# Verteilte Systeme

## Peer to Peer Systeme

# Peer to Peer Systeme

Problem:

Client/Server Systeme bieten exzellente Kontrolle über einen Dienst, jedoch können sie nur eine begrenzte Anzahl an Anfragen beantworten und müssen erweitert bzw. repliziert werden – bei unerwartet starker Anfrage schwer zu realisieren.

Idee:

Die Aufgabe des Servers in die Clients integrieren, so dass die verfügbaren Server mit den Clients immer skalieren.

# Beispiel: Filesharing/Napster

Erste prominentes Protokoll (1999), dass als "Peer-to-Peer" bezeichnet wurde, 2001 nach Rechtsstreit abgeschaltet.

Anwendungsfall:

Nutzer können Dateien bei anderen Nutzern lokalisieren und herunterladen.

Idee:

- Verzeichnis der Verfügbaren Daten (klassischer Client/Server-Betrieb)
- Datentransfer erfolgt direkt zwischen Nutzern

Inzwischen existiert ein kommerzieller Musikdienst unter dem Namen.

# Beispiel: Filesharing/ Gnutella

Reaktion auf die Abschaltung von Napster
vergleichbarer Dienst, bei dem es keinen zentralen Server gibt der abgeschaltet werden könnte.

Idee:

- Teilnehmer suchen sich einen Einstiegspunkt in das Gnutella-Netz
- Suchanfragen werden durch das Overlay-Netz geflutet
  Optimierung: “Blätter” leiten im Gegensatz zu inneren Knoten keine anfragen weiter.
- Datentransfer erfolgt direkt zwischen Nutzern

# Beispiel: Filesharing/ Bittorrent

Ziel: Möglichkeiten von Peer-to-Peer Datentransfer für große Datenmengen in vielen Anwendungen nutzen.

Idee:

- Offenes, erweiterbares Protokoll
- Segmentierter Datentransfer
- Absicherung der Integrität der Daten durch kryptographische Hashes
- Suche nach Dateien aus dem Protokoll gelöst
  -> .torrent-Dateien enthalten Beschreibung
- Zentrale Server “tracker” dienen als Verzeichnis, welche Knoten Segmente verfügbar halten
- Datentransfer erfolgt direkt zwischen Nutzern

