P2P

Seminar

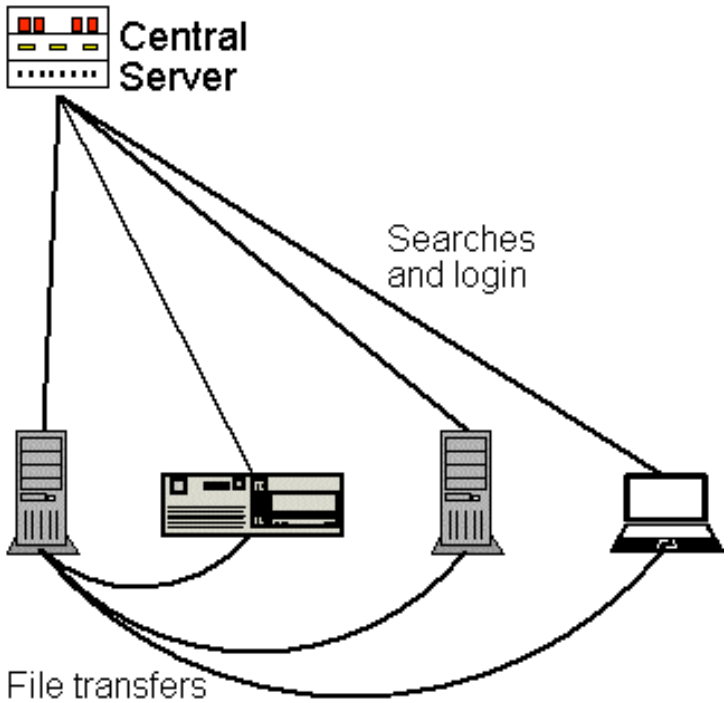
Kademlia

A Peer-to-peer Information System Based on the XOR Metric

Abgrenzung

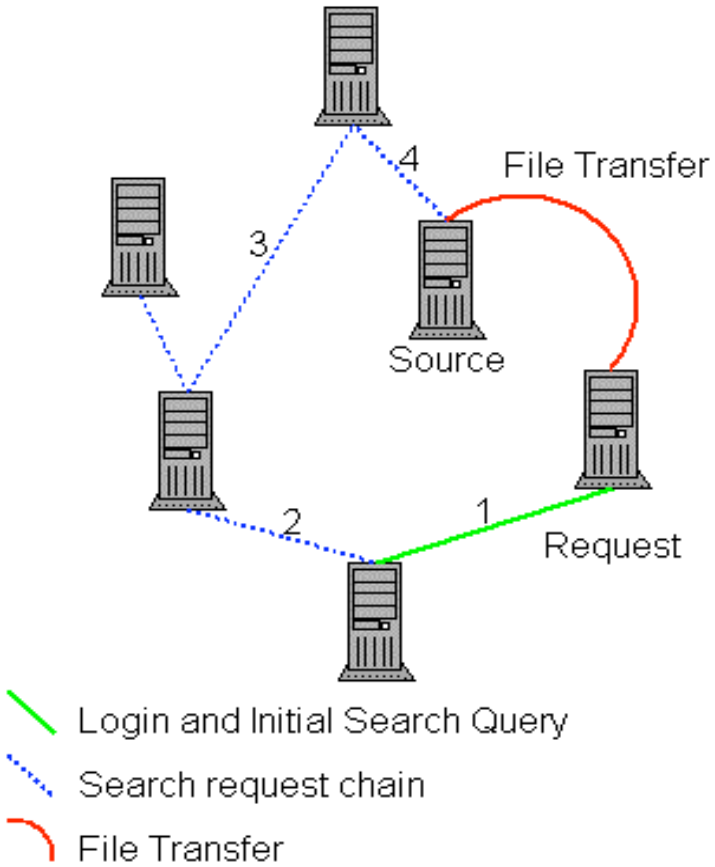
Filesharing-Tools unterscheiden sich primär im Mechanismus zum Auffinden der bereitgestellten Informationen.

Generation: Napster



- Zentraler Indexierungsserver
- Denial-of-Service-Angriffe
- Urheberrechtsverletzungen

Generation: Gnutella



- Keine zentrale Instanz
- Anfragen an einen Teil der Knoten
- Nicht alle Knoten werden abgefragt

Generation: Kademlia

Protokoll für peer-to-peer Netze (ca. 2002)

Reglementierung und Festlegung:

- Art und Aufbau des Netzes
- Kommunikation zwischen den Knoten
- Austausch von Informationen

Kademlia: abstrakt

- Nachweisbare Konsistenz und Effizienz des Systems in einer fehleranfälligen Umgebung durch Ansatz der **Distributed Hash Table**.
- Effiziente Lösung des allgemeinen Wörterbuch-Problems durch **XOR-Metrik-Topologie**.
- Toleranz der Knoten-Fehler (ohne user-delay) durch eine spezielle Anordnung der Routing-Tabellen (**k-buckets**) .

Art und Aufbau des Netzes

Zweite virtuelle Netzstruktur über LAN: DHT (anstatt Indexierungsserver)

Indexierung: durch Clients (Hash-Wert des Inhalts)

Kern: User Datagram Protocol (UDP)

- Internetprotokollfamilie (verbindungslos, minimal)
- Zuordnung der Daten der richtigen Anwendung
- Verwendung von Ports (Prozess-zu-Prozess)

Kademlia-Node

- **Kooperation:** Kademlia-Knoten verfügen über hohes Wissen übereinander.
- **Funktion der Knoten:** Informationsbehälter.
- Jeder Kademlia-Knoten hat eine eindeutige B=160 Bit **ID-Nummer,** die ihn im Netzwerk identifiziert.
- Die ID-Nummern werden anfangs für jeden Knoten zufällig generiert.

<key,value>-Paare

- Gemeinsamer virtueller Adressraum für Knoten-ID-Nummer und Schlüssel (*key*) mit 160-Bit Breite.
- **Schlüssel** (*key*) sind eindeutige, DHT-verteilte Repräsentanten der **Informationen** (*value*).
- Knoten, die ein bestimmtes Informationsblock (*value*) **finden wollen**, suchen für key-nächste Knoten-IDs.
- Knoten, die ein bestimmtes Informationsblock (*value*) **speichern wollen**, legen sie auf die key-nächsten Knoten ab.

Generelles Vorgehen

- Weise jedem Knoten im Netzwerk eine zufällig generierte 160 Bit **ID-Nummer** zu.
- Definiere eine **Metrik-Topologie**: erzeuge einen gemeinsamen virtuellen Adressraum für Knoten-ID (Node_ID) und Schlüssel (key) einer Information (value) mit 160-Bit Breite.
- Führe einen **lookup-Algorithmus** aus, um die Knoten zu finden, deren ID am dichtesten zum gesuchten Schlüssel (key) liegt.
- Lege das <key,value>-Paar dort ab.

