat das außerdem den Vorteil, dass beim Ausfall des eintragenden Peers keine „Leichen“ im System zurückbleiben

Defragmentierung

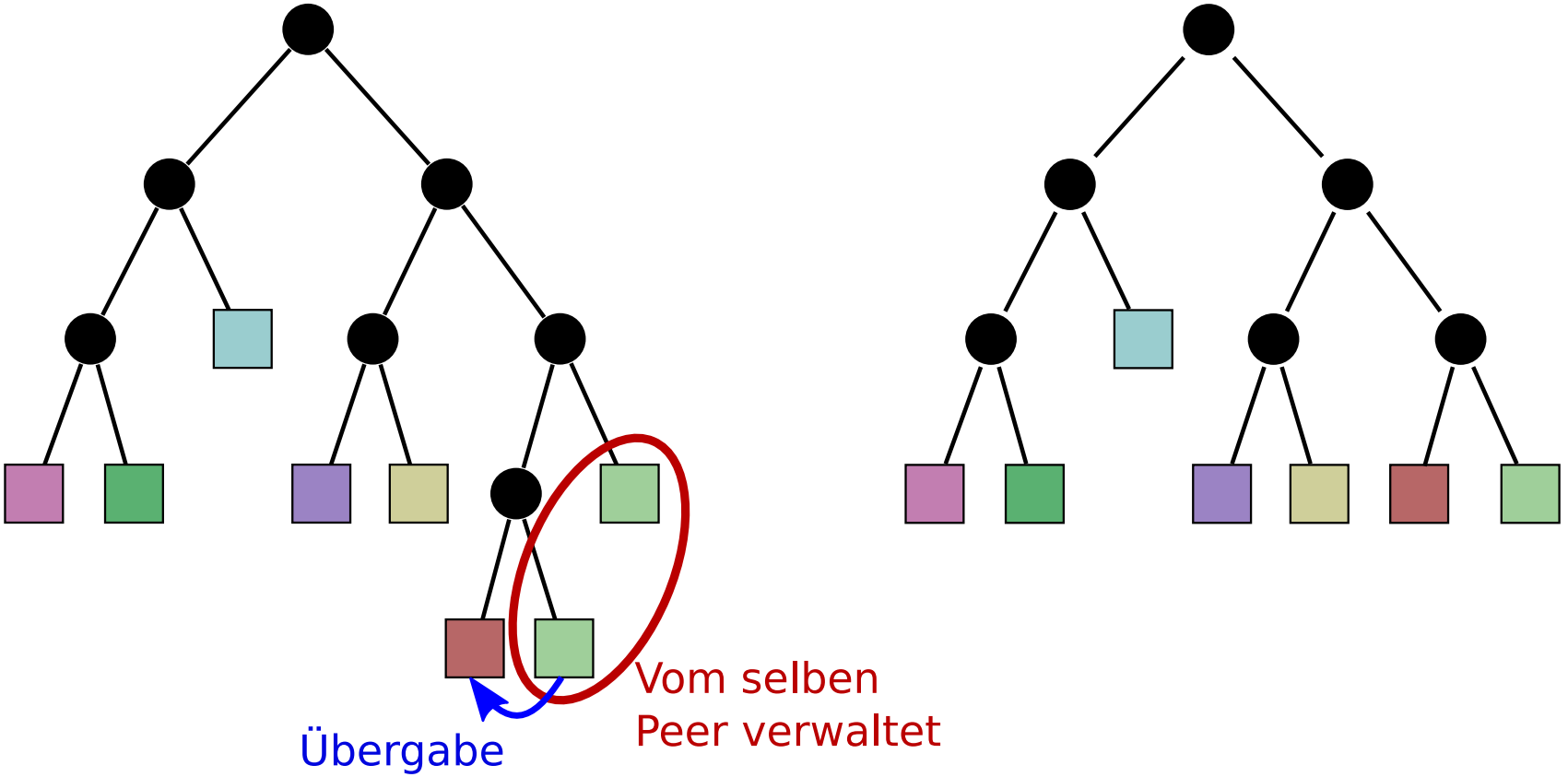
- Peers, die mehrere Zonen verwalten, können einen Defragmentierungsprozess anstoßen
- Dabei übergeben sie das kleinste von ihnen verwaltete Rechteck an einen anderen Peer
- Um den Vorgang zu verstehen, betrachten wir den CAN-Raum als Baum:





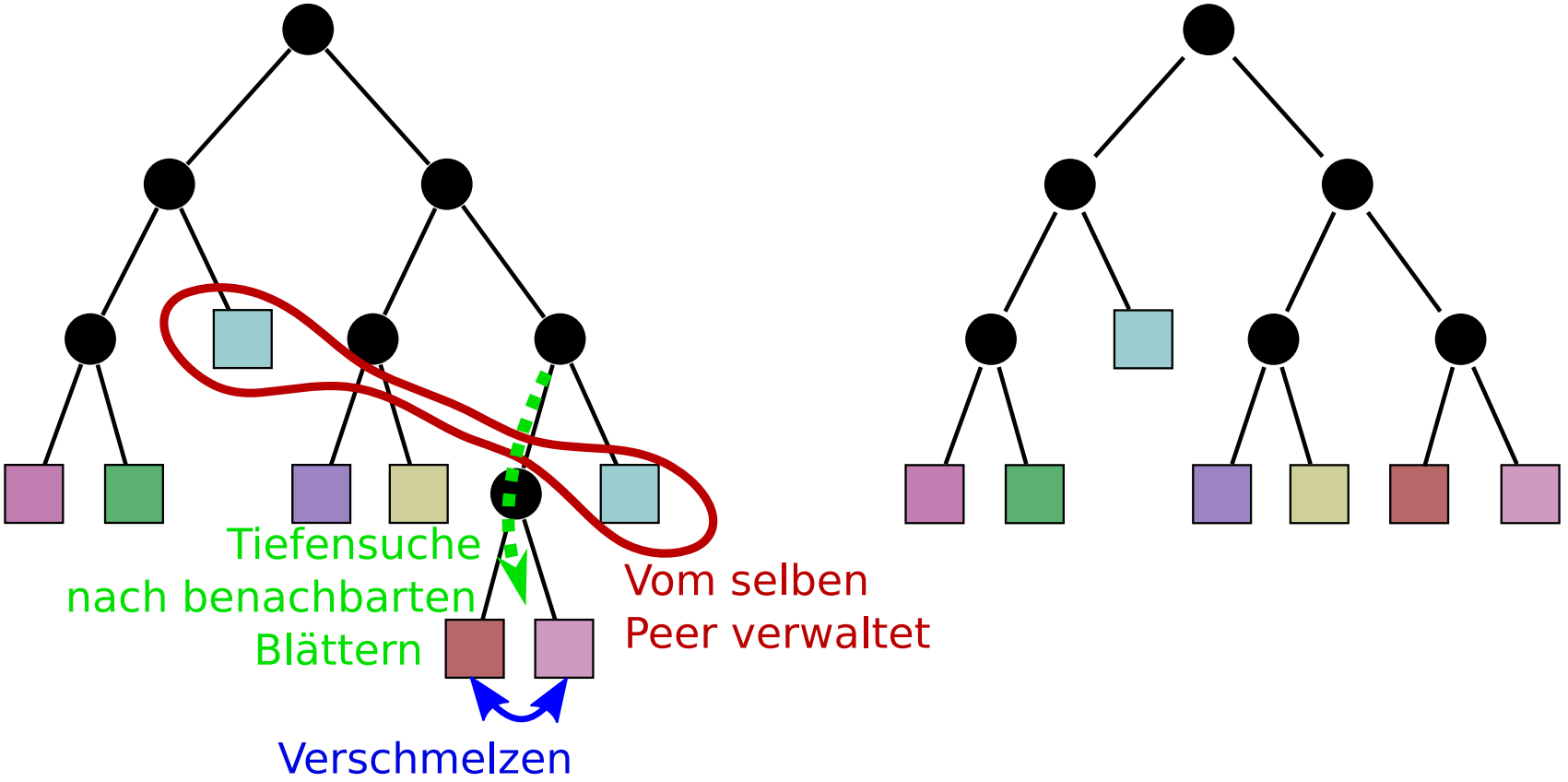
Defragmentierung: Einfacher Fall

- Im einfachen Fall ist der Nachbar-Teilbaum des kleinsten von dem Peer verwalteten Gebietes ein Blatt
- Dann wird das Gebiet an den entsprechenden Peer übergeben und so die Blätter verschmolzen



Defragmentierung: Komplexer Fall

- Andernfalls wird im benachbarten Teilbaum nach zwei direkt zueinander benachbarten Blättern gesucht
- Diese werden verschmolzen, dadurch wird ein Peer frei
- Der freie Peer übernimmt dann die Zone

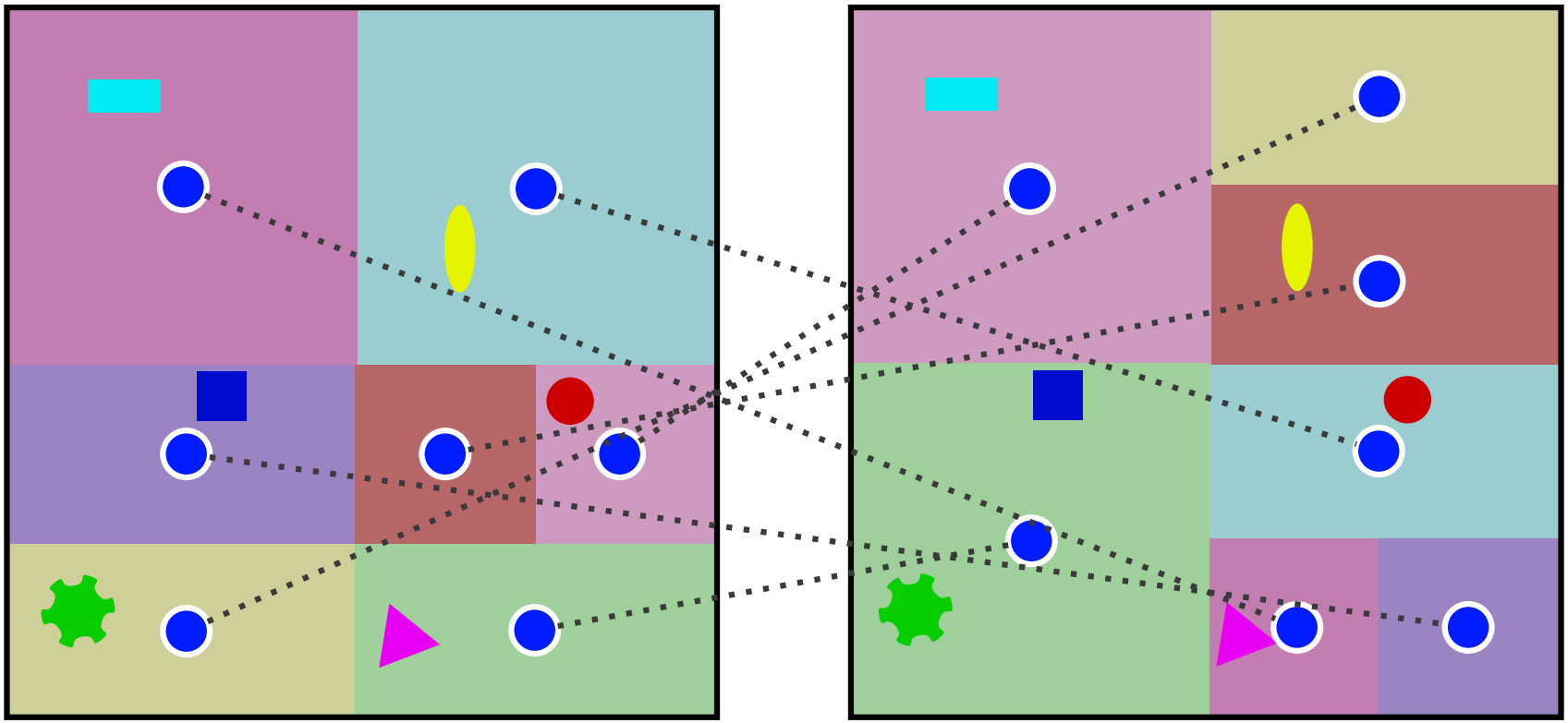


CAN-Erweiterungen

- Wir kennen nun die vollständige „Basisvariante“ von CAN
- Es gibt aber noch sehr viele Ideen, wie diese modifiziert und verbessert werden kann; im Folgenden werden wir einige davon besprechen
- Unser Fokus soll dabei auf den *Tradeoffs* liegen: Welche Abwägungen können (bzw. müssen) wir treffen, wo gibt es „Stellschrauben“?
- Mein Ziel ist, Ihren Blick zu schärfen für das Abschätzen und Beurteilen von Vor- und Nachteilen algorithmischer Modifikationen

Mehrere Realitäten

- Es ist möglich, mehrere unabhängige CAN-Räume gleichzeitig zu unterhalten
- Jeder Peer verwaltet eine Zone in jeder dieser *Realitäten*
- Die Schlüssel-Wert-Paare werden in jeder Realität repliziert



Auswirkungen mehrerer Realitäten

- Mehrere Realitäten erhöhen die Robustheit, denn ein Wert bleibt erhalten, solange mindestens einer der dafür zuständigen Peers „überlebt“
- Realitäten erhöhen außerdem die Routingeffizienz:
  - Es genügt, den gesuchten Schlüssel in *irgendeiner* Realität zu erreichen
  - Beim Routing kann jeder weiterleitende Peer deswegen frei zwischen den Realitäten wählen
  - Weiterleitung erfolgt dann zu dem Nachbarn, der den verbleibenden Abstand zum Schlüssel minimiert