# Segmentierter Datentransfer

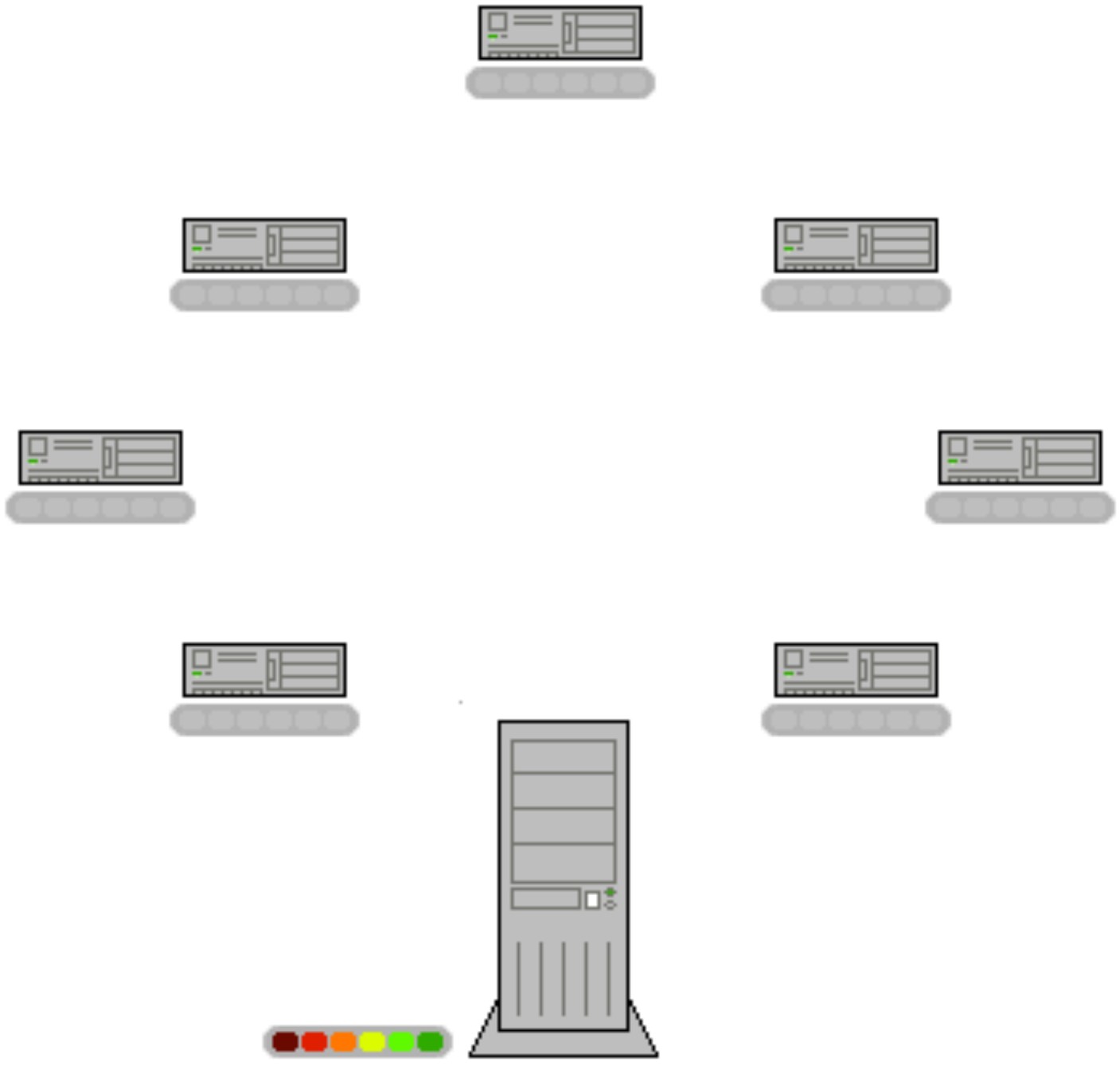
Problem:

Es soll eine große Datenmenge übertragen werden,
bei der eine erfolgreiche Übertragung am Stück
unwahrscheinlich ist (Raumsonde-Messdaten, DVD-Image).

Idee:

- Segmentierung der Datei
- Übertragung der Prüfsummen einzelner Segmente
- Übertragung der Segmente ggf.
  - in beliebiger Reihenfolge
  - von verschiedenen Knoten aus
- Validieren der Segmente, ggf. erneute Übertragung
- Zusammensetzen der Datei.

# Segmentierter Datentransfer



Quelle: http://en.wikipedia.org/wiki/File:Torrentcomp_small.gif

# Beispiel: Filesharing/ moderne Peer to Peer Systeme

- Verzicht auf zentralen Server
- Verwenden meist  ins Protokoll integrierte Verteilte Hashtabellen (DHT) als Verzeichnis

- Geschäftsmodelle:
  - Werbung
  - ISP-Bandbreitenoptimierung
  - Integration (z.B. als Update-Engine für Computerspiele)

# Verteilte Systeme

## Verteilte Hashtabellen (DHT)

# Verteilte Hashtabellen (DHT)

Problem:

Verzeichnisdienste sind hierarchisch (Skalierbarkeit) und benötigen zuverlässige bzw. vertrauenswürdige Infrastruktur (Verfügbarkeit / Integrität).

Wie baut man einen Verzeichnisdienst, der keine zentralen Schwachpunkt hat?

Idee:

Den Verzeichnisdienst über viele gleichberechtigte Knoten verteilen.

Einfaches Speichern von Key/Value Paaren

-> flacher Namensraum.

# Verteilte Hashtabellen (DHT)

neues Problem:

Wie verteilt man die Daten auf die Knoten?

Wie stellt man sicher, dass die Last gleichmäßig auf die Knoten verteilt wird

Idee:

Wir verwenden eine Hash-Funktion über die Schlüssel, um die Daten möglichst gleichmäßig zu verteilen.

