# Pastry: Ein skalierbarer und verteilter Location-Service

Timo Holm
timo.holm@fau.de

**Kurzzusammenfassung**

Pastry stellt auf Applikationsebene einen Objekt-Lokations- und Objekt-Routing-Dienst zur Verfügung, der vollständig dezentralisiert, skalierbar und selbst organisierend ist. Pastry vermeidet offensichtliche Schwächen populärer Peer-to-Peer Dienste und kann lokale Gegebenheiten der Netzwerkinfrastruktur berücksichtigen um das Routing zu beschleunigen. Pastry bietet gute Performanz bei gleichzeitig einfacher Handhabung und ist auf verschiedenen Plattformen implementiert und benutzbar. Darauf aufbauend realisiert das Projekt Past einen verteilten Dateidienst, der in das Netzwerk eingebrachte Daten effizient wiederfindet. Das Projekt Scribe bildet ein verteiltes Nachrichtenpublikationssystem und ist durch die Verwendung von auf Pastry aufbauendem Multicastverkehr für eine große Anzahl an Lesern/Nachrichten geeignet.

## 1 Motivation

Spätestens seit dem Fall Napster sind File-Sharing- und damit Peer-to-Peer-Systeme in das Auge der Öffentlichkeit gebracht worden. Durch Copyrightprobleme, schier endlose Gerichtsverhandlungen und die schlussendliche Abschaltung des offiziellen[1] Dienstes wurde der schrankenlose Austausch von Daten aber ebenso schnell ins Zwielicht gerückt wie die Möglichkeiten der verteilten Datenhaltung erkannt wurden.

Gnutella [2] trat über Nacht an Napsters Stelle und überwand durch *echte* Dezentralisierung die Abhängigkeit von Betreibern. Die Schwäche Napsters, durch das Abschalten der zentralen Server unbrauchbar zu werden, kann Gnutella Prinzip bedingt nicht zum Verhängnis werden.

Mit der eingeführten Gleichverteilung der Aufgaben (jeder Knoten ist ebenso Server wie Client - eben *servent*) handelt sich das Protokoll Gnutella aber andere Probleme ein. Durch den Benutzeransturm nach der Abschaltung von Napster zeigte sich in der Praxis sehr schnell, dass Gnutella nur sehr unzureichend skaliert. Für sehr große Teilnehmerzahlen ist es ungeeignet [3]. Dieses Manko versucht seinerseits das Protokoll Pastry [4] unter voller Beibehaltung der Dezentralität zu beseitigen.

## 2 Designziele

Um die Fehler der beiden etablierten Lösungen zu vermeiden gilt es die allgemeinen Probleme im Zusammenhang mit Peer-to-Peer Protokollen zu erkennen und durch geeignete Ansätze zu umgehen. Dabei lassen sich drei wesentliche Punkte herausarbeiten um vorhandene und zukünftige Entwicklungen aneinander zu messen.

1. inoffiziell vgl. OpenNAP [1]

## 2.1 Dezentralisierung

Fordert man für Peer-to-Peer Systeme vollkommene Dezentralisierung, so ist Napster streng genommen nicht als Solches anzusehen. Im Napsternetz muß es für die Suchanfragen nämlich ausgezeichnete Knoten geben, nicht jeder Knoten hat zu gleichen Teilen die Aufgaben eines Server bzw. Client. Natürlich ist mit einer verteilten Suche, die durch das Fehlen von globalem Wissen notwendig wird, ein hoher Kommunikationsaufwand verbunden.

## 2.2 Skalierbarkeit

Diesen erhöhten Kommunikationsaufwand versucht das Protokoll Gnutella durch eine verteilte Suche nach dem Schneeballprinzip (der Vervielfältigung von Nachrichten) zu erschlagen. Dies ist wie schnell einsichtig wird eine effektive, jedoch keine effiziente Lösung. Es kommt zu einer starken Belastung der unter dem Netzwerk liegenden Infrastruktur, die mit der Zahl der Knoten wächst. Um dies zu verhindern ist es notwendig Nachrichten möglichst direkt an den zuständigen Knoten zu schicken.