XOR-Metrik

- Um die durch DHT im Kademlia-Netzwerk gebildeten <key,value>-Paare zu publizieren oder zu finden, benutzt Kademlia den Begriff der Distanz zwischen zwei Bezeichner.
- Gegeben seien zwei 160 Bit Bezeichner x und y. Um die Distanz zwischen x und y zu berechnen, wird in Kademlia die XOR-Funktion bitweise eingesetzt, deren Ergebnis dezimal ausgewertet wird.

$$d(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$$

Intuition der XOR-Topologie

- **Intuition:** Differenzen der höherwertigen Bits zweier Bitsequenzen haben für die tatsächliche Entfernung grössere Bedeutung als die Differenzen der niederwertigen Bits.

- **Geometrische Intuition:** Knoten liegen an den Nachbarn „näher“, die sich im gleichen Unterbaum befinden, als an den Knoten, die in anderen Unterbäumen sind.

Beispiel: 4 Bit XOR D-Tree

- 0011
- 0110

Distanz: 1+4=5

Triviale Eigenschaften

1. $d(x, x) = 0$ (Identität)

2. $d(x, y) > 0$ mit $x \neq y$ (Existenz)

3. $\forall\, x, y : d(x, y) = d(y, x)$ (Symmetrie)

Weitere Vorteile der XOR-Funktion

1. $d(x, y)+d(y, z)\geq d(x, z)$ (Dreiecksungleichung)

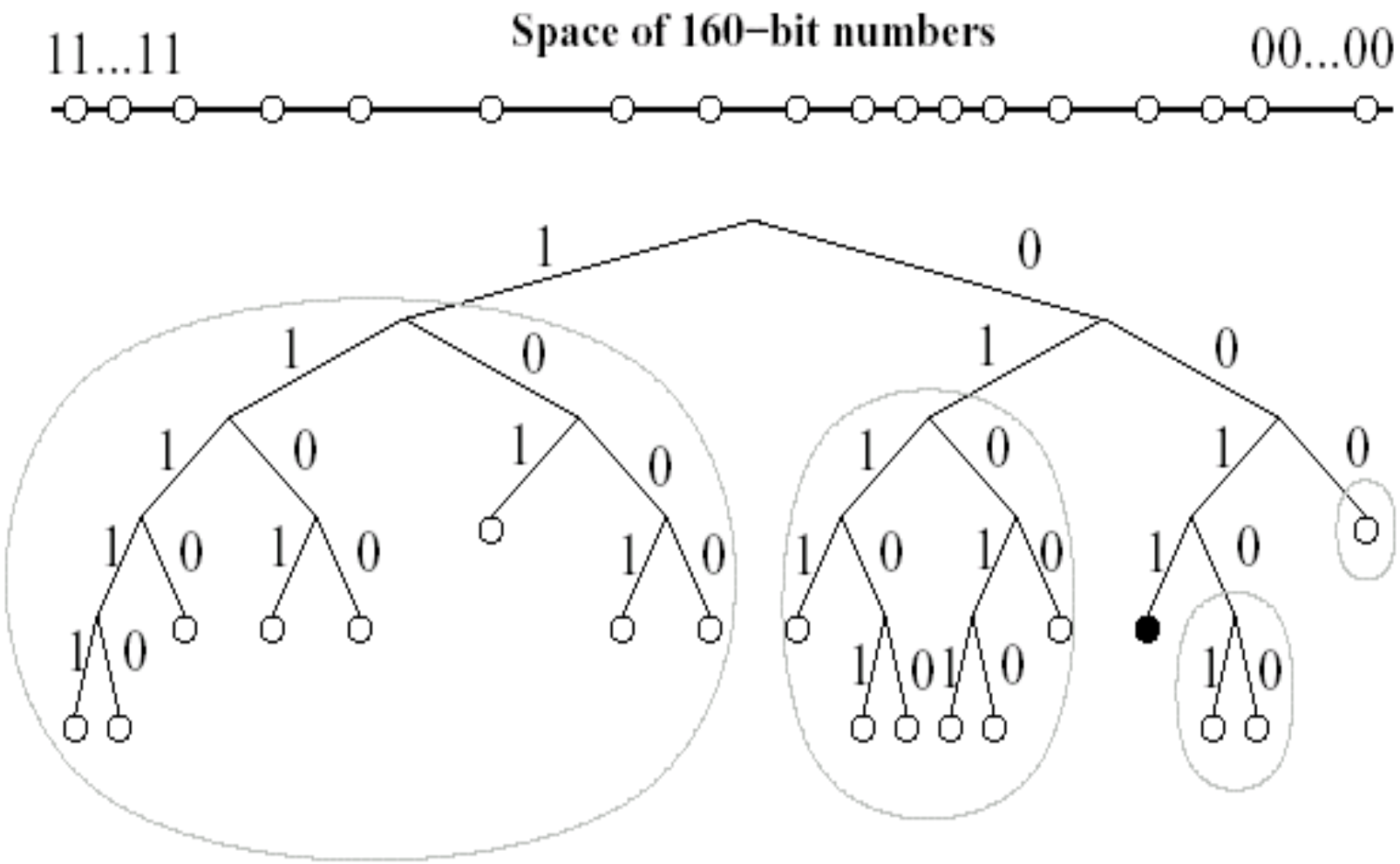
2. $\exists\, y: d(x, z)=d(x, y)$ xor $d(y, z)$ (Transitivität)

Unidirektionalität der XOR-Funktion

- XOR ist eine **unidirektionale Metrik**: für jeden gegebenen Punkt x und Distanz ε>0, existiert genau ein Punkt y, so dass d(x,y)=ε.
- Diese Eigenschaft garantiert, dass alle lookups für den selben Schlüssel gegen einen und den gleichen Pfad konvergieren – unabhängig davon, welcher Knoten die lookup-Anfrage gesendet hat.
- Somit werden die entlang eines lookup-Pfades liegenden <key,value>-Paare zwischengespeichert und können bei einer identischen lookup-Anfrage wieder aufgerufen werden.

Routing-Tabellen

- Knoten beinhalten Kontakt-Informationen übereinander, um Anfrage-Nachrichten untereinander zu verschicken.
- Jeder Knoten führt **160** Listen mit jeweils **k** Kontakten (IP-Adresse, UDP-Port, Knoten-ID)
- **k** ist ein systemweiter Parameter, der so gewählt wird, dass ein Ausfall aller Knoten eines solchen k-buckets innerhalb einer Stunde unwahrscheinlich ist. In der Praxis: **k=20**.