Realitäten vs. Dimensionen

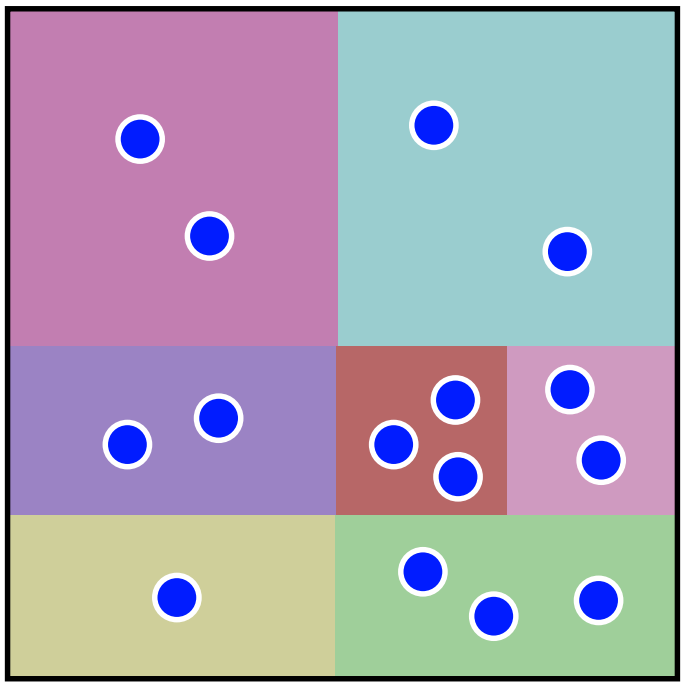
- Sowohl zusätzliche Realitäten als auch zusätzliche Dimensionen. . .
  - . . . verringern die durchschnittliche Pfadlänge
  - . . . erhöhen die Zahl der Nachbarn und vergrößern damit den Zustand pro Peer und den Aufwand für das Pflegen der Datenstruktur
- Für die gleiche Zahl von Nachbarn verringern mehr Dimensionen die Pfadlänge stärker
- Dafür bieten Realitäten aber zusätzlich den Vorteil der erhöhten Robustheit

Underlay-abhängiges Routing

- Die Zahl der Schritte im Overlay zu minimieren ist nicht (immer) das richtige Ziel
- Schritte im Overlay können sehr verschieden lang sein im darunterliegenden Netz ($\Rightarrow$ Zickzack-Routen in Gnutella!)
- Bei der Wahl des nächsten Schrittes für das Routing in CAN kann deshalb zusätzlich zur „Richtung“ auch die Rundlaufzeit (Round Trip Time, RTT) zum jeweiligen Nachbarn berücksichtigt werden („Underlay-Aware Routing“)
- Dafür müssen die Peers periodisch die RTT zu ihren Nachbarn messen
- Dadurch kann die *Latenz*, also die Gesamtlaufzeit der Anfrage, verringert werden

Überladen von Zonen

- Bisher wurde angenommen, dass jede Zone zu jedem Zeitpunkt nur von einem Peer verwaltet wird
- Um die Robustheit zu erhöhen können auch mehrere Peers gemeinsam eine Zone verwalten
- Ein Peer muss *alle* anderen Peers in seiner eigenen Zone kennen, es genügt aber, wenn er *einen* Peer aus jeder Nachbarzone kennt



Überladen von Zonen

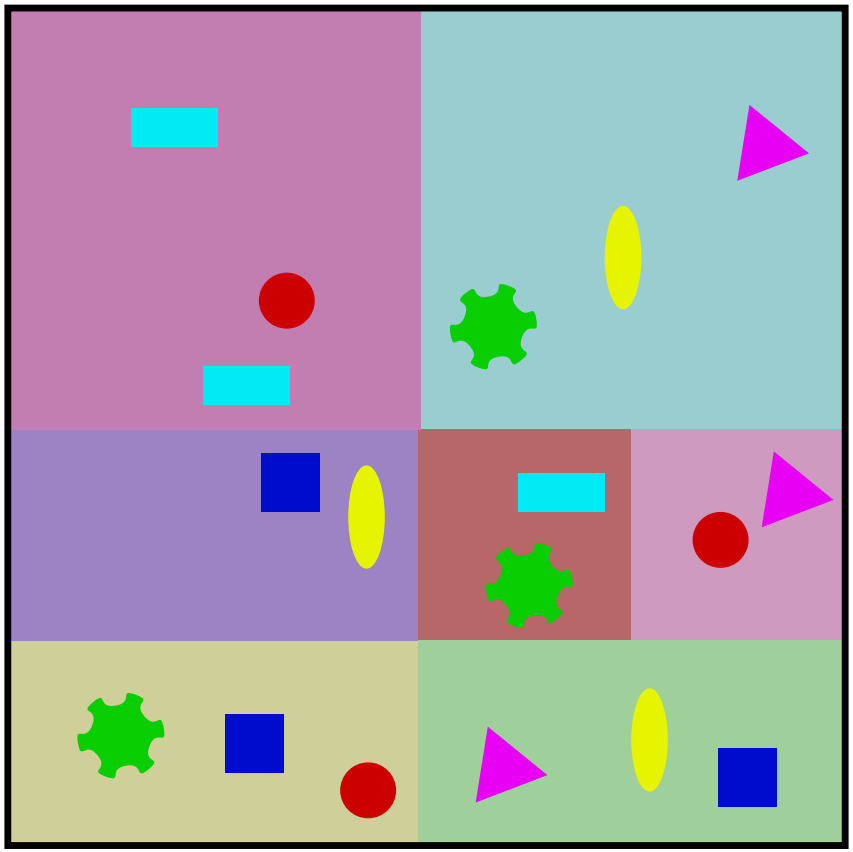
- Es wird eine Maximalzahl von Peers pro Zone festgelegt
- Beim Hinzufügen eines Peers wird die Zone erst dann aufgeteilt, wenn sonst die Maximalzahl überschritten würde
- Die Schlüssel in der gemeinsam verwalteten Zone können entweder unter den Peers aufgeteilt oder auf alle Peers repliziert werden
- Kombination mit Underlay-abhängigem Routing ist möglich, indem gezielt derjenige der Peers aus den angrenzenden Zonen als Nachbar ausgewählt wird, zu dem die geringste RTT gemessen wird

Auswirkungen des Überladens

- Überladen reduziert die Pfadlänge (analog zu einer geringeren Zahl von Peers im System!)
- Geringere Latenz, da zwischen mehreren Nachbarn gewählt werden kann
- Höhere Robustheit gegen Peer-Ausfälle, da Zonen redundant besetzt sind
- Dafür aber höhere Protokoll- und Systemkomplexität!