Idealerweise bringt man darum in das Netzwerk Struktur, die ein schnelles Auffinden des gesuchten Knoten möglich macht. Ein beliebig chaotisch aufgebautes Netz, durch das normalerweise nicht in vorhersehbar vielen Schritten geroutet werden kann, wird so verwaltbar.

Wo Gnutella Nachrichten an alle Knoten versendet, es also keine gerichtete Ausbreitung gibt, muss ein effizienteres Protokoll folglich Routing einführen. Der jeweils nächste Empfänger soll trotzdem nicht die Struktur des ganzen Netzes kennen müssen sondern nur das wiederum nächsten Ziel lokal bestimmen.

## 2.3 Selbstorganisation

Gegen eine total erfasste Struktur spricht, dass das dynamische Weg- und Hinzunehmen von Knoten erschwert würde, da alle Knoten von Ausfällen in Kenntnis gesetzt werden müssten. Genau das ist es aber, was das System effizient leisten und organisieren soll, das eigene Wachsen und Schrumpfen.

# 3 Pastry

Pastry versucht die oben genannten Designziele zu verwirklichen. Es bietet auf Applikationsschicht einen skalierbaren und verteilten Objekt-Lokations- und Objekt-Routing-Dienst in einem potentiell sehr großen Netzwerk. Einzelnen Knoten wird dafür eine Verarbeitungsschicht samt API angeboten, die es ermöglichen soll eine Vielzahl von Peer-to-Peer Applikationen zu verwirklichen (z.B. verteilte Datei- oder Gruppenkommunikationssysteme).

## 3.1 Struktur

Struktur in das Peer-to-Peer Netzwerk bringen bedeutet in erster Linie die Knoten eindeutig unterscheidbar zu machen. Hierfür erhält jeder Knoten (wie auch jede Nachricht) eine 128 Bit Adresse (ID). Das Routing sendet eine Nachricht immer zu dem lokal bekannten Knoten, der numerisch die naheste ID besitzt.

Fordert man effizientes Routing zwischen den Knoten eines vermaschten Netzes kommt man schnell auf die Idee eine binärbaumartige Struktur zu realisieren. Diese ermöglicht theoretisch logarithmischen Aufwand bei der Übermittlung von Nachrichten. Bei einer Schlüssellänge von 128 Bit würden im herkömmlichen Binärbaum aber 128 Hops notwendig um zu einem Blatt zu gelangen, was keine gute Performanz verspricht. Pastry erhöht deshalb die Anzahl der Unterknoten auf $2^b$ (b standardmäßig 4), wodurch der Baum abflacht und im ungünstigsten Fall 32 Hops notwendig sind. Ein Pastry Node kennt also maximal $2^b$ direkte Nachbarknoten, die entsprechend den Ebenen eines Baumes ein gemeinsames Präfix besitzen.

### 3.1.1 Routing

In der so genannten **Routing Tabelle** werden Paarungen von NodeID und Netzwerkadresse (z.B. IP) gespeichert, wobei die *k*-te (bei 0 beginnend) von $2^b$-1 Tabellenzeilen immer NodeIDs mit gleichem Präfix enthalten die mit den *k* ersten Stellen der lokalen NodeID übereinstimmen.

Jeder Knoten hat darüber hinaus im **Leaf-Set** *L* NodeIDs von ihm (numerisch) nahen Knoten gesichert (davon $|L|/2$ größer und $|L|/2$ kleiner als die eigene NodeID - $|L|$ typischerweise $2^b$ oder 2 x $2^b$), die quasi den Blättern des zu durchroutenden Baumes entsprechen. Immer wenn der lokale Knoten von neuen Knoten erfährt überprüft er, ob diese näher sind als irgendeiner des Leaf-Set und ersetzt diesen bei Bedarf.