Verfügbarkeit wird über hohen Replikationsgrad erreicht.

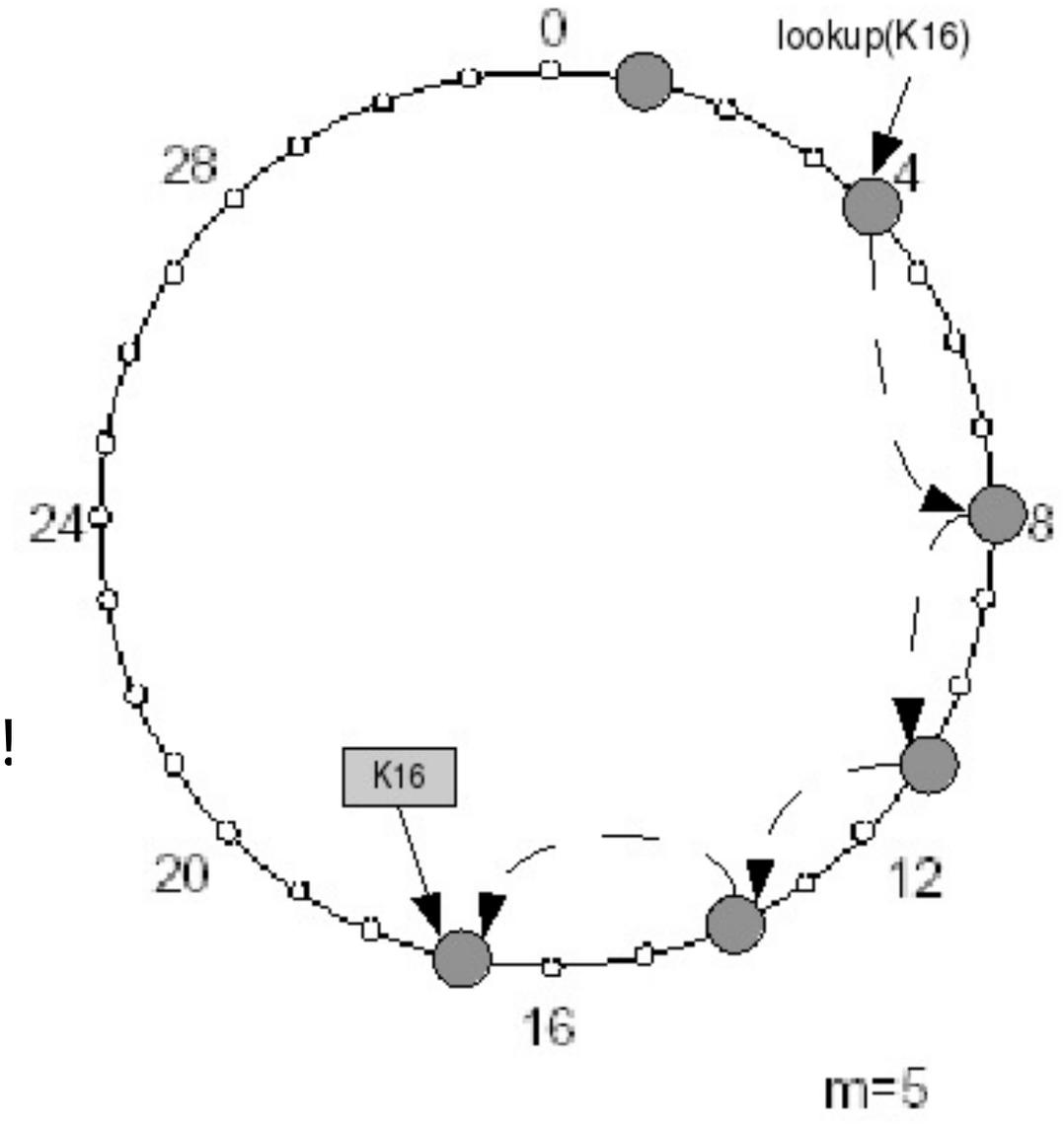
# Chrod

Lineare Suche:

Jeder Knoten kennt seinen Nachfolger.

