Studienarbeit P2P-Taskbag

Philippe Morier & Martin Weber

Betreuer: Prof. Dr. Josef Joller

HSR Hochschule für Technik Rapperswil

Herbstsemester 2010

Erklärung

Wir, Philippe Morier und Martin Weber erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.

Rapperswil, 22. Dezember 2010

Philippe Morier

Martin Weber

Inhalt

1 Einführung

Die folgende Arbeit behandelt ein Konzept, welches beschreibt wie ein Peer-To-Peer Netz mit einem Auftrag mittels des REST-Prinzips beauftragt werden kann und wie die Ergebnisse des Auftrags wieder abgefragt und somit ausgewertet werden können. Interessant ist dabei wie REST auf das Peer-To-Peer Netz angewendet wird. Die Verwendung dieser beiden Technologien stellt eine beinahe unbekannte Kombination dar. Als weiterer Aspekt wurde darauf geachtet, dass auch das Verhalten bei Auftritt von Fehlern untersucht wurde. Also zum Beispiel wie das Netzwerk bei Ausfall von Nodes reagiert.

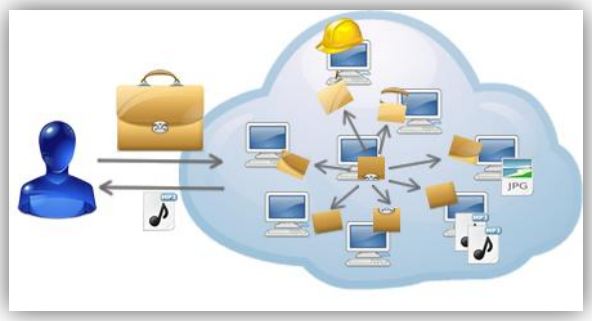

Abbildung 1-1: Übersicht

In den nachfolgenden Kapiteln wird zuerst unsere Vision, die Ziele, Vorgaben und Einschränkungen, welche für die Studienarbeit gesetzt wurden, behandelt. Danach werden die dem P2P-Taskbag zugrunde liegende Technologien etwas genauer erläutert. Da der P2P-Taskbag auf diesen Technologien und Konzepte aufbaut, ist es wichtig, dass diese verstanden werden. In den darauffolgenden Kapiteln wird die effektive Implementation des P2P-Taskbags beschrieben. Diese beinhalten die Beschreibung der Architektur. Das heisst es werden der Aufbau des P2P-Taskbags sowie mögliche Fehler und deren Lösungen aufgezeigt. Des Weiteren wird erwähnt wie die Funktionalitäten des P2P-Taskbag getestet werden können und was für Erweiterungen, Ideen und Verbesserungen für den P2P-Taskbag denkbar wären. Nach der Entwicklung wurde die Performance und Skalierbarkeit getestet. Gegen Ende der Dokumentation wird auf Probleme, Entscheidungen sowie Persönliche Erfahrungen eingegangen. Ebenfalls gibt es ein Kapitel welches beschreibt, wie ein Netz unter dem Einsatz des P2P-Taskbags aufgebaut werden kann und sollte als eine Art Bedienungsanleitung betrachtet werden. In den letzten Kapiteln sind ein Glossar sowie diverse Referenzen auf interne und externe Quellen aufgeführt.

2 Vision

Grundsätzlich soll ein Behälter in Form eines P2P-Netzwerkes aufgebaut werden, der verschiedene Aufträge entgegennehmen kann und diese bearbeitet. Der Auftraggeber soll zu jeder Zeit den Status seines Auftrages und dessen Ergebnis abfragen können. Durch die Aufteilung des Auftrags in mehrere Jobs können diese nebenläufig abgearbeitet werden.

Der Aufbau resp. die Kommunikation im P2P-Netzwerke soll nach der REST-Architektur umgesetzt werden. REST vereinfacht durch das Verwenden von HTTP die Kommunikation und Serviceintegration auf der Applikationsebene.

Die REST-Nachrichten werden mit der Extensible Markup Language kurz XML dargestellt. XML ist eine standardisierte Darstellungssprache und wird daher von den verschiedensten Plattformen vollständig unterstützt.

Oft werden in P2P-Systeme immer noch zentrale Server eingesetzt. Diese stellen einen „Single-Point-Of-Failure“ dar. In unserem System sollte der zentrale Server durch eine DHT ersetzt werden.

2.1 Wirtschaftliche Motivation

Als eine realistische und wirtschaftliche Anwendung wäre vorstellbar, dass man unser Netz als eine günstige Alternative zu einem eigenen Rechenzentrum für Firmen anbieten würde. Eine Firma kann sich eine bestimmte Rechenleistung (Anzahl Nodes) kaufen resp. mieten. Diese kann sie für Ihre eigenen Aufträge nutzen. Auf der anderen Seite sollen Privatpersonen ihren Rechner resp. ihre Rechenleistung gegen Geld zur Verfügung stellen können. Dieser Ansatz erzielt auf beiden Seiten Motivation. Auf der einen Seite erhalten Firmen günstige Rechenzentren und auf der anderen Seite können Privatpersonen ihre ungenutzte Rechnerleistung verkaufen.