Schließlich kennt jeder Node noch eine Menge *M* mit $|M|$ (typischerweise $2^b$ oder 2 x $2^b$) ihm netzwerktopologisch nahe stehender Knoten, die Anhand einer geeigneten Heuristik (IP-Hops, Ping-Zeit, etc.) als Nachbarknoten im sog. **Neighborhood-Set** stehen.

Der eigentliche Routingvorgang dient dem Weiterleiten der Pakete im Pastry-Netz. Er im wesentlichen in folgende Teilschritte gegliedert:

- Falls der Zielknoten eines Paketes im Leaf-Set verzeichnet ist, sende es direkt dorthin
- Ermittle Länge des gemeinsamen Präfix von eigener NodeID und Nachrichten ID und suche denjenigen Knoten in der Routing Tabelle, dessen gemeinsamer Präfix am mindestens um eins länger ist. Sende Nachricht dorthin.
- Falls kein passenderer Knoten in der Routingtabelle gefunden wird (was in der Praxis sehr selten der Fall ist), sende die Nachricht an den numerisch nahesten der in der Vereinigung von Leaf-Set, Routingtabelle und Neighborhood-Set zu finden ist.

Um die Adresse eines Knotens zu finden dessen gemeinsames Präfix um eins länger ist als das eigene bedient man sich der Routing-Tabelle.

| Routing Tabelle | | | |
|---|---|---|---|
| -0-2212102 | **1** | -2-2301203 | -3-1203203 |
| **0** | 1-1-301233 | 1-2-230203 | 1-3-021022 |
| 10-0-31203 | 10-1-32102 | **2** | 10-3-23302 |
| 102-0-0230 | 102-1-1302 | 102-2-2302 | **3** |
| 1023-0-322 | 1023-1-000 | 1023-2-121 | **3** |
| 10233-0-01 | **1** | 10233-2-32 | |
| **0** | | 102331-2-0 | |
| | | **2** | |

**Abb. 3.1 Routing Tabelle des Knoten 10233102**

In der jeweils k-ten Zeile befinden sich diejenigen Knoten, deren NodeID mit der eigenen ein k Zeichen langes gemeinsames Präfix besitzten. In den Spalten der Routing-Tabelle befinden sich Adressen mit unterschiedlichen darauf folgenden Zeichen. Auf diese Weise ist es einfach den nächsten Knoten zu finden, es kann jedoch passieren, dass die entsprechende Zeile leer ist.

Normalerweise erfolgt jedoch kein Zugriff auf leere Zeilen, da immer zuerst im Leaf-Set nachgesehen wird. Beinhaltet dieses mehr als $2 * 2^b$ Einträge, so findet man Knoten mit einem kürzeren Präfix darin. Dies hat ferner die Auswirkung, dass die Nachricht i.A. direkt zum Zielknoten geleitet wird sobald dieser im Leaf-Set auftaucht. Dass die Routing-Tabelle also nach unten hin schwächer besetzt ist ist irrelevant, da bis dorthin nicht geroutet wird. Geht man davon aus, dass über den Raum der NodeIDs die Teilnehmer gleich verteilt sind, so ergeben sich durchschnittlich $\log_{2^b}$ (*Anzahl der Teilnehmer*) Hops.

### 3.1.2 Lokalität

Die Separation der ersten beiden Schritte der Routingprozedur machen bei flüchtiger Betrachtung wenig Sinn, könnte man die Knoten des Leaf-Set doch direkt in die Routing Tabelle aufnehmen. Hier kommt die Lokalität ins Spiel, die bereits angesprochene heuristische Nähe (geographisch nahe, Ping-Time, etc.). Anders als das Leaf-Set, in dem ausschließlich Wert auf die Nähe im Adressraum gelegt wird, ändert sich die Routing Tabelle mit steigender Kommunikation und dem Bekanntwerden neuer Knoten. Stellt sich heraus, dass ein neuer Knoten heuristisch näher ist als ein vergleichbarer bereits in der Routing Tabelle eingetragener (z.B. selbe Länge des gemeinsamen Präfix), so wird eine Ersetzung vorgenommen.