Die Suche folgt so lange den Verweisen auf den Nachfolger, bis der zuständige Knoten erreicht wird.

Nachteil: Skalierbarkeit!



# Chrod

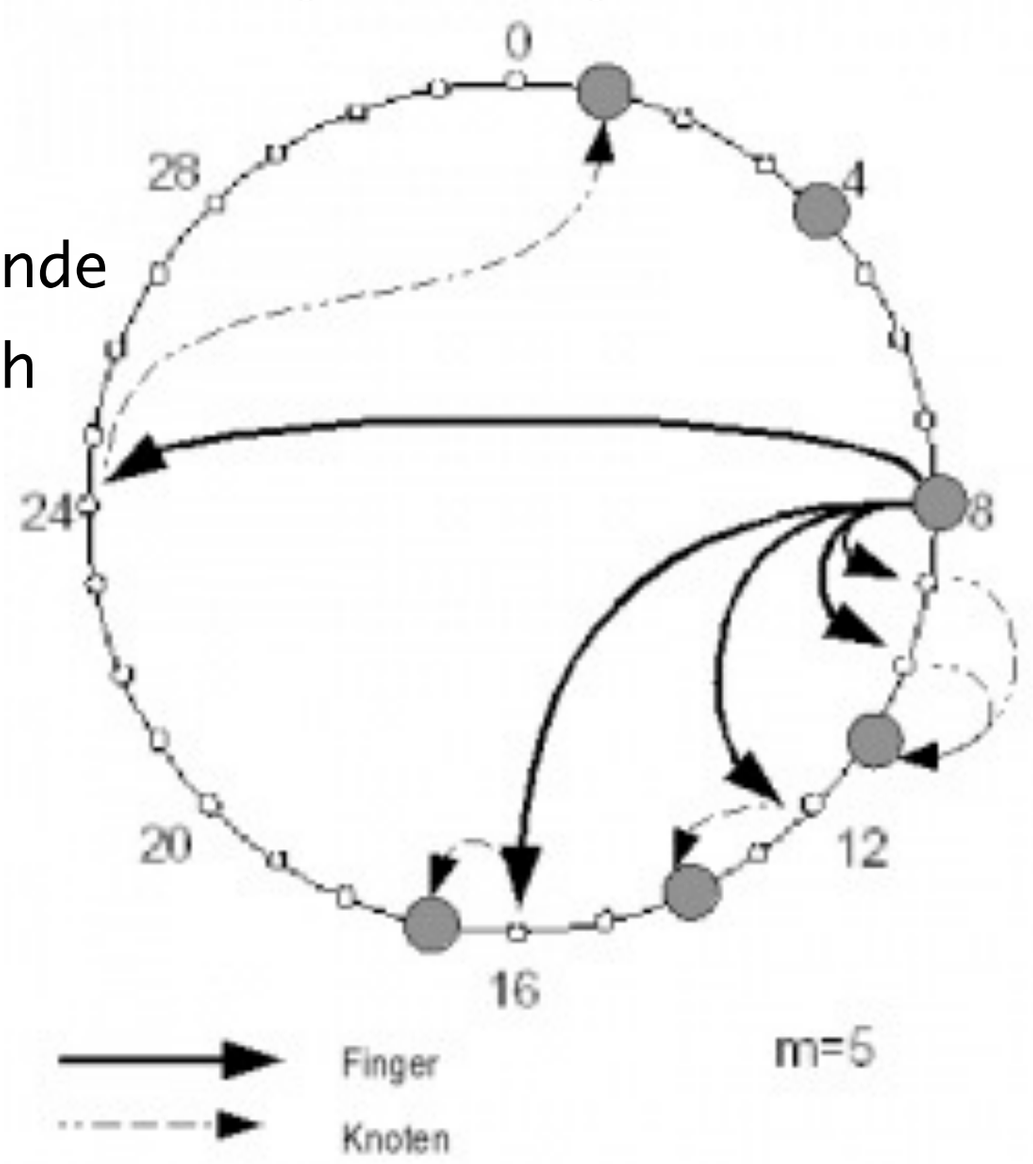
Optimierte Suche (log m):

Jeder Knoten k kennt log m Nachfolger (Finger) für die Werte $(k+2^{i-1}) \bmod m$.