Verwendung mehrerer Hashfunktionen

- Schlüssel-Wert-Paare können auch mittels *mehrerer, unabhängiger* Hashfunktionen an mehreren Stellen im CAN-Raum platziert werden



Verwendung mehrerer Hashfunktionen

- Wenn k Hash-Funktionen verwendet werden, erhöht sich der Speicheraufwand um den Faktor k
- Erhöht die Redundanz und damit die Robustheit
- Außerdem kann wiederum die Pfadlänge reduziert werden, indem eine Anfrage an die nächstgelegene Kopie geleitet wird
- Minimierung der Latenz wird möglich, indem alle Kopien *gleichzeitig* angefragt werden, denn die zuerst eintreffende Antwort genügt – das geht dann jedoch auf Kosten des Nachrichtenaufwands

Caching und Replikation

- Bestimmte Schlüssel-Wert-Paare könnten evtl. sehr viel häufiger angefragt werden als andere
- Dadurch können „Hot Spots“ im Netzwerk entstehen und einzelne Peers sehr stark belastet werden
- Dem kann man durch Caching- und Replikationstechniken entgegenwirken:
  - Peers können vor kurzem angefragte Schlüssel-Wert-Paare zwischenspeichern und damit später durch sie geroutete Anfragen nach dem Schlüssel direkt beantworten (Caching); so steigt die Zahl der verfügbaren Kopien mit der Popularität der Daten
  - Wenn ein Peer durch die Anfragen nach einem bestimmten Schlüssel zu stark belastet wird, kann er die betreffenden Daten auf die Nachbarn dieses Peers replizieren; dadurch können entsprechende Anfragen schon auf dem Weg zu dem Peer „abgefangen“ und die Last auf die gesamte Nachbarschaft verteilt werden

CAN – Diskussion

- CAN kann im Gegensatz zu Gnutella garantieren, dass vorhandene Daten gefunden werden
- Trotzdem ist der Aufwand für eine Suche vergleichsweise niedrig:

  Zahl der Nachbarn in $O(d)$

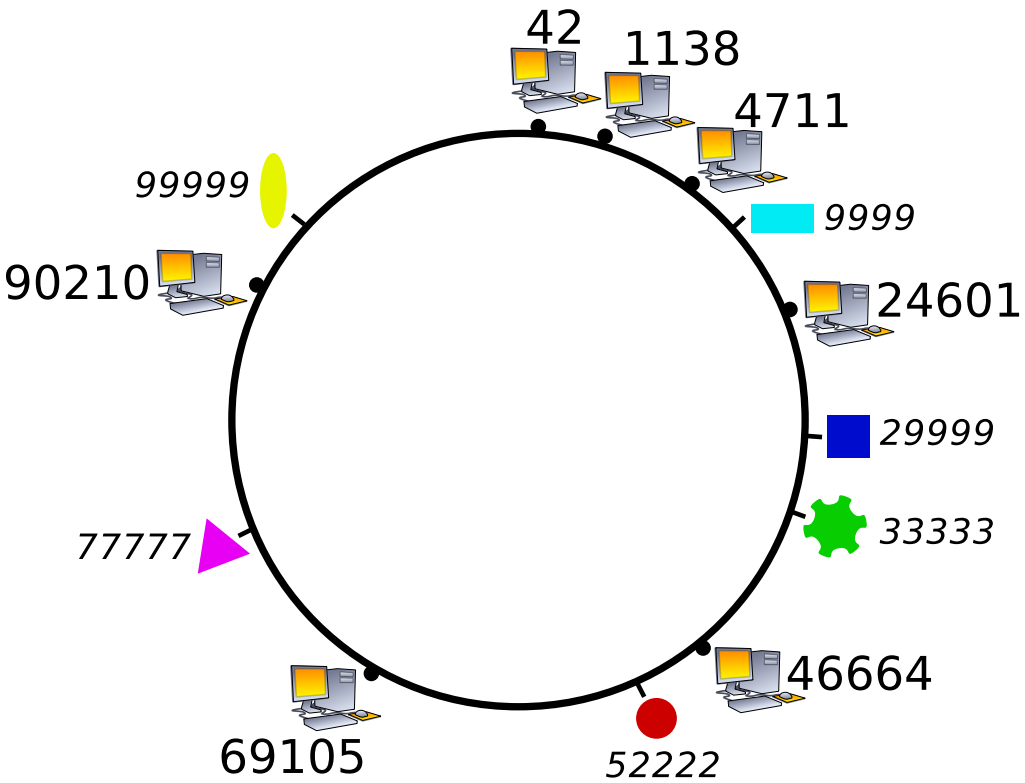
  Routing in $O(dn^{1/d})$ Schritten
- Es geht aber noch effizienter!

Chord

- Ungefähr zeitgleich zu CAN entstand Chord
- Gleiches Problem – andere Lösung
- Wiederum werden Schlüssel mit einer Hashfunktion abgebildet, diesmal jedoch auf Ganzzahlen im Bereich $0, \ldots, 2^m - 1$
- Auch die IDs der Peers werden in diesen Bereich gehasht