Stark vereinfachend besteht der Routing-Prozess somit aus zwei Phasen:

- Suche in der heuristischen Nähe den zum gesuchten Schlüssel nummerisch nahesten Knoten und sende Nachricht dorthin,
- denn die Wahrscheinlichkeit ist groß, dass dieser den Empfänger im Leaf-Set hat.

Durch diesen Mechanismus ist (wie man bei der Performanzmessung feststellt) gewährleistet, dass der via Routing zurückgelegte Weg realtiv kurz verglichen mit einer Punkt-zu-Punkt Verbindung ist.

### 3.1.3 Selbstorganisation

Ein Knoten, der an einer bestehenden Netzwerkstruktur teilnehmen möchte, muß an die dafür notwendigen Informationen (Routing-Tabelle und insb. Leaf-Set) gelangen. Hierfür kontaktiert er einen beliebigen Knoten des Verbundes. Sind die ersten k Stellen der NodeIDs identisch, so kann der neue Knoten auch die ersten k Zeilen der Routing-Tabelle übernehmen. Danach sendet der kontaktierte Knoten eine Nachricht an den zur neuen NodeID nahesten Knoten, die diesen über den Neuankömmling informiert. Dabei wandert die Nachricht auf einer Route, deren NodeIDs immer ähnlicher der des neuen Nodes wird. All diese Knoten senden wiederum Nachrichten an den Neuankömmling, der damit seine Routingtabelle weiter komplettiert. Am Ziel (dem Knoten vor dem letzten Hop) angekommen, sendet dieses sein Leaf-Set, welches vom neuen Knoten vorerst übernommen werden kann.

Der Aufbau des neuen Neighborhood-Set gestaltet sich dagegen schwieriger: Pastry setzt voraus, dass der initial kontaktierte Knoten topologisch nah ist und somit dessen Neighborhood-Set übernommen werden kann. Ist dies nicht der Fall, so ersetzt der neue Knoten mit steigender Kommunikation schlechte Einträge und beschleunigt dadurch allmählich sein Routing.

Der Ausfall eines Knoten wird nicht nur zum Problem, da er als Zielknoten fehlt, sondern auch da eventuell über ihn geroutet werden soll. Dies muß vom Netzwerk erkannt und korrigiert werden. Zunächst aber überprüfen topologisch nahe Knoten mit Hilfe ihres Neighborhood-Set zyklisch ob ihre Nachbarn noch funktionsfähig sind. Antwortet ein Knoten auf diese Nachfrage nicht mehr, so wird eine Nachricht mit dessen NodeID versendet. Diese kommt logischerweise bei einem Knoten an, der den ausgefallenen Teilnehmer wiederum in seinem Leaf-Set hat. Das Fehlen wird hier erkannt und der defekte Knoten entfernt. Auf diese Weise verschwindet der Knoten aus allen Leaf-Sets. Die Korrektur der Routing-Tabellen gestaltet sich schwieriger, da noch weitere Knoten über den ausgefallenen Node routen könnten. Hier wird der Fehler erst bei fehlgeschlagenem Routing bemerkt. Da aber immernoch über das Leaf-Set geroutet werden kann, ist wenigstens das ankommen der Nachricht gewährleistet - dann aber nicht mehr in garantiert logarithmischer Zeit.

### 3.1.4 Pastry API

Das Pastry Application Programming Interface bietet eine überschaubare Anzahl an Befehlen um die komplette Kommunikation innerhalb des Netzwerkes zu initialisieren und abzuwickeln:

- **nodeID = pastryInit(Credentials, Application)**
  Der Aufruf der Funktion *pastryInit* startet den lokalen Knoten und registriert ihn im Netzwerk. Er liefert die lokale Knoten-ID zurück und verlangt beim Aufruf sowohl eine applikationsabhängige Struktur als auch ein Application-Handle um Rückmeldungen weiterleiten zu können.
- **route(msg, key)**
  Die Funktion *route* veranlasst Pastry die übergebene Nachricht an den Knoten mit der ebenfalls übergebenen Knoden ID zu übermitteln.