Vorteil: Die der verbleibende Suchbereich halbiert sich in jedem Suchschritt.

Nachteil: Es müssen mehr Zeiger verwaltet und aktualisiert werden.

Jederzeit Fallback auf lineare Suche möglich!



# Chrod

Einfügen eines Wertes x:

Suche den kleinsten Knoten i mit i > x und Speichere x und die Daten zu x in i

Einfügen eines Knotens j:

- Suche den kleinsten Knoten i mit i > j.
- Setzte Nachfolgern von j auf i
- Teile i mit, dass sein Vorgänger jetzt j ist.
- Verschiebe die Werte x aus i mit x < j nach j.

# Chrod

Aktualisieren der Nachfolgerrelation für Knoten j:

- Frage regelmäßig den Nachfolger i.succ nach seinem Vorgänger j.succ.pred.
- Ist j.succ.pred != j:
  setzte j.succ := j.succ.pred.

Aktualisieren der Finger für Knoten j:

- Suche regelmäßig nach $j+2^i$ für alle $i \in (0, \log(m))$
- Aktualisiere den Zeiger im Finger auf das Suchergebnis.

Bootstrapping der Finger für Knoten j:

- Kopiere die Finger von j.succ (und geringer Unterschied, aktualisiere später)

# Diminished Chrod

Problem:

Wir wollen den Chrod-Ring für verschiedene Anwendungen nutzen.

Idee:

Wir betten die Werte der Anwendung als Untergruppe in den Ring ein, so dass die Finger im Ring als Suchbaum in der Untergruppe verwendet werden können.

Die Werte der Eltern im Suchbaum lassen sich durch arithmetische Operationen bestimmen und wie gewohnt suchen.

# Kademlia

Problem:

Chord erfordert auf Grund der starken Struktur sehr viel Wartungsaufwand.

Wir wollen die garantien der Datenstruktur aufweichen, aber trotzdem die Suchlaufzeit beibehalten.