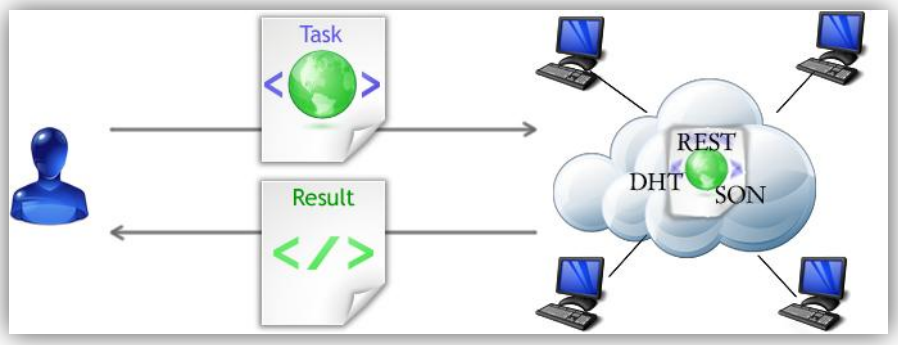

Abbildung 2-1: Vision

3 Ziele

Diese Arbeit verfolgt folgende Ziele, die aus dem Auftragsdokument übernommen wurden.

3.1 Erstellen eines P2P-Netzwerkes

Es soll ein P2P-Netzwerk erstellt werden, welches auf Befehl eines Nodes generische Aufgaben ausführt und je nach Aufgabe die Ergebnisse dem Peersystem zur Verfügung stellt. Die Ergebnisse sollten wenn möglich mithilfe einer verteilten Hashtabelle gefunden und passend kombiniert werden.

3.2 REST

Die ganze Kommunikation soll dem REST-Prinzip folgen. Mithilfe von REST soll ein eigenes Protokoll definiert werden. Dieses soll erweiterbar sein, so dass spätere Zusatzanforderungen in das Protokoll integriert werden können.

3.3 Wahlfunktion

Das Netzwerk sollte eine Wahlfunktion zur Verfügung stellen. Mit Hilfe dieser es möglich sein sollte, einen Node für eine bestimmte Aufgabe zu wählen. Vorstellbar wäre, dass ein Node als eine Art Garbage-Collector fungieren würde und so alle ungültigen Ressourcen in der Hashtabelle löschen würde. Die Wahlfunktion könnte auch für das Bestimmen eines Eintrittsnode verwendet werden.

3.4 Semantic Overlay Network

Je nach Job stellt ein Node Informationen in Form einer Ergebnisdatei dem Netzwerk zur Verfügung. Dieses soll nun eine effiziente Suche nach dessen Informationen bieten. Dazu sollen einerseits die Nodes anhand Ihrer Systemressourcen in unterschiedlichen Gruppen eingeteilt werden. Jede Gruppe soll nur eine bestimmte Art von Jobs erledigen. Andererseits sollen die resultierenden Ergebnisdateien mit Metadaten versehen werden. So soll es zum Beispiel möglich sein nach einem gewissen Musikinterpret, Genre oder Softwaresparte zu suchen.

3.5 Statusabfrage

Neben applikatorischen Funktionen (suchen, beauftragen) müssen weitere systemnahe Funktionen beispielsweise für die Administration des Netzwerkes bzw. Abfrage des Status der Peers und ähnliches definiert und mindestens exemplarisch implementiert werden. Zu den systemnahen Funktionen gehört auch die bereits erwähnte Wahlfunktion.

4 Vorgaben & Einschränkungen

Unsere Semesterarbeit unterliegt gemäss dem Auftragsdokument verschiedenen Vorgaben und Einschränkungen. Diese Einschränkungen wurden schon vorzeitig definiert. Auf die Untersuchung von alternativen Möglichkeiten wurde deswegen verzichtet. Diese Vorgaben und Einschränkungen sollen im Folgenden kurz erläutert werden.

4.1 P2P

Das zu erstellende Netz soll auf dem P2P-Prinzip basieren. Dies bedingt, dass alle Nodes gleichwertig sind. Es dürfen also keine höher privilegierten Nodes im Netz existieren. Dies ist die Grundvoraussetzung, um Aufträge verteilt abarbeiten zu können.

4.2 REST

Die Architektur und Kommunikation zwischen den einzelnen Nodes soll nach der REST-Architektur implementiert werden. Die Hauptaufgabe liegt also darin Ressourcen zu identifizieren und zu adressieren.