Applikationen die Pastry verwenden müssen ihrerseits die folgenden Methoden exportieren:

- **deliver(msg, key)**
  Die Methode *deliver* wird von Pastry aufgerufen falls der lokale Knoten der nummerisch naheste zu *key* ist. Übergeben wird dann die Nachricht *msg*.
- **forward(msg, key, nextId)**
  Wird eine Nachricht empfangen, die eigentlich zur Weiterleitung gedacht ist, trifft sie per *forward* ein. Sie kann modifiziert oder durch setzen von *msg* auf NULL terminiert werden.
- **newLeafs(leafSet)**
  Die Methode *newLeafs* wird von Pastry immer dann aufgerufen wenn sich Änderungen im lokalen Leaf-Set ergeben.

## 3.2 Performance

Eine frühe Pastry-Implementation in Java wurde mit Hilfe einer Netzwerk-Emulationsumgebung auf einem Compaq AlphaServer ES40 (4 x 500 MHz 21264 Alpha CPU) getestet. Den N Nodes wurden hierfür jeweils ein zufällig bestimmter Punkt auf einer 1000 x 1000 Felder großen Ebene zugeordnet um die Entfernungsinformation zu generieren - dies beschreibt das reale Internet jedoch sehr unzureichend, da sich dort i.d.R Cluster von Knoten bilden.

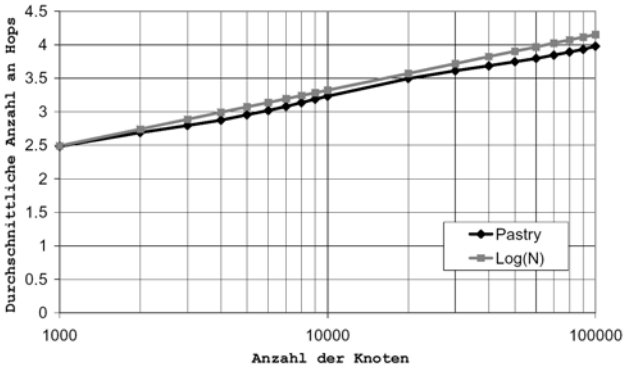


**Abb. 3.2 Routingperformance**

Gezeigt werden konnte, dass die reelle Anzahl an Routing-Hops offensichtlich sehr gut mit dem $\log_{2^b}$ (*Anzahl der Teilnehmer*) übereinsimmt (in der Regel sogar eine Spur besser ist). Das Netzwerk skaliert folglich sehr gut und erfüllt die Erwartungen.

Außerdem wurde die relative Entfernung gemessen, die bedingt durch das Routing zuviel zurückgelegt wird.

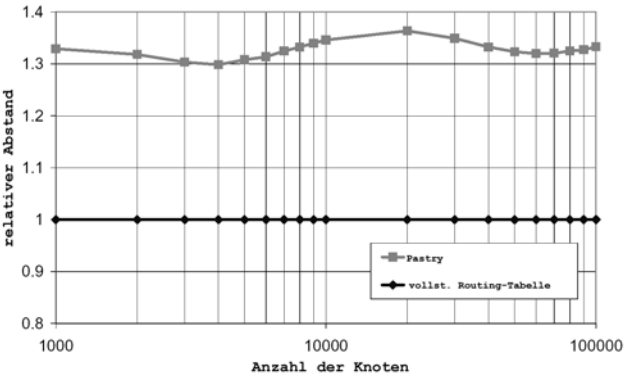


**Abb. 3.3 relative Entfernung**

Aufgetragen wurde die relative Entfernung über die Anzahl der Pastry-Knoten. Unter relativer Entfernung versteht man in diesem Zusammenhang die Summe der Routing-Teilstücke im Verhältnis zum direkten Abstand der Start und Endknoten. Gezeigt werden konnte, dass die verhältnismässig kleinen Routing-Tabellen (im Vergleich zum großen Pastry Netz) keine signifikant längeren Wege produziert.