Space of 160−bit numbers
11...11
00...00

Organisation

- Die k-buckets eines Knotens werden nach der Distanz zwischen dem Knoten und seinen Kontakten organisiert.

Ein Knoten (*node*) besitzt **B=160** 20-buckets.

Für ein bucket (*j*) und seine Kontakte (*contact*) gilt:

$$\forall\, 0 \le j < B : 2^j \le d(node, contact) < 2^{j+1}$$

Beispiel

Bucket **Nr. 0** beinhaltet die Kontakte eines Knotens, die um $2^0, \ldots, 2^1$ von dem Knoten entfernt sind.

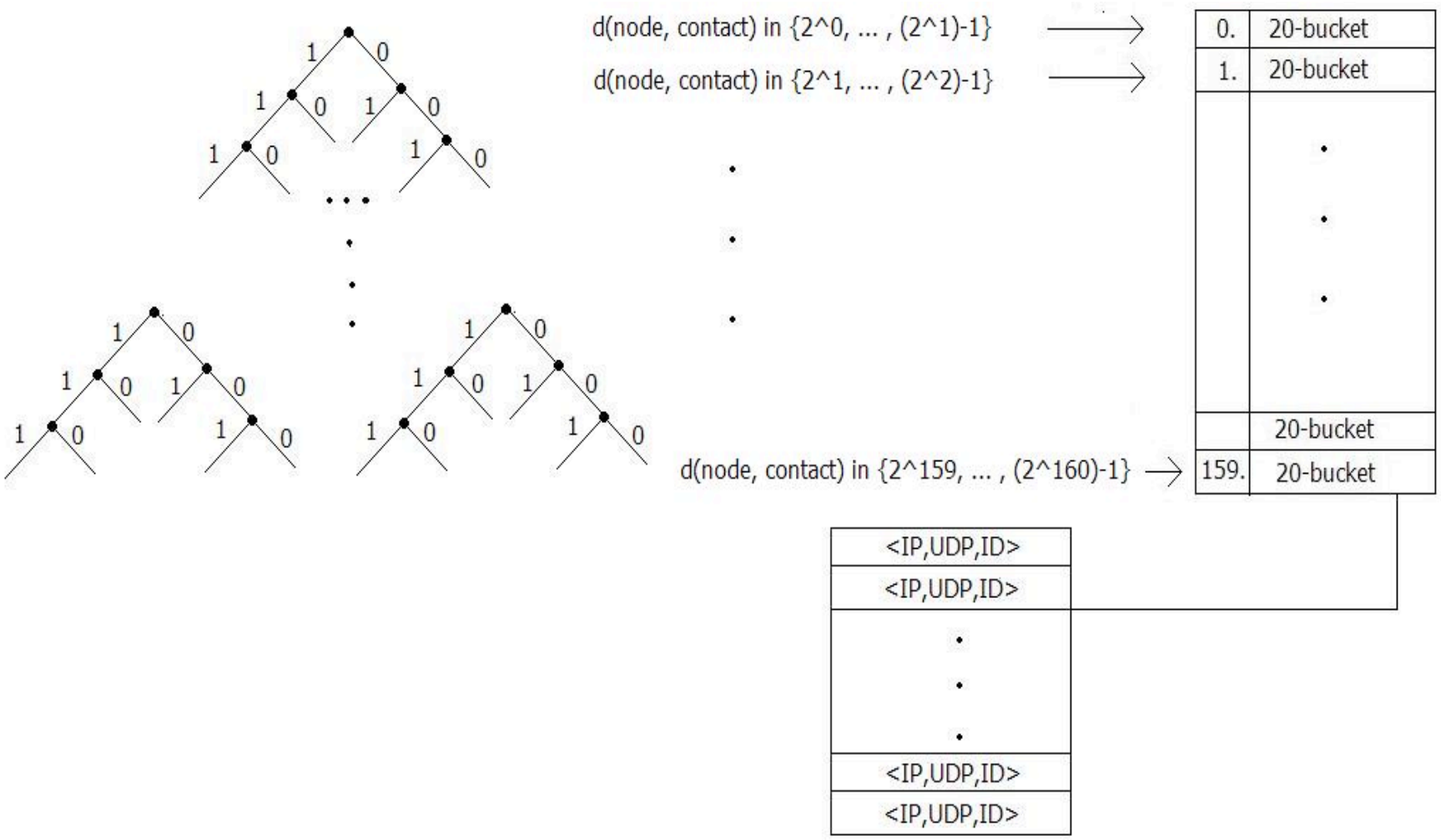
.

.

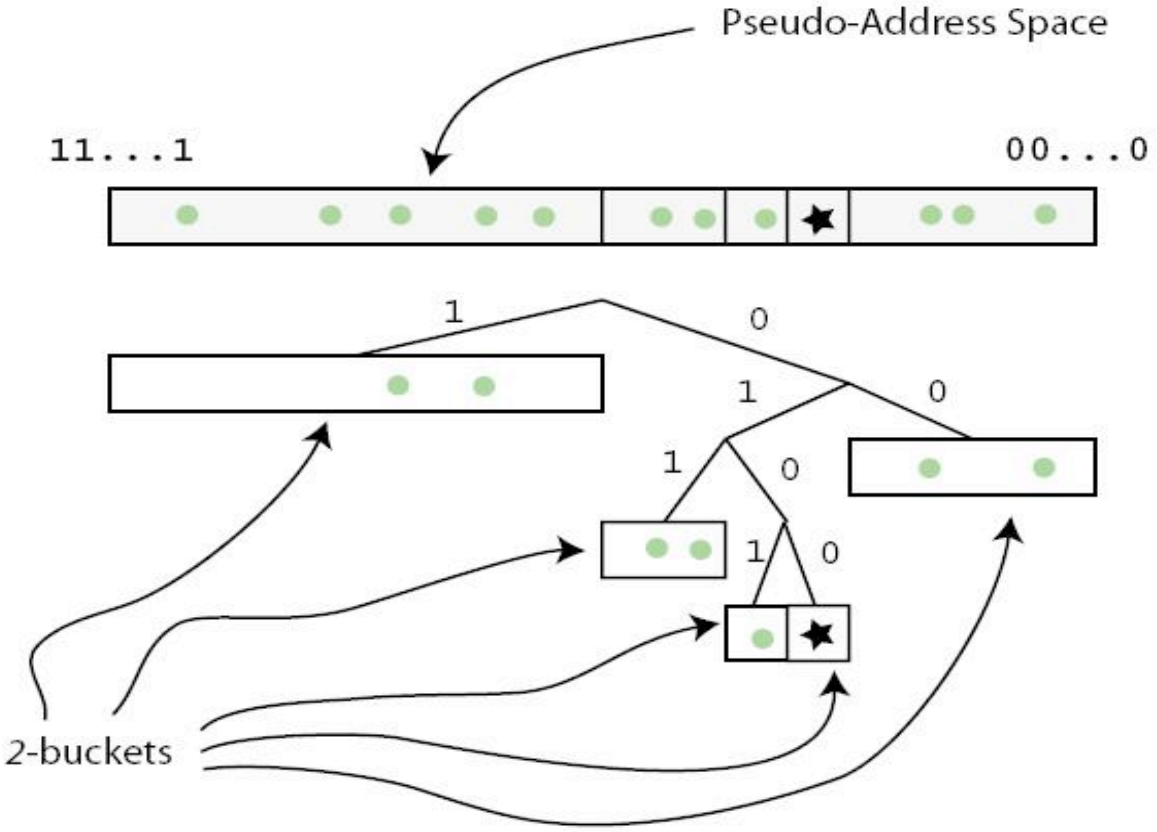
.