4.3 HTTP

Als Kommunikationsprotokoll soll HTTP verwendet werden. HTTP bringt bereits eine Hand voll Methoden (GET, POST, PUT, DELETE, etc.) und eine grosse Anzahl von Statuscodes (1xx, 2xx, 3xx, etc.) mit sich. Diese sollen grösstenteils verwendet und allenfalls implementiert werden.

4.4 DHT

Um einen „Single-Point-Of-Failure“ im Netz zu verhindern, sollte kein zentraler Server sondern eine Distributed Hash Table kurz DHT verwendet werden. Diese soll hauptsächlich für das Wiederauffinden der Ressourcen im Netz verantwortlich sein.

4.5 XML

Die REST Ressourcen werden mit XML (Extensible Markup Language) repräsentiert. D.h. das Projekt soll dem XML-Standard gerecht werden. Durch die Verwendung von XML wird die Datendarstellung plattformneutral.

4.6 Microsoft .Net Framework 4.0

Für die Umsetzung wird das von Microsoft stammende .Net Framework in der Version 4.0 verwendet. Dies bedeutet, dass wir von dessen Funktionsumfang abhängig sind. Für die Entwicklung serviceorientierter Architektur wird das WCF (Windows Communication Foundation) verwendet.

5 Technologien

Dieses Kapitel beschreibt grob die verwendeten Technologien. Dadurch wird sichergestellt, dass die fundamentalen Technologien bekannt sind. Dieses Verständnis wird später in der Dokumentation vorausgesetzt.

5.1 HTTP

REST verwendet diverse Funktionen des HTTP (Hypertext Transfer Protocol). HTTP dient dazu Daten über ein Netzwerk zu übertragen. Dabei ist eine Verbindung zwischen dem Server und Client zustandslos. Das Protokoll befindet sich auf der Anwendungsschicht. Die Anwendungsschicht entspricht den Schichten 5-7 im ISO/OSI-Schichtenmodell. Die Daten werden als sogenannte Nachrichten zwischen den Teilnehmer hin und her gesendet. Dabei wird die Nachricht in zwei Bereiche unterteilt. Diese sind der Nachrichtenkopf (Header) und der Nachrichtenkörper (Body). Der Header enthält Informationen über den Body und der Body selbst stellt den eigentlichen Inhalt der Nachricht dar. Eine Anfrage vom Client an den Server wird Request genannt. Die darauf folgende Antwort wird Response genannt. [1]

5.1.1 Request Methoden

Der Client hat verschiedene Optionen wie er einen Request an den Server stellen kann. Nachfolgend werden nur die Methoden beschrieben, die für REST relevant sind.

5.1.1.1 GET

Einen GET-Request wird für das Abfragen einer Ressource vom Server unter Angabe eines URIs verwendet. Dabei ist wichtig, dass ein GET-Request idempotent ist d.h. ein GET-Request ist frei von Nebeneffekten und kann daher bedenkenlos mehrmals gestellt werden, ohne eine Veränderung der Ressource mit sich zu ziehen. GET-Requests können und werden oft in einem Cache zwischengespeichert. Im folgenden Beispiel ist ein GET-Request dargestellt:

```
GET   /task/412 HTTP/1.1
Host: 10.10.10.41
```

Tabelle 5-1: GET-Request

5.1.1.2 POST

Die POST-Methode ermöglicht es theoretisch eine unbegrenzte Menge an Daten an eine Ressource auf dem Server zu senden. Dadurch können bestehende Ressourcen verändert oder neue untergeordnete Ressourcen erstellt werden. Da ein POST-Request Logik auf dem Server aufruft, kann dadurch ein Zustandswechsel auf dem Server ausgelöst werden. Darum ist ein POST-Request nicht frei von Seiteneffekte und somit auch nicht idempotent. Beispiel eines POST-Requests:

```
POST   /task/412 HTTP/1.1
Host:  10.10.10.41

job=542
```

Tabelle 5-2: POST-Request

5.1.1.3 PUT

Eine neue Ressource kann mit der Methode PUT auf dem Server erstellt werden. Dabei muss der entsprechende URI bekannt sein und die zu erstellende Ressource darf noch keinen weiteren Ressourcen zugeordnet sein.

```
PUT    /task
Host:  10.10.10.41
       <task>
              <name>Taskbeispiel</name>
       </task>
```

Tabelle 5-3: PUT-Request

5.1.1.4 DELETE

Mit der DELETE-Methode lässt sich eine bestehende Ressource löschen. Dabei muss natürlich wieder der URI der Ressource bekannt sein.

```
DELETE /task/412
Host:  10.10.10.41
```

Tabelle 5-4: DELETE-Request

5.1.2 Statuscodes

Der Server liefert als Antwort auf einen Request jeweils einen genau definierten Statuscode. So ist der Serverstatus nach jedem Request ersichtlich. Die einzelnen Statuscodes werden in insgesamt fünf Gruppen eingeteilt. [2]

5.1.2.1 1xx – Informationen

Alle Statuscodes, die mit einer Eins beginnen, teilen dem Client mit, dass der Server noch mit der Verarbeitung des Requests beschäftigt ist. Dies ist nur eine temporäre Antwort.