## 3.3 Implementationen

Derzeit existiert sowohl eine freie Pastry Implementation der Rice University als auch ein Packet das von Microsoft Research zur Verfügung gestellt wird. Beide Versionen befinden sich in einem bestenfalls als fragmentartig zu bezeichnendem Stadium und werden von unterschiedlichen Institutionen gepflegt.

### 3.3.1 VisPastry/SimPastry in Microsoft .NET

VisPastry steht in der Version 1.1 als Binärdistribution zum Download bereit und beinhaltet neben einem Kommandozeilentool zur Simulation eines Netzwerkes auch das grafische VisPastry. Beide Komponenten wurde in Microsofts C# entwickelt und benötigen das .NET Framework

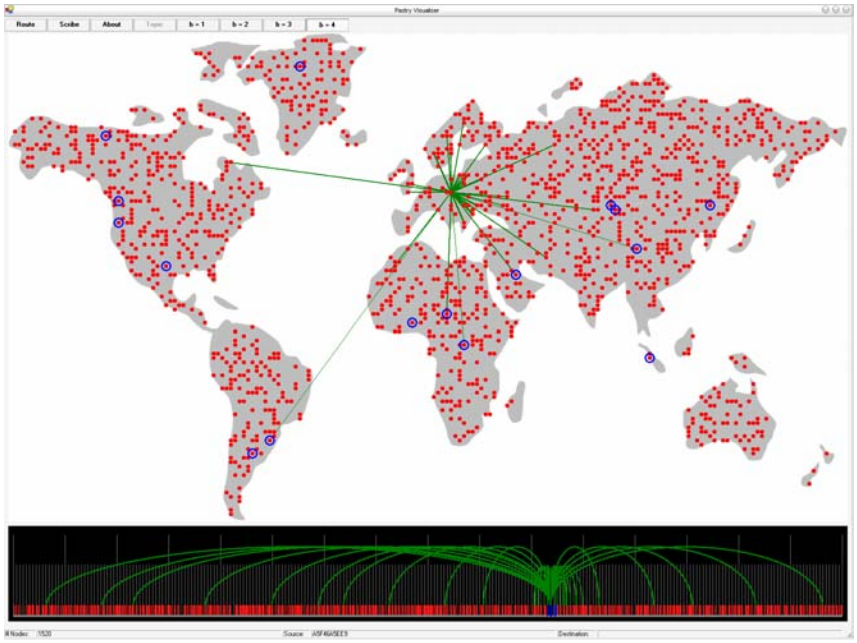

**Abb. 3.4 VisPastry aus SimPastry 1.1 auf .NET**

SDK zur Ausführung. VisPastry ermöglicht es Parameter der Binärbaumstruktur zu variieren und verdeutlicht sowohl das Routing als auch die Bedeutung des Leaf-Set. Die Gleichverteilung der Knoten auf dem 128 Bit Adressraum wird genauso gezeigt wie die Wirkungsweise von Scribe. Für die Zukunft ist geplant die vorhandenen Bibliotheken besser zu dokumentieren und vor allem Beispielapplikationen bereitzustellen um Pastry für Windowsanwendungen verfügbar zu machen.

### 3.3.2 FreePastry

FreePastry ist eine OpenSource Implementation von Pastry und wird in Java entwickelt. Die derzeitige Versionsnummer 0.1 beta deutet schon auf den experimentellen Charakter des Paketes hin und soll im wesentlichen Interessierte dazu animieren sich das Protokoll näher anzusehen um seine Vorzüge zu erkennen und bei der Entwicklung und Forschung mitzuwirken. Für die Zukunft soll vor allem an der Ausfallsicherheit und Robustheit von FreePastry gearbeitet werden um es später in einer Vielzahl von OpenSource Anwendungen wiederzufinden.