Bucket **Nr. 159** beinhaltet die Kontakte eines Knotens, die um $2^{159}, \ldots, 2^{160}$ von dem Knoten entfernt sind.

Strukturierung der 20-buckets



Beispiel: k=2

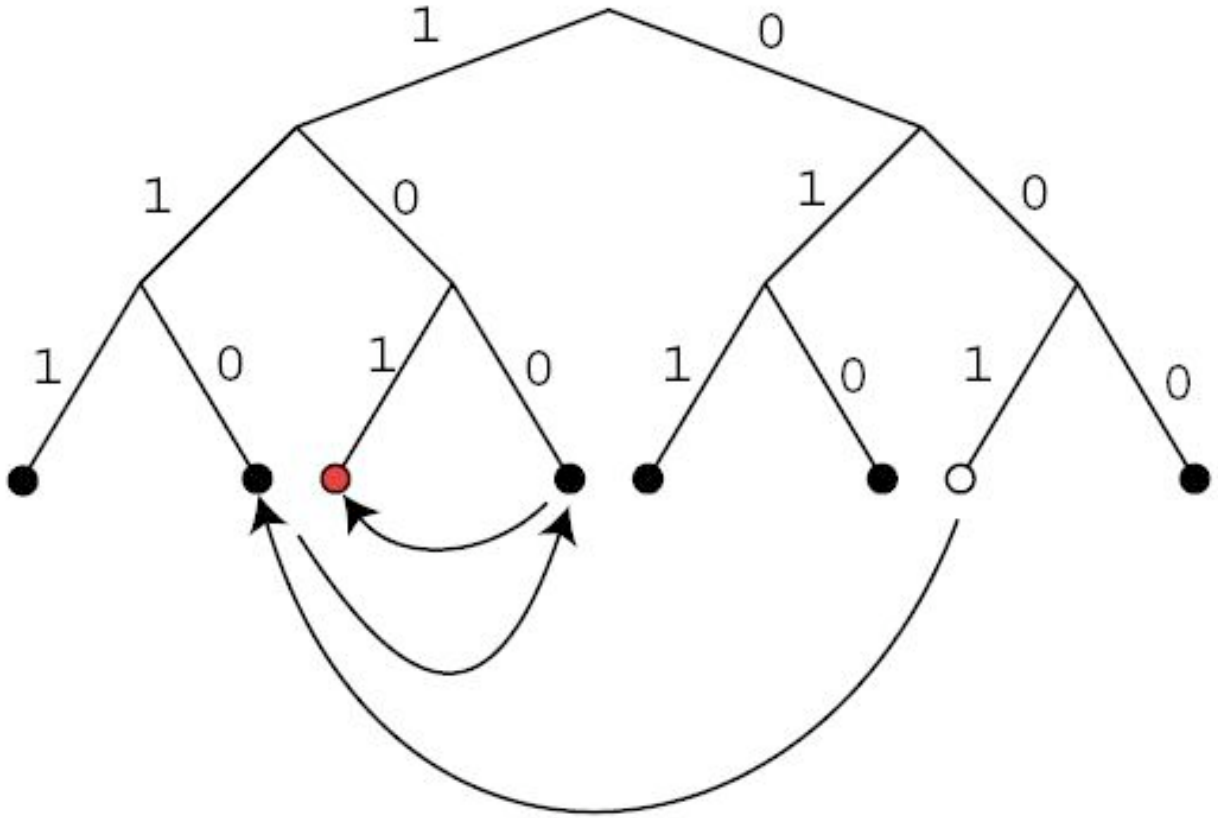


Folgerung

- Für kleinere Werte *j* sind generell die k-buckets fast leer.
- Für große Werte *j* können die Listen bis zur Größe k=20 wachsen.
- Ein Knoten verfügt über ein hohes Wissen über seine Nachbarn, kennt aber nur wenige Knoten, die sehr weit entfernt von ihm sind.

$$|\{2^i, \ldots, 2^{i+1}\}| < |\{2^{i+1}, \ldots, 2^{i+2}\}| \quad , \quad 0 \leq i < 160$$

Komplexität



LS-Strategie

- Die Knoten innerhalb eines 20-buckets werden nach **last seen** Strategie sortiert.
- Knoten, mit denen am längsten nicht kontaktiert wurde, befinden sich ganz am Anfang eines 20-buckets (***head***).
- Knoten, mit denen neulich kontaktiert wurde, stehen ganz am Ende eines 20-buckets (***tail***).

Szenario

- Ein Kademlia-Knoten erhält von einem anderen Knoten im Netz eine Nachricht (Anfrage oder Antwort).
- Ein entsprechender 20-bucket des Empfängers wird mit der ID-Nummer des Senders aktualisiert.

- **Fall 1:** Sender-ID ist im 20-bucket des Empfängers vorhanden. Sender-ID kriegt da die neue Position ***tail***.

- **Fall 2.1:** Sender-ID ist im 20-bucket des Empfängers nicht vorhanden. Der entsprechende 20-bucket ist noch nicht voll. Sender-ID wird am ***tail*** der Liste neu eingefügt.

- **Fall 2.2:** Sender-ID ist im 20-bucket des Empfängers nicht vorhanden. Der entsprechende 20-bucket ist schon voll. Der Empfänger sendet seinem aktuellen ***head*** Knoten ein ***ping*** Signal.