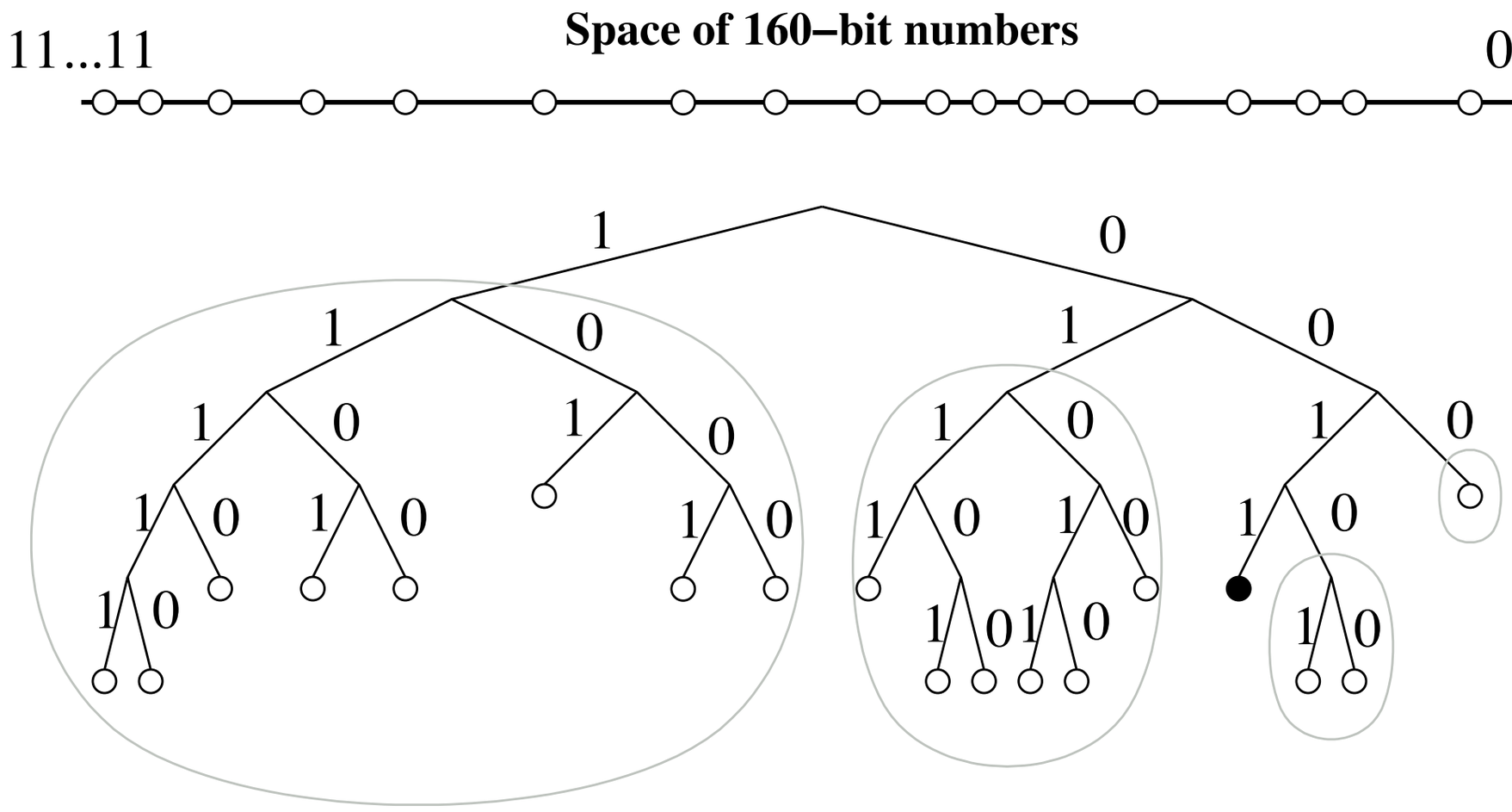
Idee:

Verwende XOR als Abstand zwischen zwei Knoten:

- x XOR x = 0
- x XOR y = y XOR x
- (x XOR y) + (y XOR z) >= (x XOR z)