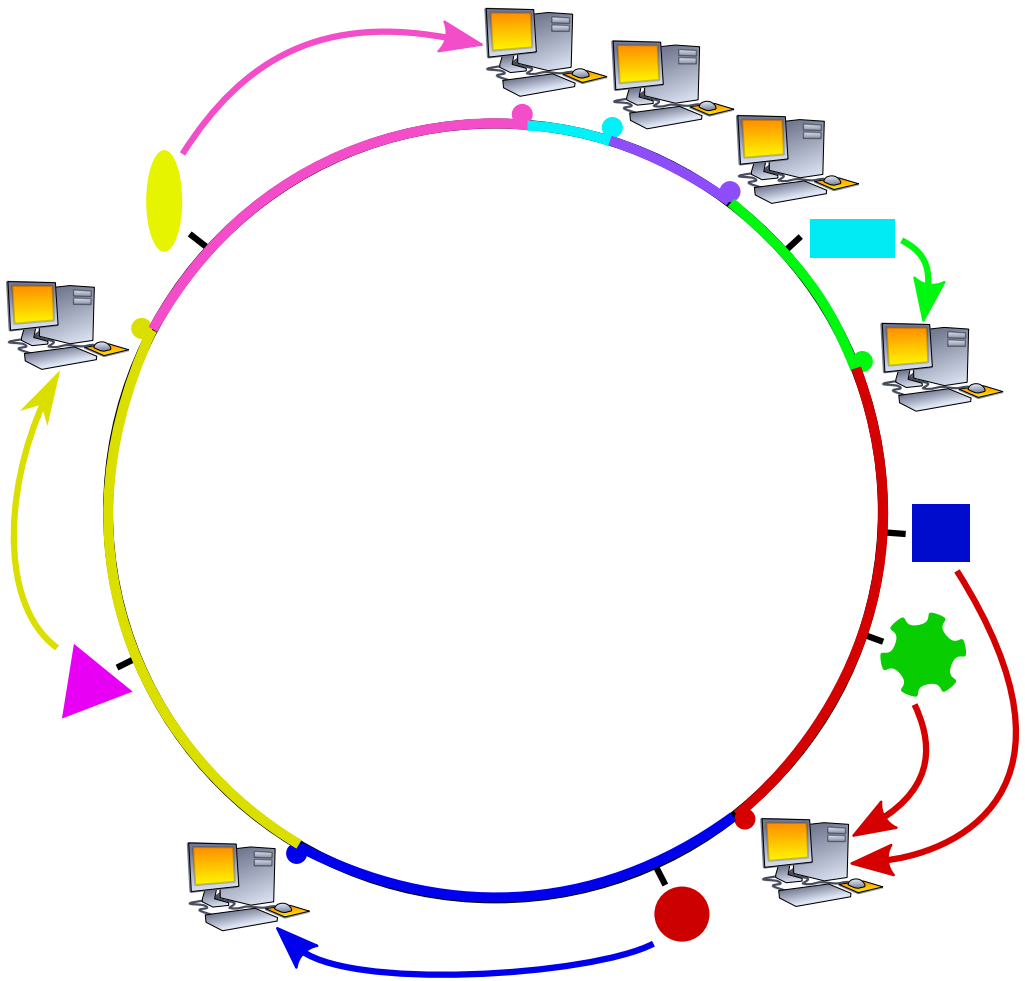
Chord-Ring

- Der Wertebereich wird als Ring aufgefasst
- Auf diesem Ring liegen die Schlüssel *und* die Peers
- Berechnungen in Chord immer modulo 2^m



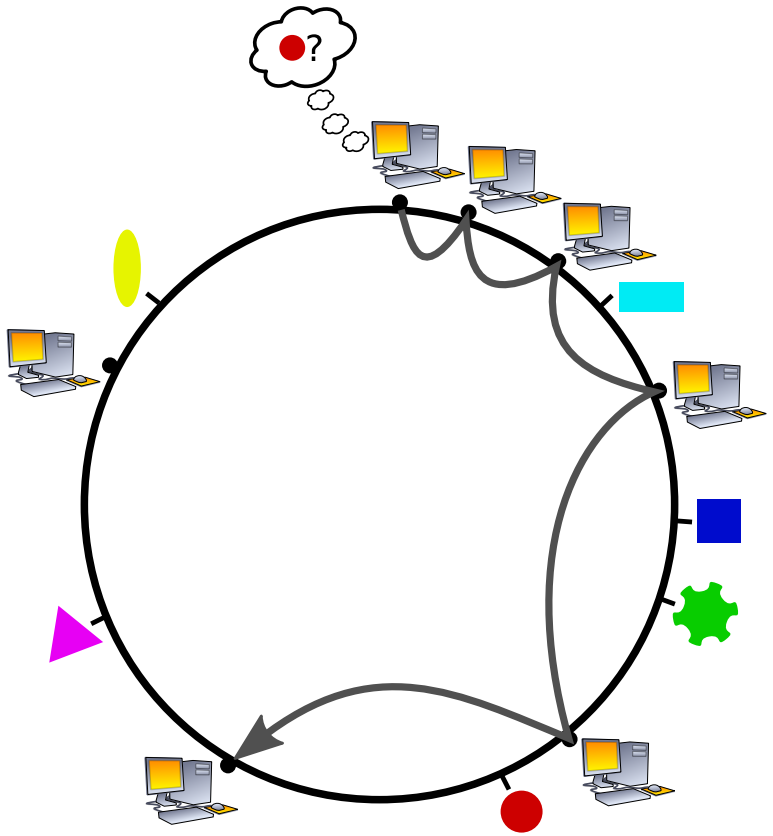
Schlüsselverantwortlichkeiten in Chord

- Jeder Peer verwaltet die Schlüssel-Wert-Paare, die auf dem Ringsegment liegen, das bei diesem Peer endet



Routing über Successor-Links

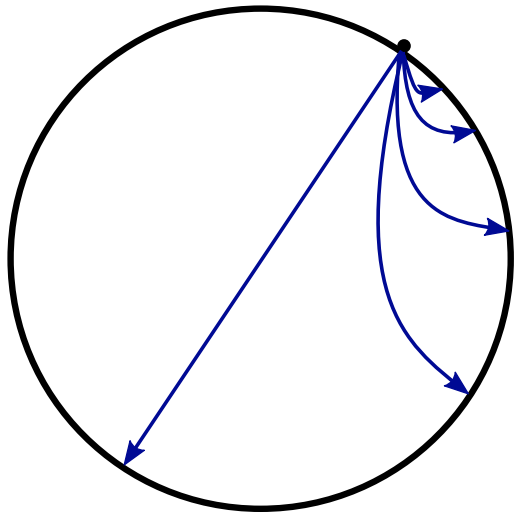
- Der *Nachfolger* oder *Successor* succ(x) einer Position x sei der von x aus nächstgelegene Peer im Uhrzeigersinn
- Der Nachfolger eines Peers mit Position p ist succ($p + 1$)
- Jeder Peer kennt seinen Nachfolger
- Das genügt (im Prinzip) schon zum Routing!



Finger

- Routing über Successor-Links ist nicht effizient:
  $O(n)$ Schritte bei n Peers
- Deshalb zusätzliche Routingeinträge: „Finger“
- Ein Peer an Position p kennt Ringposition und IP von