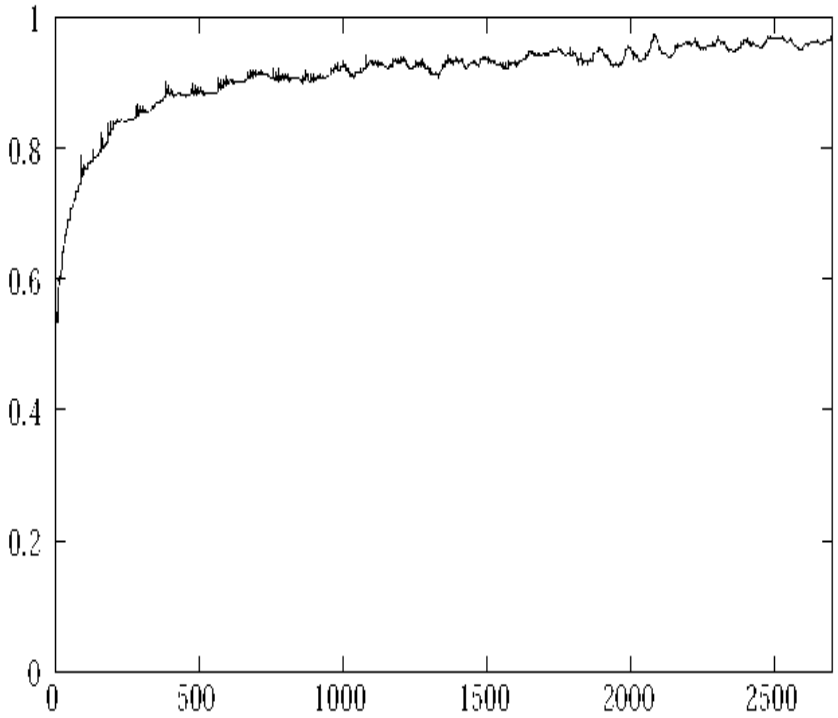
- **Fall 2.2.1:** Der Knoten am ***head*** antwortet nicht. Er wird aus der Liste eliminiert und der neue Sender-Knoten wird in die Liste am ***tail*** eingefügt.

- **Fall 2.2.2:** Der Knoten am ***head*** anwortet. Ihm wird die neue Position ***tail*** in der Liste zugewiesen. Die Anfrage des neuen Sender-Knotens wird dann abgelehnt.

Problem der Instabilität

- **Ideal-Fall:** Sobald ein Knoten dem Kademlia-Netz beigetreten ist, verlässt er es nie.
- **Real-Situation:** Ein zufällig ausgewählter online-Knoten bleibt weiterhin noch eine Stunde länger online mit Wahrscheinlichkeit ½.
- Daraus resultiert die **uptime function.**

Uptime Probability



- **x-Achse:** Minuten
- **y-Achse:** Die W-keit, dass ein beliebiger Knoten weiterhin x+60 Minuten online bleibt, wenn er schon x Minuten online geblieben ist.

Vorteile

- Je länger ein Knoten aktiv bleibt, desto wahrscheinlicher ist es, dass er sich noch eine Stunde länger online aufhält.
- **Idee:** halte die längsten online Kontakte durch *Cashing-Strategie* immer zugriffsbereit.
- Überfüllung der Routing-Tabellen ist durch den Ansatz der 20-buckets unmöglich.

Protokoll: Instruktionen

- **PING**

Ermittelt, ob ein Knoten online ist.

- **STORE**

Speichert ein <key, value>-Paar auf einem Knoten ab.

- **FIND_NODE**

Ermittelt die k (=20) nächsten Knoten zu einer gegebenen Knoten-ID.

- **FIND_VALUE**

Sucht nach einem bestimmten Schlüssel (key) in <key,value>-Paare

Knotensuche mit FIND_NODE

- **Ziel:** lokalisiere rekursiv maximal k viele Tupel <IP,UDP,ID>, deren Knoten-IDs am dichtesten an der gesuchten Ziel-Knoten-ID liegen.
- **Argument T**: 160 Bit Target-Node-ID.
- **Vorgehen:** Für jede Stufe i führe FIND_NODE RPC um die jeweiligen k nächsten Knoten zum **T** rekursiv zu befragen.
- **Im Folgenden:** lookup-Algorithmus (O(log n))

Lookup algorithm skeleton

- **Goal:** Find the k nodes closest to a given target $T \in \{0,1\}^{160}$
- **RPC:** $\mathsf{find\_node}_n(T)$ returns all contacts from the (first non-empty) k-bucket in n's routing table that is closest to T
- **Lookup:**

  $n_o =$ ourselves (the node that is performing the lookup)

