

Vorlesung P2P Netzwerke

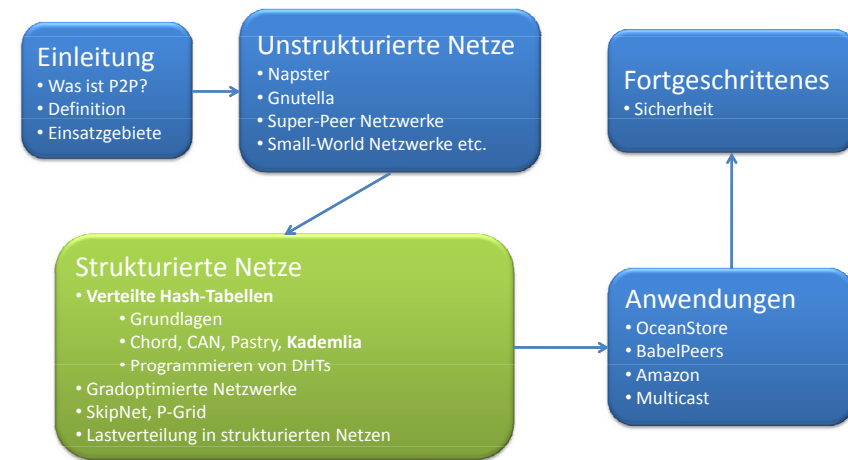
6: Kademia



Dr. Dominic Battré
Complex and Distributed IT-Systems
dominic.battre@tu-berlin.de



Inhalte der Vorlesung (vorläufig)



28.05.2009

Dominic Battré - P2P Netzwerke

2



Kademia

- Kademia
 - Petar Maymounkov and David Mazierès: "Kademia: A Peer-to-peer information System Based on the XOR Metric", 1st International Workshop on Peer-to-peer Systems, 2002



Kademia

- Populäre DHT (muTorrent, Ktorrent, Vuze, BitTorrent, Cspace, eDonkey 2000, MLDonkey, ... siehe Wikipedia)
- Geringer Wartungsaufwand für konsistente Routing Tabellen
- Knoten haben genügend Informationen und Flexibilität um Low-Latency Pfade zu wählen
- Parallele, asynchrone Anfragen, um Verzögerungen durch Timeouts zu umgehen
- Sehr Robust (u.A. gegen gewisse Denial of Service Angriffe)

28.05.2009

Dominic Battré - P2P Netzwerke

3

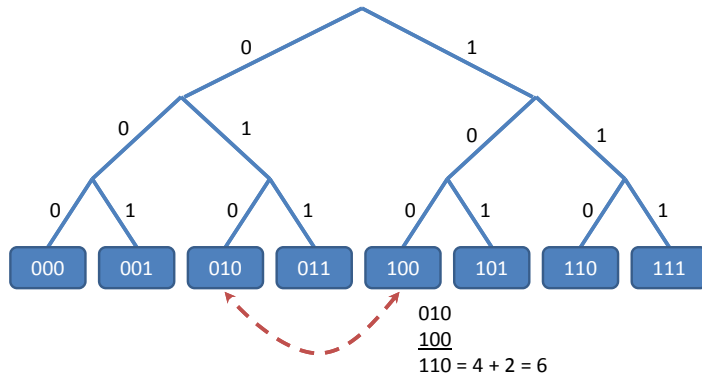
28.05.2009

Dominic Battré - P2P Netzwerke

4

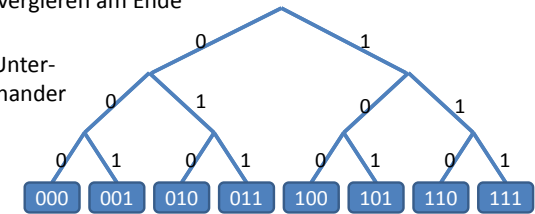
XOR Metrik

- Distanz $d(A, B) = A \oplus B$

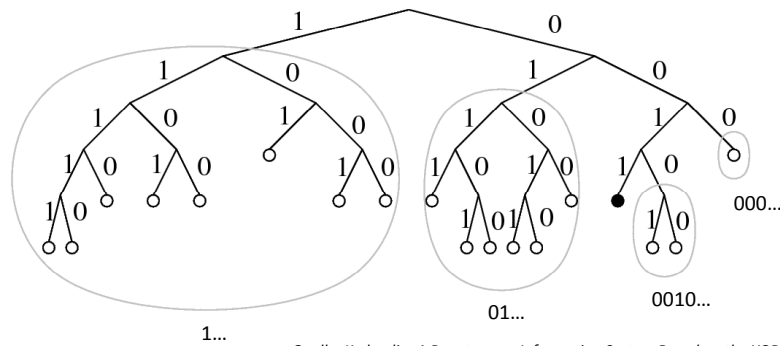
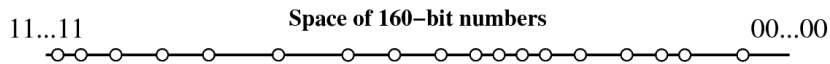