5.1.2.2 2xx – Erfolgreiche Operation

Mit den 200er-Nachrichten wird dem Client mitgeteilt, dass seine Anfrage erfolgreich behandelt wurde. So werden 200er-Nachrichten immer dann gesendet, wenn ein Request erfolgreich übertragen werden konnte.

5.1.2.3 3xx – Umleitung

Empfängt der Client eine 300er-Nachricht so wird ihm vom Server mitgeteilt, dass er einen weiteren Schritt respektive Anfrage an ev. einen anderen Server senden muss. Zum Beispiel kann es vorkommen, dass eine Ressource unter dem angegebenen URI nicht mehr vorhanden ist, sondern sich neu unter einem anderen URI befindet.

5.1.2.4 4xx – Client-Fehler

Sendet der Server einen Statuscode der mit vier beginnt so teilt er dem Client mit, dass dessen Anfrage fehlerhaft war. Solche Nachrichten werden zum Beispiel bei fehlerhafter Authentifizierung oder bei einer fehlerhaften Anfrage zurück gesendet.

5.1.2.5 5xx – Server-Fehler

Stellt der Server fest, dass ein Problem bei ihm aufgetreten ist, sendet er dem Client eine 500er-Nachricht. Solche Probleme können das Nichtvorhandensein eines Dienstes oder einer Funktion auf Seite des Servers sein.

5.2 REST

Wie bereits erwähnt baut REST (Representational State Transfer) auf den Funktionalitäten von HTTP auf. Darum besitzt es auch sehr ähnliche Eigenschaften. REST verwendet die von HTTP beschriebenen vier Methoden, auch Verben genannt, GET, PUT, POST und DELETE. Diese wurden bereits beschrieben. REST beschreibt einen architektonischen Style für verteilte Hypermedia Systeme. Dabei liegt die Hauptaufgabe darin zu beschreiben wie Ressourcen verschoben und neu angelegt werden können. Die grösste REST Anwendung ist wohl das World Wide Web. [3] [4] [5] [6]

5.2.1 Eigenschaften

Folgende sechs Eigenschaften charakterisieren REST.

5.2.1.1 Server/Client

Bei REST sind die Clients und Server voneinander getrennt und kommunizieren über ein gemeinsames Interface. Somit wurde in erster Linie das User-Interface von der Speicherung der Daten getrennt. So können beliebige Clients auf unterschiedlichen Plattformen unabhängig entwickelt werden. Durch die Trennung wird die Skalierbarkeit und Erweiterbarkeit des Systems unterstützt.

5.2.1.2 Zustandslose Kommunikation

REST setzt voraus, dass die ganze Kommunikation zustandslos ist. Deshalb müssen die Nachrichten selbstbeschreibend und in sich geschlossen sein. Eine Nachricht vom Client zum Server muss alle nötigen Informationen enthalten damit der Request vollständig interpretiert werden kann. Der Server kennt und speichert keinen Client-Zustand. Es werden also keine Sessions auf dem Server verwaltet. Der Client speichert als Einziger Session-Zustände. Dieser bestimmt auch die Reihenfolge der Methodenaufrufe auf dem Server. Dieses Prinzip resp. Style wird Client-Stateless-Server (CSS) genannt und ist im folgendem Bild dargestellt.