  $N_1 = \mathsf{find\_node}_{n_o}(T)$

  $N_2 = \mathsf{find\_node}_{n_1}(T)$

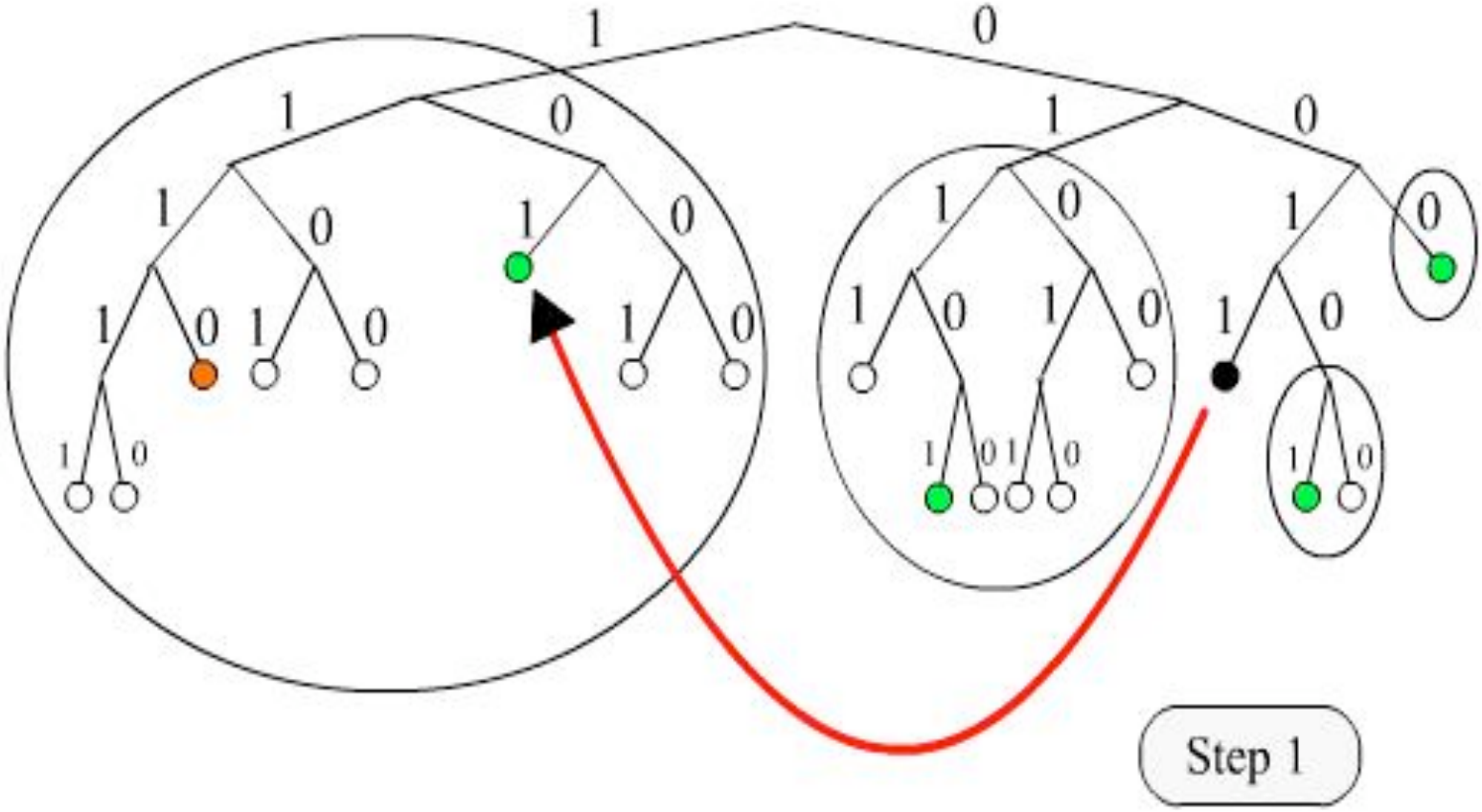
  $\ldots$

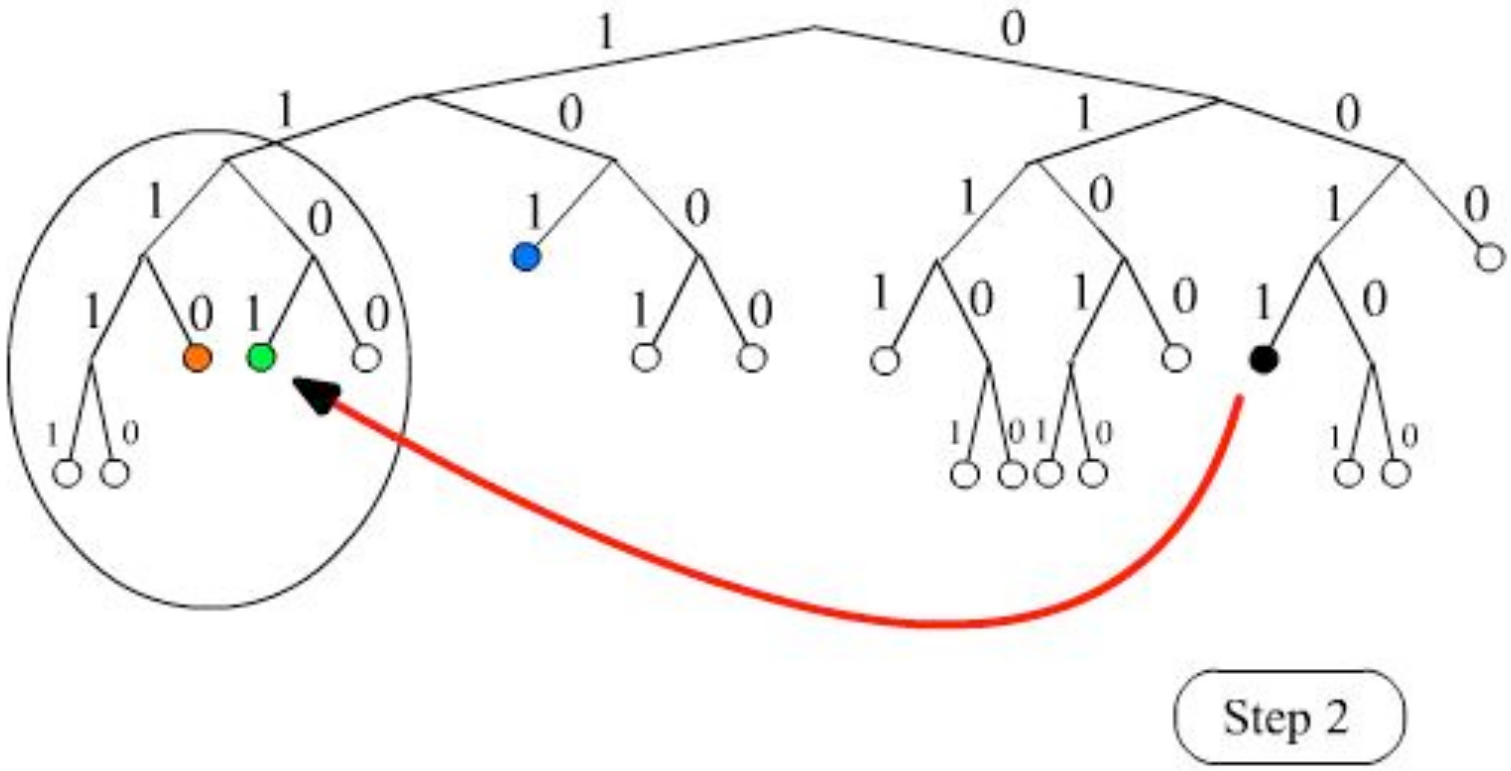
  $N_l = \mathsf{find\_node}_{n_{l-1}}(T)$,