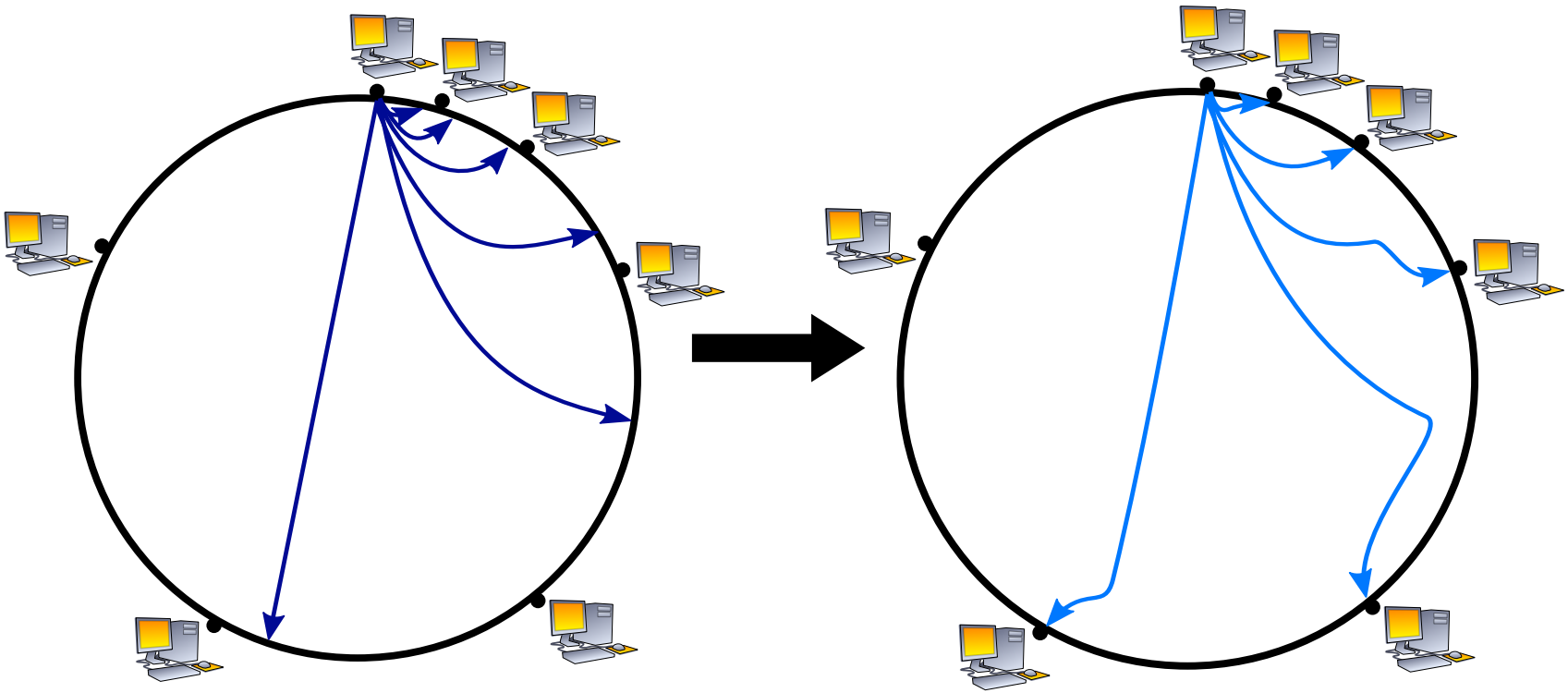
$$f_i = \text{succ}(p + 2^{i-1}) \quad \text{für} \quad i = 1, \ldots, m$$



- Beachte: Der erste Finger $f_1 = \text{succ}(p + 1)$ ist immer der Nachfolger des Peers

Finger

- Die Finger zeigen wegen der Successor-Funktion immer auf den ersten Peer *an oder hinter* den Positionen $p + 2^{i-1}$



Routing mit Fingern

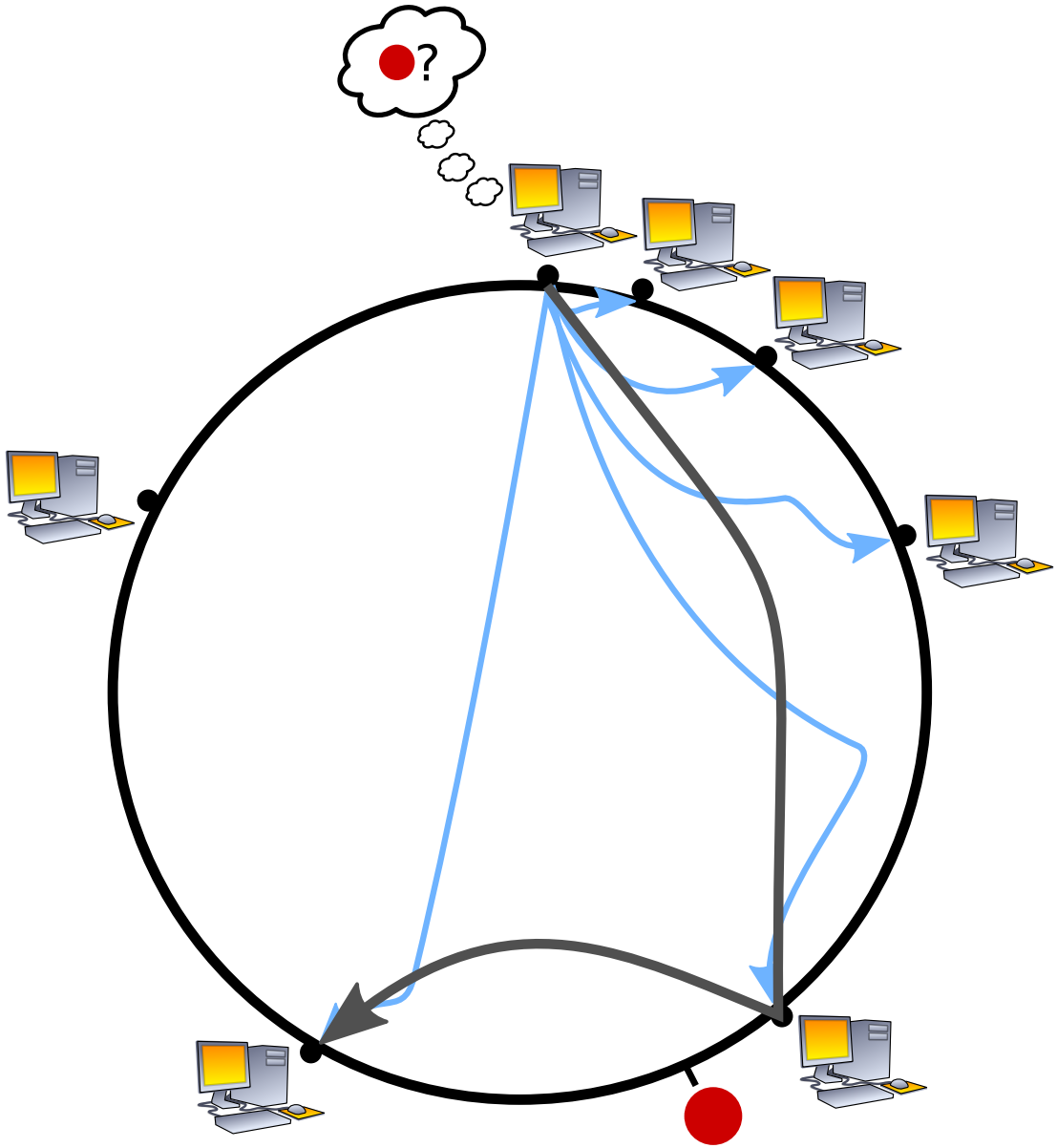
Routing zu einem gesuchten Schlüssel an Position k wird in Chord folgendermaßen durchgeführt:

1. Falls $k \in (p, \text{succ}(p+1)]$

   $\Rightarrow$ Successor ist zuständig: fertig!