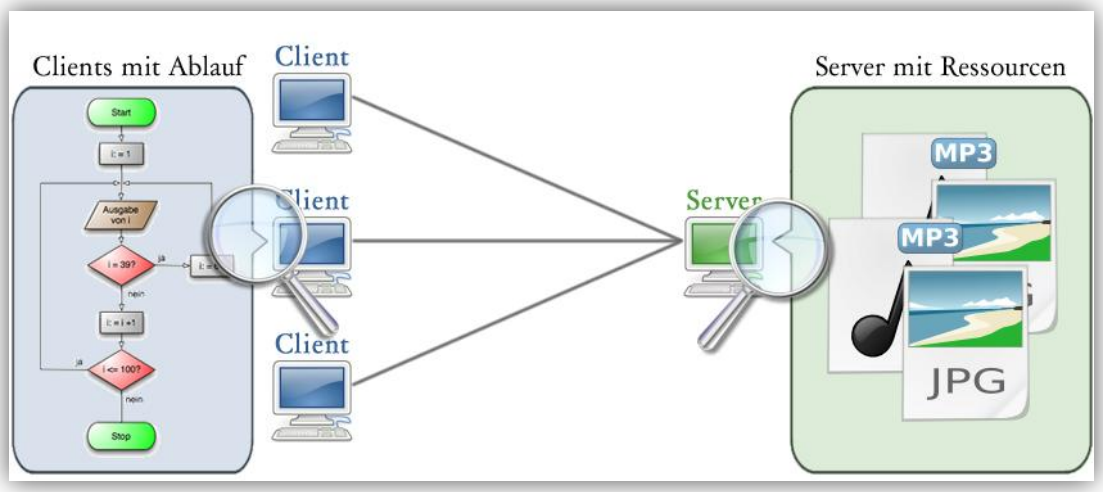

Abbildung 5-1: Client-Stateless-Server

5.2.1.3 Cache

Eine Response kann explizit als cacheable markiert werden. Empfängt der Client eine cacheable Response, ist es ihm erlaubt die Ressource zwischen zu speichern, um diese bei einem äquivalentem Request wieder zu verwenden. Durch das Hinzufügen eines Caches wird die Anzahl Interaktionen zwischen Server und Client vermindert. Dies steigert die Skalierbarkeit und die Effizient und vermindert die durchschnittliche Latenz der Interaktionen.

5.2.1.4 Einheitliches Interface

Einen sehr wichtigen Aspekt der REST Architektur ist das einheitliche Interface zwischen den Komponenten. Das einheitliche Interface vereinfacht die Architektur und trägt zur Effizienzsteigerung bei. Zudem ermöglicht es das unabhängige Entwickeln des Servers bzw. des Clients. Das Interface definiert vier Punkte:

- Identifikation der Ressource
- Veränderung der Ressource
- Selbstbeschreibende Nachrichten
- Verwendung von Hypermedia

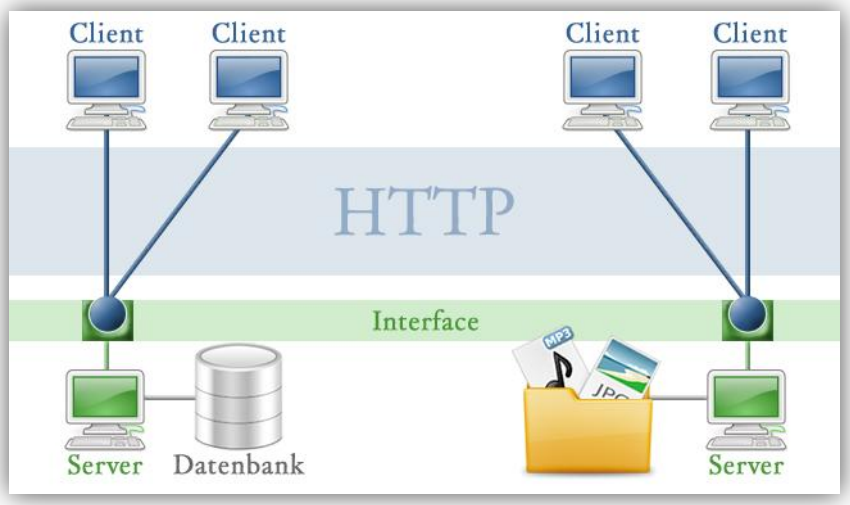

Abbildung 5-2: Einheitliches Interface

5.2.1.5 Layer- und Tier-Architektur

In der Layer-Architektur sind die einzelnen Schichten hierarchisch in Layer organisiert. Jede Schicht stellt der übergeordneten Schicht Dienste zur Verfügung und nutzt selber Dienste der darunterliegenden Schicht. Eine hierarchische Struktur vermindert die Kopplung. Jede Schicht kennt nur die Schichten mit denen sie auch effektiv verbunden ist.

Bei einer Tier-Architektur weiss ein Client nicht ob er wirklich mit dem Endserver verbunden ist oder nicht. Er sieht nur seinen mit sich direkt verbundenen Server. Dieser kann weitere Dienste von anderen Servern nutzen, ohne dass der Client davon etwas weiss resp. bemerkt.

Der Hauptnachteil dieser Architektur liegt darin, dass jede zusätzlicher Tier Overhead generiert und zur Vergrösserung der Latenz bei dem Bearbeiten von Daten beiträgt.

5.2.1.6 Code on Demand

Die Funktionalität eines Client kann durch herunterladen und ausführen von Code erweitert werden. Die Erweiterungen werden in der Form von Applets oder Scripts zur Verfügung gestellt.