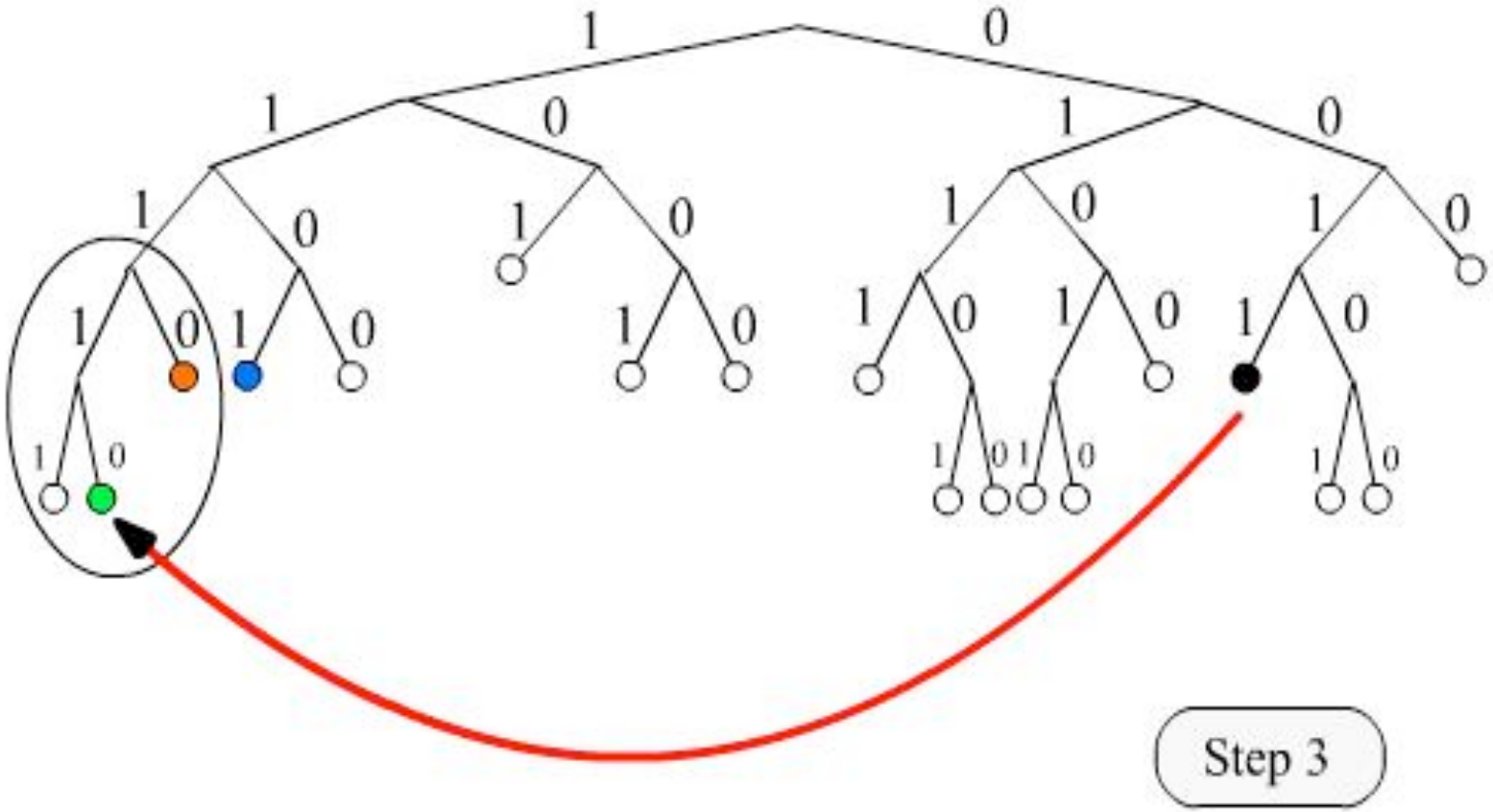
  this completes when N_l contains no contacts that haven't been called already