2. Andernfalls verwende denjenigen Finger, dessen Zielposition am nächsten *vor* dem gesuchten Schlüssel k liegt („Herantasten“ an den gesuchten Schlüssel)

Routing mit Fingern



Größe der Finger-Tabelle

- Die Finger-Tabelle hat m Einträge – aber wie viele davon sind *verschieden*?
- M. h. W. liegt im Intervall $(p, p + 2^m/n^2]$ hinter einem Peer an Position p kein weiterer Peer

$\Rightarrow$ Für alle $i \leq m - 2\log_2 n$ ist m. h. W. $f_i = f_1 = \text{succ}(p + 1)$

- Alle außer den letzten $2\log_2 n$ Einträgen der Finger-Tabelle zeigen also m. h. W. auf denselben Peer

Die Finger-Tabelle hat m. h. W. $O(\log n)$ (statt $O(m)$) verschiedene Einträge

Routing mit Fingern: Komplexität

Ein gesuchter Schlüssel k wird in $O(m)$ Schritten erreicht

Beweis: Betrachten wir die Zahl der Schritte, bis eine vom Peer p gestartete Suche den Peer x unmittelbar vor dem gesuchten Schlüssel k erreicht. Der interessante Fall ist $k \notin (p, \text{succ}(p+1)]$.

Sei f_i der Finger, für den der gesuchte Peer x in $[p + 2^{i-1}, p + 2^i)$ liegt. Dieses Intervall enthält mindestens einen Peer (nämlich x), deshalb wird p als nächsten Routing-Schritt den Peer wählen, auf den f_i zeigt.

Der Abstand zwischen p und f_i ist mindestens 2^{i-1}. Da f_i und x beide im Intervall $[p + 2^{i-1}, p + 2^i)$ liegen, ist der Abstand zwischen diesen beiden Peers höchstens 2^{i-1}. Also ist f_i nicht weiter von x als von p entfernt und es wird mindestens die Hälfte der verbleibenden Distanz zurückgelegt.

Der ursprüngliche Abstand von höchstens 2^m wird also in jedem Schritt (mindestens) halbiert.

Routing mit Fingern: Komplexität

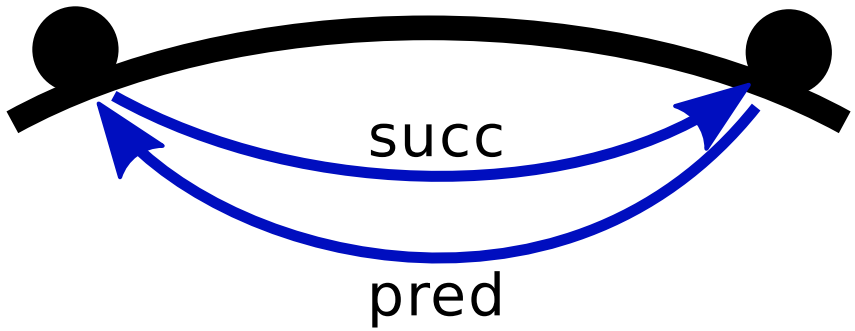
- Nach $2 \log_2 n$ Schritten (also $(2 \log_2 n)$-maligem Halbieren der Distanz) ist der Abstand auf höchstens $2^m / n^2$ geschrumpft
- Analog zum Argument zur Größe der Finger-Tabelle ist dann m. h. W. nur ein weiterer Schritt nötig

Die Zahl der Routing-Schritte ist damit m. h. W. in $O(\log n)$

- Tatsächlich zeigt sich, dass die durchschnittliche Pfadlänge $\frac{1}{2} \log_2 n$ ist

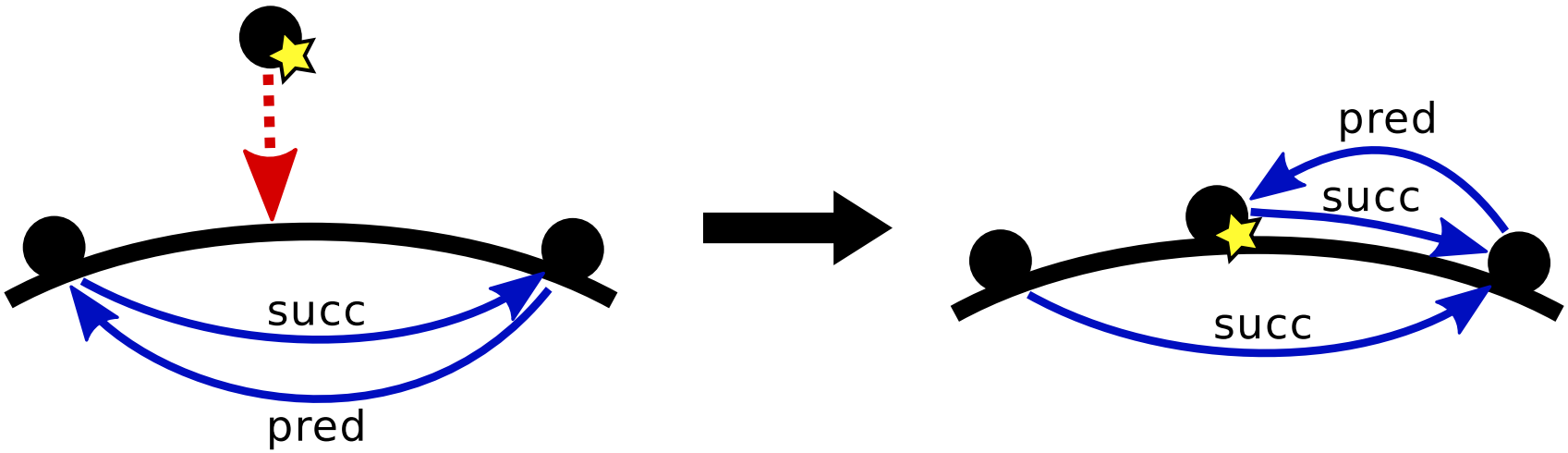
Verwalten des Chord-Rings

- Wir wenden uns nun der Frage zu, wie ein Chord-Ring bei kommenden oder gehenden Peers gepflegt wird
- Für Verwaltungszwecke (nicht zum Routing!) kennen Peers zusätzlich zu ihrem Nachfolger (*Successor*, $\text{succ}(p) = f_1$) auch ihren Vorgänger im Ring (*Predecessor*, $\text{pred}(p)$)