5.3 Peer-To-Peer

Peer-To-Peer ist ein grundlegendes Prinzip einer modernen Netzwerkstruktur und wird für den Einsatz von verteilten Systemen verwendet. Peer kommt aus dem Englischen und steht für „Gleichgestellter“. Peer-To-Peer oder kurz P2P bedeutet demnach „von Gleichgestelltem zu Gleichgestelltem“. Im Gegensatz zur klassischen Client-/Serverarchitektur sind alle am Netz angeschlossenen Geräte gleichwertig. Ein Endgerät kann zugleich in die Rolle eines Servers wie auch in die Rolle eines Clients schlüpfen. Der Aufbau des Netzes ist nicht geregelt und erlaubt somit die Kommunikation mit allen Endgeräten. Das globale Verhalten eines P2P-Systems wird durch das

lokale Verhalten aller Nodes bestimmt. P2P-Systeme erlauben im Normalfall enorme Parallelisierung und sind deswegen für viele Firmen von grossem Interesse. [7]

5.3.1 Eigenschaften

5.3.1.1 Selbstorganisation

Ziel ist es, dass keine manuelle Konfiguration für die Verwendung eines P2P-Netzes erforderlich ist. Die Nodes sollten eigenständig ein Netz aufbauen und aktuell halten.

5.3.1.2 Dezentralisierung

Dadurch, dass jedes Endgerät gleichwertig ist und die Daten des Netzes nicht auf einem zentralen Server gespeichert werden, wird das ganze System dezentralisiert. Dies bringt Vorteile wie beispielsweise die Nicht-existenz eines „Single-Point-Of-Failure“ sowie die Vermeidung von stark erhöhtem Datenverkehr zu gewissen Knoten (Flaschenhals) wie es z.B. bei Servern oft der Fall ist. Die Eigenschaft der Dezentralisierung ist bei der Verteilung von Daten von besonderer Bedeutung. Sie ist oftmals einer der Hauptgründe dafür, dass Filesharing unter der Verwendung von P2P-Systemen betrieben wird.

5.3.1.3 Skalierbarkeit

Ein wesentlicher Punkt von P2P-Systemen ist die Skalierbarkeit. Das System muss sowohl mit einer kleinen wie auch grossen Anzahl von Nodes fehlerfrei funktionieren. Wenn möglich sollten zusätzliche Nodes zur Verbesserung des ganzen Netzes beitragen z.B. durch eine Lastenverteilung. Der Ausfall einzelner Nodes ist somit beinahe bedeutungslos.

5.3.1.4 Fehlertoleranz

Das Netz muss trotz Ausfällen von Nodes oder grösseren Teilen des Netzes einwandfrei funktionieren. Ausfälle dürfen auf keinen Fall das ganze System ausser Betrieb setzen. Die Fehlertoleranz ist ein wichtiger Aspekt in einem P2P-Netz. Nur durch eine gut implementierte Fehlertoleranz kann ein effektives P2P-Netz garantiert werden. Um die Ausfallsicherheit zu gewährleisten sind oftmals komplexe Replikationsalgorithmen erforderlich.

5.4 DHT

5.4.1 Beschreibung

Eine Distributed Hash Table oder kurz DHT ist eine, in einem Peer-To-Peer-Netz verteilte, selbstorganisierende Datenstruktur. Sie legt die grundlegende Kommunikation der einzelnen Peers fest. Die Datenstruktur ist auf dem Prinzip einer normalen Hash Table aufgebaut. Sie erlaubt also das Speichern sowie das Abfragen von Key-Value Paires. Durch den Einsatz einer Hashfunktion wird die Speicherung dieser Key-Value Paires auf einen linearen Wertebereich eingeschränkt. Jeder

Node im P2P-Netz ist für die Verwaltung eines Teilbereiches dieses sogenannten Schlüsselraumes zuständig. Somit ist gewährleistet, dass jedes Key-Value Pair eindeutig einem Node im Netz zugeteilt wird. Bei der Suche nach einem Schlüssel werden O(log n) Hops benötigt, bis der entsprechende Node gefunden wird. [8] [9]

5.4.2 Eigenschaften

Das Konzept der DHT muss viele Eigenschaften, welche für ein P2P-Netz von entscheidender Bedeutung sind, ebenfalls unterstützen. Um ein effizientes P2P-System zu schaffen, muss die im P2P-System verwendete DHT folgende Eigenschaften unterstützen.

5.4.2.1 Selbstorganisation

Eine DHT sollte grundsätzlich so implementiert werden, dass keine manuelle Konfiguration nötig ist. Neue Nodes werden einfach in die DHT integriert und austretende Nodes von ihr entfernt. Die Verteilung des Schlüsselraumes sowie die Erstellung von Redundanz muss also von der DHT automatisch verwaltet werden.

5.4.2.2 Skalierbarkeit

Das System muss sowohl mit vielen, wie auch mit wenigen Nodes jeweils gut und effizient arbeiten. Wenn möglich sollten zusätzliche Nodes zur Verbesserung des Netzes beitragen, indem sie beispielsweise die Grösse des zu verwaltenden Schlüsselraums pro Node reduzieren oder eine bessere Lastverteilung ermöglichen.