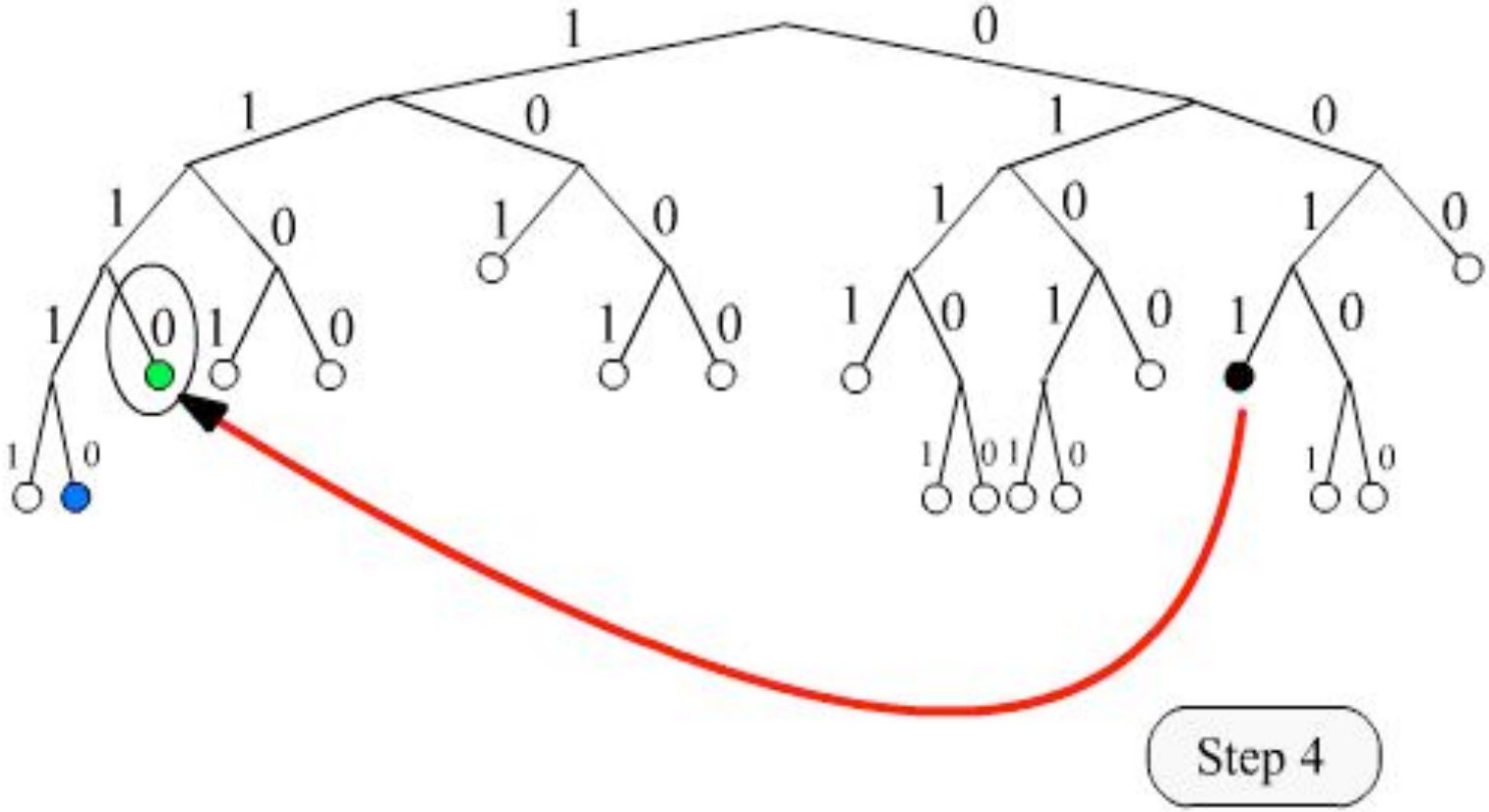
- n_i is any contact in N_i

Illustration

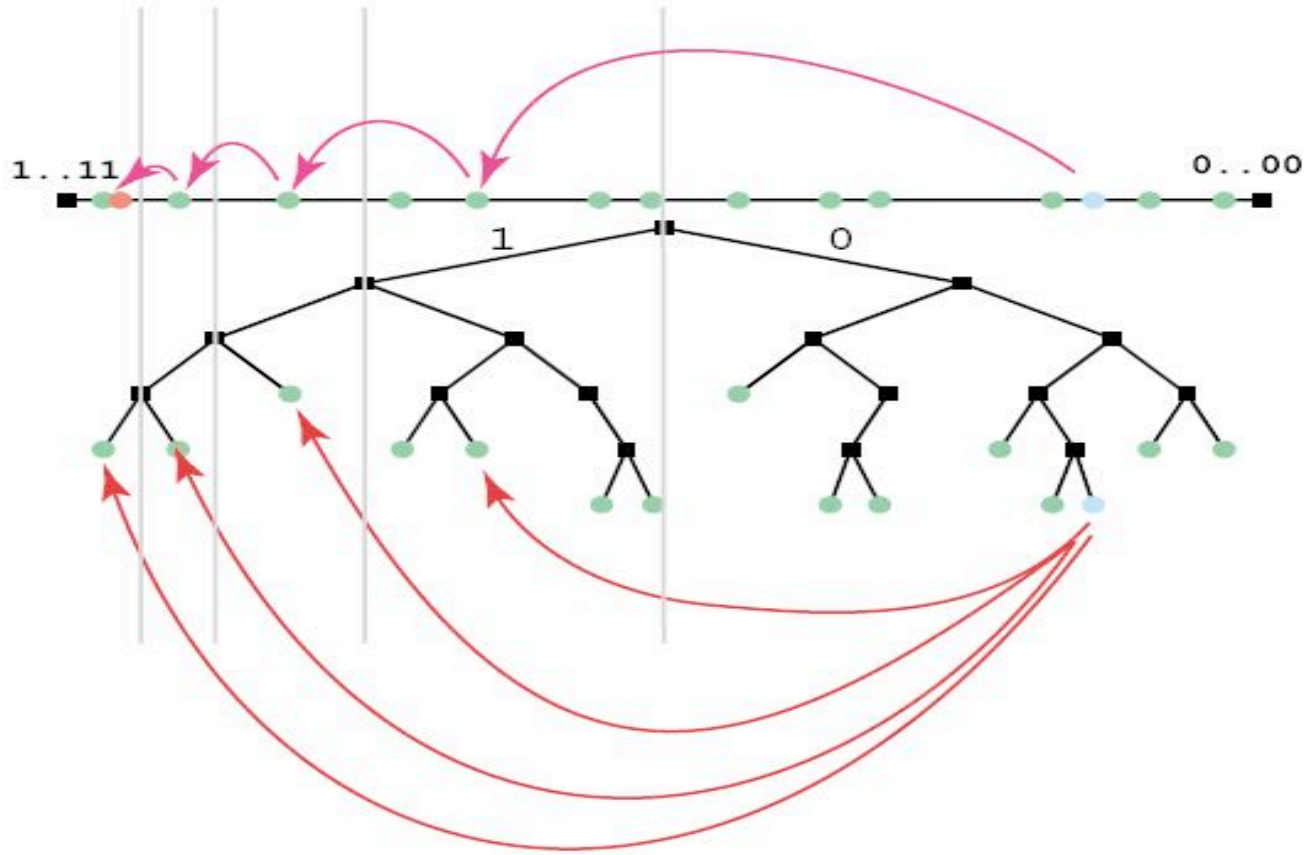



Step 2


Step 3


Step 4

Komplexität O(log n)



Eigenschaften

- In jedem Schritt der lookup-Iteration wird die XOR-Metrik-Distanz zwischen n_i und T um Faktor ½ reduziert.
- Die Menge der potenziellen Kontakt-Kandidaten wird somit auch im jeden Schritt um Faktor ½ kleiner.

Erweiterung des Algorithmus

Idee: Nutze Parallelität.

- Wähle α viele Kontakte aus einem (nicht leeren) Ziel-ID nächsten k-bucket.
- Sende parallel FIND_NODE an alle α Knoten. Jeder der Knoten gibt k viele Tupel zurück.
- Aus diesen α*k vielen Tupel wähle wieder α viele Ziel-ID nächsten Knoten.
- k viele Tupel <IP,UDP,ID> werden an den *lookup*-Initiator zurückgegeben.

In der Praxis: α=3

Publizierung von <key,value>

- Führe **FIND_NODE** Prozedur nach dem entsprechenden Schlüssel (*key*) aus: *k* viele key-nächsten Knoten werden bestimmt.
- Mit Hilfe von **STORE** RPC wird das entsprechende <key,value>-Paar auf diesen Knoten abgelegt.
- “Freundschaftsbedingung” zwischen w und u
- **Vorteil:** der Lebenszyklus eines Knotens im Netz bleibt immer konsistent.

Aktualität von <key,value>

- Verfallzeitpunkt der Schlüssel-Wert-Paare: nach 24 Stunden.
- Aktualisierung im 24 Stunden-Takt durch STORE RPC von dem Inhaber nötig.
- Abspeicherung der Schlüssel-Wert-Paare im Stunden-Takt durch den Besitzer-Knoten.

Schlüsselsuche mit FIND_VALUE

- Es wird nach einem Schlüssel gesucht (key), der einen konkreten Inhalt (value) repräsentiert.
- Der Knoten startet eine parallele lookup-Anfrage und bekommt in der Regel die k key-nächsten Knoten zurück.
- Besitzt jedoch einer der angefragten Knoten das gesuchte Schlüssel-Wert-Paar, bricht die Suche ab.

Bootstrapping

- Knoten ***u*** hat anfangs keine Kenntnisse über seine nahe Umgebung und demnach keine Einträge in seinen *k-buckets*.

Vorgehen:

- Generiere eine eigene Knoten-ID mit einer konsistenten Hash-Funktion.
- Führe einen Knoten-lookup nach Node-ID(u) im Netzwerk durch.
- Aktualisiere die *k-buckets* von u mit den nächten Nachbar-Knoten.
- Aktualisiere auch die *k-buckets* seiner Nachbar-Knoten.

Vorteile des Kademlia-Systems

- Die Steigerung der **Resistenz gegen DoS-Attacken** durch die Nutzung der dezentralen Indexierungsstruktur: dynamisches Netz.
- Realisierung der Such-Funktion in einem strukturierten *peer-to-peer*-Netz mit **logarithmischer Komplexität**.
- **Cashing** der Schlüssel-Wert-Paare: auch beim Ausfall vieler beteiligter Knoten ist keine wesentliche Latenzsteigerung zu erwarten.

Verbreitung

- Overnet
- eMule
- BitTorrent

Vielen Dank!

Martynas Ausra