XOR Metrik

- Eigenschaften:
 - $d(X, X) = 0$
 - $d(X, Y) = 0 \rightarrow X = Y$
 - $d(X, Y) = d(Y, X) \rightarrow$ Anfragen kommen von Knoten, die in Routing Tabelle stehen können (\rightarrow Verbessern der Tabellen)
 - $d(X, Y) \leq d(X, Z) + d(Z, Y)$
 - Zu jedem Knoten gibt es **genau** einen Knoten mit Distanz d
 \rightarrow Routingpfade konvergieren am Ende
 \rightarrow Caching möglich
 - Knoten im gleichen Unterbaum sind nah beieinander



Routing Tabelle

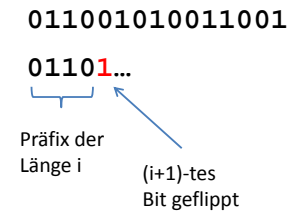


Quelle: Kademlia: A Peer-to-peer Information System Based on the XOR Metric
 Petar Maymounkov and David Mayières

Routing Tabelle

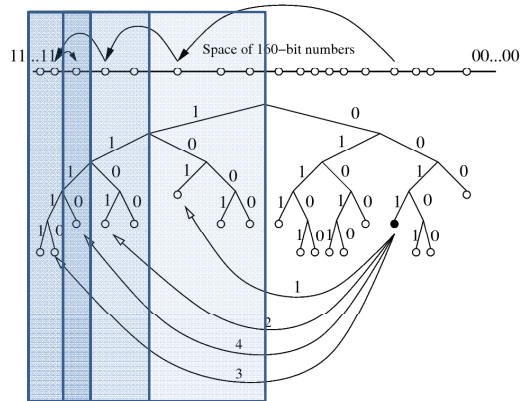
- Jeder Knoten kennt mindestens einen Knoten in jedem Unterbaum, sofern dieser Unterbaum einen Knoten enthält.

- Knoten Id:
- Unterbaum i:

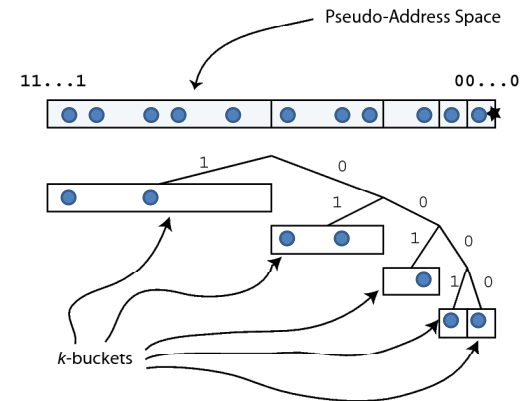


Routing Beispiel

- Routing von 0011... nach 1110...



Routing Tabelle



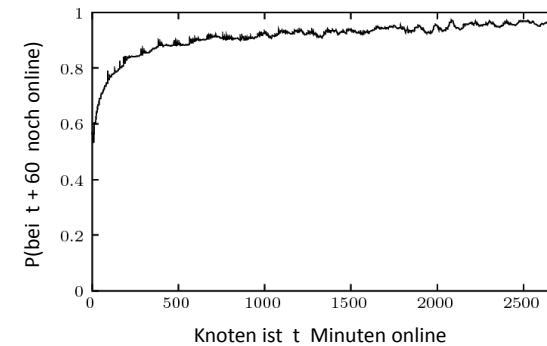
Quelle: Kademlia: A Peer-to-peer Information System Based on the XOR Metric
Petar Maymounkov and David Mayières