5.4.2.3 Fehlertoleranz

Die DHT muss mit dem Ausfall von Nodes oder dem Verlust von Nachrichten umgehen können. Der Ausfall von Nodes muss erkannt und behandelt werden. Zusätzlich muss die Verfügbarkeit der Daten eines ausgefallenen Nodes weiterhin gewährleistet werden. Hierfür werden oftmals komplexe Algorithmen zur Generierung von Redundanz eingesetzt. Im Vergleich zu einem zentralen Server liegt bei einer DHT die Schwierigkeit besonders bei der Verfügbarkeit. Bei einem zentralen Server kann prinzipiell davon ausgegangen werden, dass dieser läuft und verfügbar ist. Bei einer DHT bilden allerdings alle Nodes zusammen die DHT. Dies bedeutet, dass auch diejenigen Nodes, welche nur eine sehr kurze Uptime haben, daran beteiligt sind. Um die DHT also aufrecht zu erhalten ist ein hoher Kommunikationsaufwand zu betreiben. Die Fehlertoleranz ist eine der wichtigsten, aber schwierigsten zu implementierenden Eigenschaften einer DHT. Durch eine gut implementierte Fehlertoleranz kann die Kommunikation zwischen Nodes enorm reduziert werden.

5.4.2.4 Lastenverteilung

Der Einsatz einer Hashfunktion auf allen Datenobjekten führt zu einer zufälligen Verteilung der Datenobjekte im gegebenen Schlüsselraum. Durch diese zufällige Verteilung entsteht eine gleichmässige Verteilung der Objekte auf die Nodes und somit eine Lastverteilung im P2P-Netz.

5.4.2.5 Dezentralisierung

Das System ist selbstorganisierend und die Nodes sind alle gleichwertig. Diese Gleichwertigkeit wird durch die gleichmässige Verteilung der Daten durch die Hashfunktion gewährleistet. Diese Eigenschaften führen zur Dezentralisierung des Systems. Ein Server wird weder für Initialisierung oder Aufrechterhaltung der DHT noch für spezielle Verwaltung von Daten benötig.

5.4.3 Chord

Chord ist eine Open-Source Implementation einer DHT und erfüllt alle Bedingungen einer DHT. Sie ermöglicht einfache resp. effiziente Suchabfragen und kann sich an Veränderung der Netzstruktur, wie z.B. dem Ausfallen oder Hinzukommen eines Nodes effizient anpassen. [10] [11] [12]

5.4.3.1 Aufbau

Jedem Node wird durch die Verwendung einer Hashfunktion wie zum Beispiel SHA-1 eine eindeutige Identifikationsnummer zugewiesen. Die Nodes werden anhand dieser Nummer sortiert und in einem logischen Ring angeordnet. Jeder Node kennt sowohl seinen Vorgänger, wie auch seinen Nachfolger. Die Stabilität und Effizienz des Systems wird zusätzlich erhöht indem jeder Node weitere, in logarithmischem Abstand entfernte, Nodes kennt. Die ebenfalls gehashten Datenobjekte werden jeweils dem nächst höheren Node des Schlüsselraums zugeordnet. Chord bietet nun eine einzige Funktion und zwar das Finden eines Nodes anhand eines gegebenen Schlüssels. Die Speicherung von Datenobjekten wird nicht explizit unterstützt.

5.4.3.2 Die Suche nach einem Schlüssel

Die einfachste Suche (Lookup) nach einem Schlüssel im Chord-Ring ist die lineare Suche. Wenn jeder Node nur seinen direkten Nachfolger kennt, kann eine lineare Suche realisiert werden. Diese Suche skaliert in O(n) wobei n die Anzahl Teilnehmer ist.

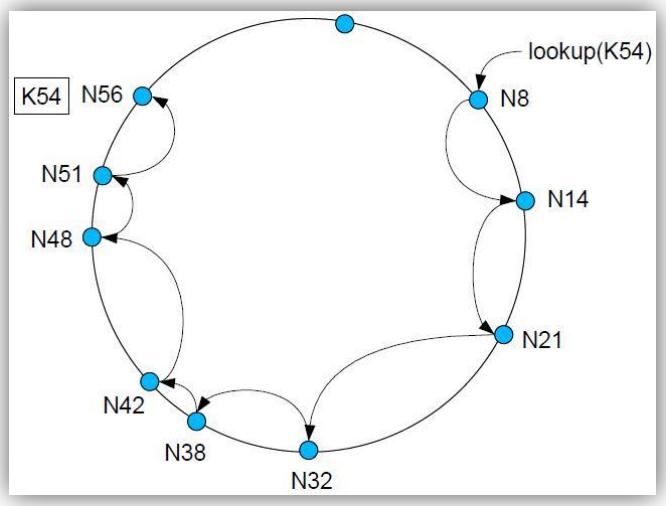

Abbildung 5-3: Lineare Suche in einem Chord-Ring

Für die Realisierung einer schnelleren Suche benötigt jeder Node eine zusätzliche Routing Tabelle, die auch Fingertable genannt wird. Diese Fingertable enthält Verweise auf m Nachfolger im Chord-Ring, wobei m die Anzahl Bits der verwendeten ID ist. Diese Verweise werden Finger genannt. Der letzte Eintrag in der Fingertable zeigt somit auf einen Node der sich mindestens n/2 Nodes vom Ursprung entfernt befindet. Eine Suchanfrage wird jeweils an den am weitesten entfernten Vorgänger des gesuchten Schlüssels weitergeleitet. Bei einer Suche deckt somit jeder Node die Hälfte aller noch möglichen Schlüssel ab. Dies führt dazu, dass die maximale Anzahl Sprünge bei einer Suche auf O(log n) reduziert wird.