# 4 Anwendungen

Aufbauend auf Pastry wurde in jüngster Zukunft eine ganze Reihe von Applikationen entwickelt, darunter ein verteilter Web-Cache (Squirrel), ein verteiltes Dateisystem (Past) und ein Nachrichten-Kommunikationsystem genannt Scribe.

## 4.1 Past

Past [5] ist ein auf Pastry aufbauendes verteiltes Speichersystem. Bei der Entwicklung wurde das Ziel verfolgt Dateien so zu duplizieren, dass einzelne Knoten ausfallen und die gewünschten Daten dennoch wieder gefunden werden können.

Die Datei wird dabei mit einem erzeugten Schlüssel versehen und ins Past-Netzwerk eingespeist. Dieses leitet die Nachricht an denjenigen Knoten weiter, dessen NodeID am nahesten, der wiederum das File an k viele benachbarte Knoten (IDs) weitergibt. Da die NodeIDs im Pastry Netz gleichmäßig verteilt sind, verteilen sich auch die Dateien gleichmässig im Netzwerk - lokale Ausfälle sind somit als weniger dramatisch anzusehen.

Past besitzt über diese Grundfunktionalität hinaus eine Quota-Verwaltung und diverse Verschlüsselungsfunktionen, die allerdings allesamt über Smartcards mit Kryptofunktionalität realisiert sind. Die Verwaltung der Quotas geschieht zentral in Form einer Instanz, die die Smartcard ausgibt und Schlüssel für das Netz verteilt. Ein Nutzer ist deshalb nicht in der Lage mehr Speicherplatz im Netzwerk zu belegen als er selbst zur Verfügung stellt. Will er ein File einspeisen, so wird es in der Smartcard signiert und die Filegröße mit seinen Quotas verrechnet.

Leider ist deshalb das System auch weniger performant: Smartcards sind für das durchschleusen großer Datenmengen nicht konzipiert und auch eine zentrale Quota-Verwaltung macht bei einem dezentralen Peer-to-Peer System wenig Sinn.

## 4.2 Scribe

Scribe [6] ist ein verteiltes Nachrichtenpublikationssystem, das ebenfalls auf Pastry aufbaut. Die Schlüssel werden hier so genannten Nachrichtenkategorien (engl. *topics*) zugeordnet und zum Knoten mit der nahesten ID als Rendezvous-Punkt für Sender und Empfänger geroutet. Jeder Knoten des Scribe Netzwerkes darf beliebig viele Nachrichtenkategorien erstellen. Andere Knoten können dieses Topic abonnieren und (sofern sie dazu berechtigt sind) Nachrichten in diese Kategorie posten. Scribe sorgt anschließend dafür, dass die Nachricht entlang eines Multicast-Trees mit dem Rendezvous-Punkt als Wurzel an alle Abonnenten verteilt wird.

Scribe kann also als Publikations-/Abonnementsystem verstanden werden, das Pastry im Wesentlichen nutzt um das Verteilen der Nachrichten auf die Abonnenten zu übernehmen und darüber hinaus dessen fehlertolerante Eigenschaften nutzt.

# 5 Literaturverzeichnis


[1] OpenNap: Open Source Napster Server, http://opennap.sourceforge.net/

[2] The Gnutella Protocol Specification v0.4, http://www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf

[3] Jordan Ritter, Why Gnutella can't Scale. No, really., http://www.darkridge.com/~jpr5/doc/gnutella.html

[4] Antony Rowstron, Peter Druschel. Pastry: Scalabel, decentralized object location and routing for large-scale peer-to-peer systems. - http://www.research.microsoft.com/~antr/PAST/pastry.pdf

[5] Peter Druschel, Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. - http://www.research.microsoft.com/~antr/PAST/hotos.pdf

[6] Miquel Castro, Peter Druschel, Anne-Marie Kerrmarec, Antony Rowstron. SCRIBE: A large-scale and decentralized publish-subscribe infrastructure.- http://www.research.microsoft.com/~antr/PAST/jsac.pdf