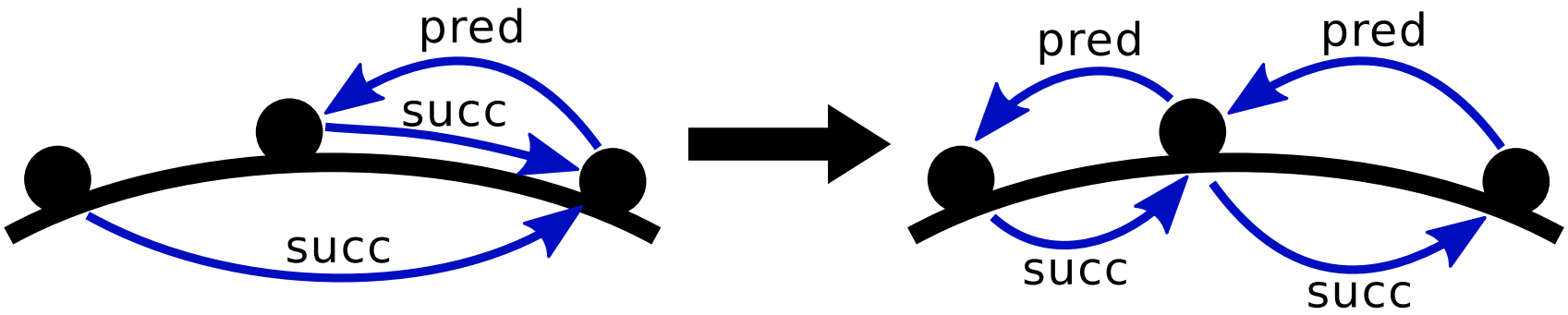
Einfügen von Peers

- Zum Einfügen wählt ein Peer eine zufällige Position auf dem Ring
- Er sucht den Nachfolger der Position im Ring und informiert diesen über seinen neuen Vorgänger
- Außerdem werden ggf. Schlüssel übernommen – das war's auch schon!



Stabilize

- Periodisch führt jeder Peer p eine *Stabilisierungsfunktion* aus, die korrekte Successor-Pointer garantiert
- Dabei fragt p seinen Nachfolger nach dessen Vorgänger (sei n der so ermittelte Peer)
- Falls $n \neq p$ ist, wird
  1. succ(p) $\leftarrow$ n gesetzt
  2. n darüber informiert, dass p jetzt sein Vorgänger sein *könnte*; dies wird von n akzeptiert und gespeichert, falls noch kein näherer Vorgänger bekannt ist



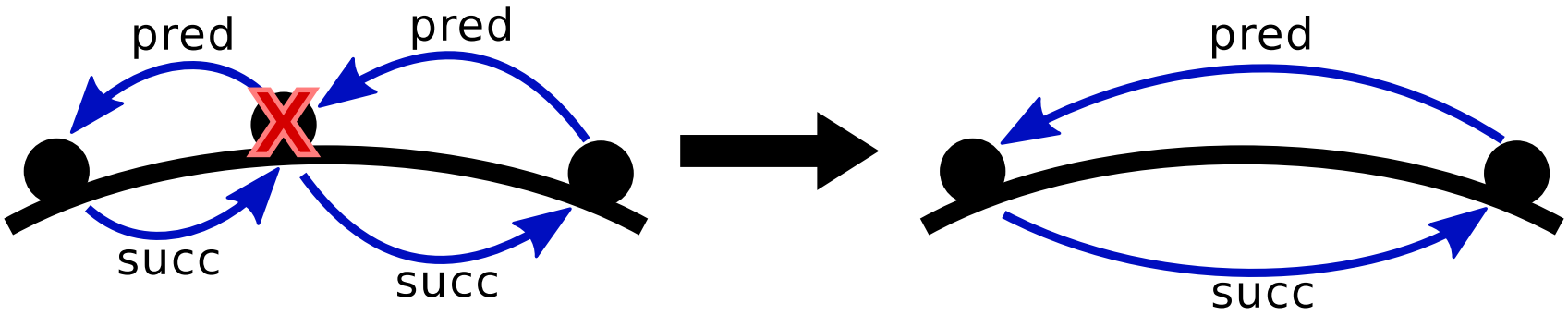
Fix-Fingers und Check-Predecessor

- Die Ziele der Finger eines Peers können sich durch kommende oder gehende Peers mit der Zeit verändern
- Die Peers suchen deshalb regelmäßig im Overlay nach den aktuellen Zielen ihrer Finger („Fix-Fingers“)
  - also: Suche nach den Peers $f_i = \text{succ}(p + 2^{i-1})$
- So wird die Finger-Tabelle aktuell gehalten

- Ebenfalls periodisch wird geprüft, ob pred(p) noch erreichbar ist („Check-Predecessor“)
- Falls nicht, wird einfach pred(p) $\leftarrow$ null gesetzt – den Rest erledigt die Stabilisierungsfunktion!

Entfernen von Peers

- Ein bewusst den Chord-Ring verlassender Knoten benachrichtigt seinen Vorgänger und Nachfolger und übergibt seine Schlüssel an den Nachfolger
- Das genügt bereits, um eine konsistente Struktur zu erhalten

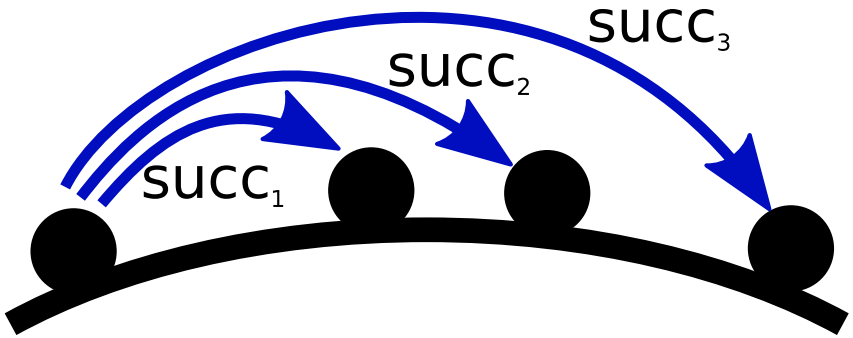


Ausfall von Peers

- Beim Ausfall eines Peers werden die Finger benutzt, um den Ring zu reparieren
- Wenn ein Peer feststellt, dass sein Nachfolger nicht mehr erreichbar ist, setzt er den Nachfolger auf den nächstgelegenen erreichbaren Finger
- Die Stabilisierungsfunktion wird dann (ggf. in mehreren Schritten) die Ringstruktur wiederherstellen
- Es gibt dann aber pathologische Fälle, die sich nicht reparieren lassen (wenn auch unwahrscheinlich – z. B. gleichzeitiger Ausfall aller Finger-Ziele)

Successor-Listen

- Fehlerhafte Successor-Pointer können (bis zu ihrer Reparatur) zu fehlschlagenden Suchen führen
- Um die Robustheit von Chord zu erhöhen, können *Successor-Listen* mit $k > 1$ Einträgen zum Einsatz kommen



- Stabilize, Fix-Fingers und der Suchalgorithmus sind dann entsprechend anzupassen
- Es wurde gezeigt, dass Chord mit Successor-Listen der Länge $\Omega(\log n)$ sogar bei massiven parallelen Ausfällen weiterhin robust *und sogar effizient* ist

Chord – Diskussion

- Chord implementiert eine verteilte Hashtabelle mit

  $O(\log n)$ Zustand und

  Routing in $O(\log n)$ Schritten
- Konsistenz der Ringstruktur wird über einen Selbststabilisierungs-Ansatz erhalten
- Vergleich mit CAN (Zustand $O(d)$, Routing $O(dn^{1/d})$)?
- Was passiert, wenn $d = \log n$ gesetzt wird?