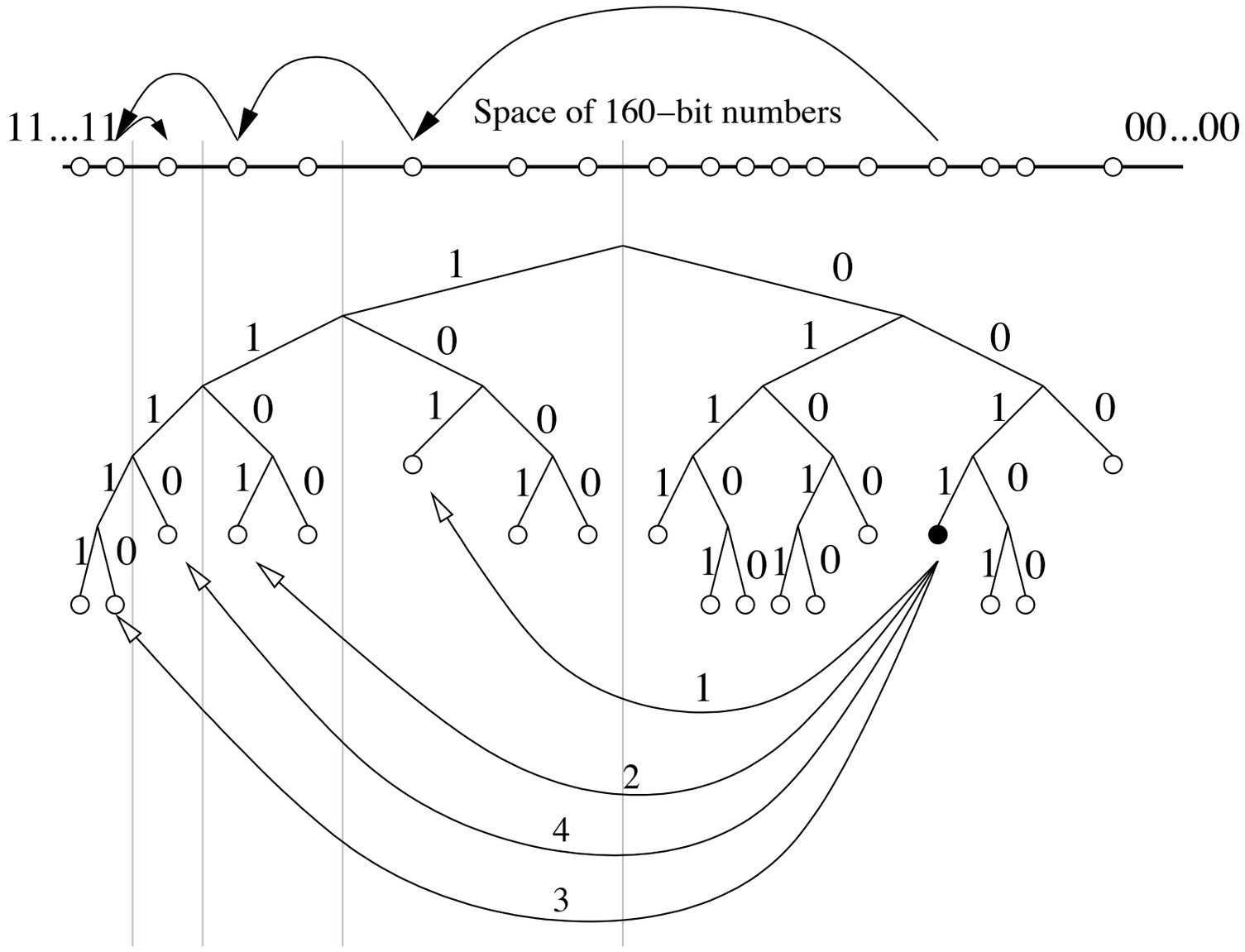
# Kademlia

Jeder Knoten hält eine Liste von j Knoten mit Abstand log(i)




Der schwarze Knoten kennt also aus jedem der umkreisten Unterbäume bis zu j Knoten.

# Kademlia




Optimierung: Die Knoten auf jeder Ebene liefern den dem Ziel nächsten Knoten aus ihrer Liste.

# Kademlia

Wartung:

- Die Listen der Knoten werden nach Alter sortiert
- Knoten in der Liste werden regelmäßig geprüft und bei nicht-Erreichbarkeit aus der Liste entfernt
- Da von “alten” Knoten erwartet wird, dass sie besonders zuverlässig sind, werden sie bevorzug in der Liste behalten.
- Neue Knoten landen in einem “Replacement-Cache” (LRU) und ersetzen ggf. nicht mehr erreichbare Knoten.

# Probleme von DHTs

Sybil-Angriffe:

Massenhaftes erzeugen von knoten, um möglichst Große Teile der DHT unter eigene Kontrolle zu bringen.

DoS-Angriffe:

Publizieren von sinnlosen Werten.

# Anwendungsbeispiele

- Content Distribution
- Caching (Codeen, IBM WebSphere eXtreme Scale)
- Verteilte Datisysteme (OceanStore)
- Verteilte Suchmaschine (Fardoo, BTDigg)
- Zensuresistente Overlaynetz (Freenet, GNUnet)

# Übung 7 zum 5. Juli 2011

Wir werden über die nächsten Übungszettel eine kleine verteilte Peer-to-Peer Wirtschaftssimulation schreiben.

Jeder Knoten soll nun entweder einen Planeten, auf dem verschiedene Handelswaren angeboten werden,

oder ein Handelsraumschiff darstellen.

Wir erweitern das Raumschiff nun um die letzten fehlenden Kommandos:

start <ip> <port> legt den Startort fest.

travell <planet> reist zum Nachbarplanet <plante>

buy <good> <amount> kauft eine Handelsware

sell <good> <amount> verlauft ein Handelsware

Jede Reise kostet den Absolutbetrag der Portdifferenz an Treibstoff (Gold)