k-Buckets

- Ein k-Bucket für jede Präfixlänge $0 \leq i < 160$
- Jeder Bucket enthält bis zu k Knoten (z.B. $k=20$)
- Einträge in Buckets nach „least-recently seen“ sortiert
- Bei Empfang von Nachricht von Peer p
 - Wenn p schon in Bucket: verschiebe an Ende von Bucket
 - Wenn p nicht in Bucket:
 - ◆ Wenn Bucket weniger als k Peers enthält: am Ende aufnehmen
 - ◆ Sonst:
 - wenn erster Peer in Bucket auf Ping reagiert: verschiebe diesen ans Ende des Buckets
 - Sonst: lösche ersten Peer aus Bucket, nimm p am Ende des Buckets auf
- LRU-Verdrängung, aber: aktive Knoten werden nie verdrängt

k-Buckets

- Bevorzugung alter Knoten verbessert Stabilität



Quelle: Kademlia: A Peer-to-peer Information System Based on the XOR Metric
Petar Maymounkov and David Mayières

- PING
 - Testet ob Knoten online ist
- STORE
 - Speichert Datum bei Knoten
- FIND_NODE(160 Bit ID)
 - Liefert die k nächstgelegenen bekannten Knoten (bezüglich XOR Metrik!)
 - Aus einem k-Bucket, wenn dieser voll ist; sonst aus mehreren
- FIND_VALUE
 - Wie FIND_NODE, aber liefert den Wert, wenn er bekannt ist

```

1  p.lookup(T)
2  N = PriorityQueue(maximale Länge k)
3  N.add( $\alpha$  nächste Knoten)
4  solange nicht alle Knoten aus N erfolgreich gefragt:
5      k = nächster, nicht gefragter Knoten aus N
6      N' = k.FIND_NODE(T);
7      wenn N' = {}: // Fehler
8          lösche k aus N
9      sonst:
10         N = k nächste Knoten aus N und N'
11     wenn N weniger als k Knoten enthält:
12         markiere alle Knoten als ungefragt, goto 3
13     liefere nächsten Knoten aus N
  
```

```

1  p.lookup(T)
2  N = PriorityQueue(maximale Länge k)
3  N.add( $\alpha$  nächste Knoten)
4  solange nicht alle Knoten aus N erfolgreich gefragt:
5      warte, bis weniger als  $\alpha$  Anfragen ausstehen
6      p' = nächster, nicht gefragter Knoten aus N
7      sende FIND_NODE(T) an p'
8  wenn N weniger als k Knoten enthält:
9      markiere alle Knoten als ungefragt, goto 3
10 liefere nächsten Knoten aus N
  
```

```

1  On FIND_NODE Time Out (k):
2  Lösche k aus N
  
```

```

1  On FIND_NODE Answer (k, N'):
2  N = nächste k Knoten aus N und N'
  
```

- Häufig $\alpha = 3$ (sozusagen die Anzahl Worker Threads)
- Geringere Latenz für höheren Netzwerkverkehr getauscht
- Bevorzugt Routing durch Knoten mit geringer Latenz
- Vermindert Verzögerungen, wenn Knoten offline gegangen sind
- Korrektheit: s. Paper

- Store(key, value)
 - Ermittle k nächste Knoten
 - Sende STORE an jeden dieser Knoten
- Softstate
 - Neueinfügen jede Stunde
- Lookup
 - FIND_VALUE statt FIND_NODE
 - Daten werden beim zuletzt gefragten Knoten, der sie nicht speicherte, repliziert
 - Lebenszeit des Replikats: exponentiell antiproportional zu Anzahl Knoten zwischen aktuellem und zur ID nächsten Knoten

- Neuer Knoten u kennt Knoten w
- u fügt w in entsprechenden k-Bucket ein
- u macht Lookup nach sich selbst und gewinnt neue Knoten
- u wählt sich zufällige IDs in leeren k-Buckets und routet dorthin
- Während dieser Vorgänge fügt sich u automatisch in k-Buckets bei anderen Knoten ein.

- Pings werden nicht wirklich wie beschrieben verschickt (es wären zu viele)
 - Pings werden verzögert, bis sinnvolle Nachricht geroutet werden muss
 - Ein Replacement Cache hält Knoten, die ausgefallene Knoten kompensieren können
- Statt Binärbaum: 2^b -ärer Baum, reduziert Tiefe

- In echten Netzen erprobt
- Robust
- Logarithmisches Routing
- Neuer Aspekt: Nebenläufige Anfragen

Netzwerk	Referenzen nach scholar.google.com
Chord (2001+2003)	7606
CAN (2001)	5204
Pastry (2001 + 2001)	5338
Tapestry (2001)	1927
Kademlia (2002)	929