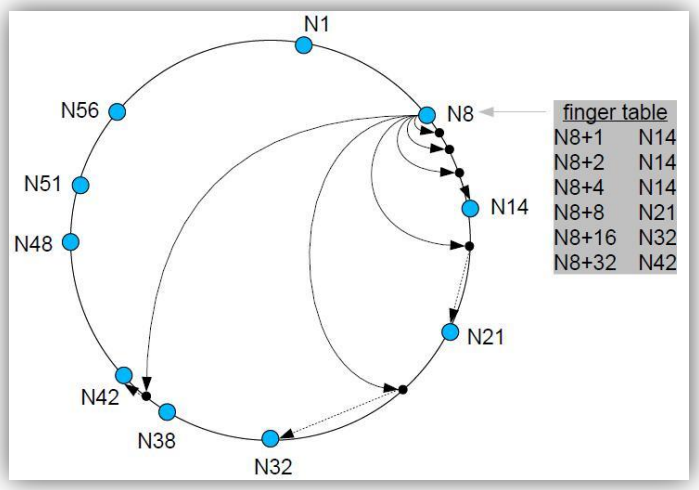

Abbildung 5-4: Fingertable und Finger eines Nodes

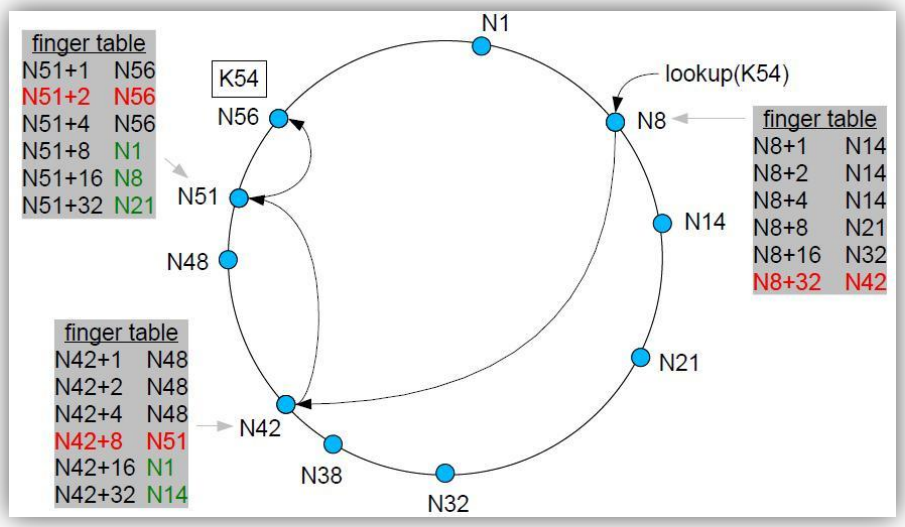

Abbildung 5-5: Skalierbare Suche

6 Anforderungen

6.1 Muss-Anforderungen

6.1.1 REST Protokoll

Die Nodes kommunizieren mit einem eigenen REST-Protokoll untereinander. Das Protokoll verwendet als Darstellungssprache XML und wird mit HTTP übertragen. D.h. es werden eigene Ressourcendarstellungen definiert. Es existieren folgende Ressourcentypen:

- Task
- Job
- Result
- Result-Ressource
- Source

Das REST-Prinzip definiert jeweils einen Client und einen Server. Da in unserem Netz jeder Node jeden Nachrichttypen versenden kann, fungieren die Nodes gleichzeitig als Client und Server.

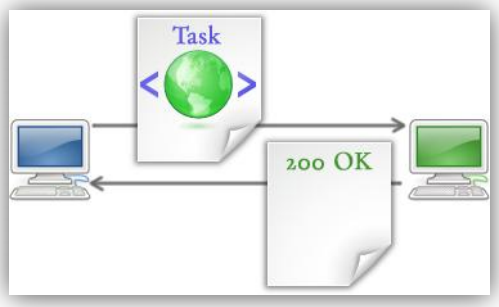

Abbildung 6-1: REST-Prinzip

Mögliche Darstellung einer REST-Nachricht, die einen Task mit der HTTP-Methode PUT anlegt:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<head>
    <version>1.0</version>
    <type>1</type>
</head>
<body>
    <job>
        <id>24</id>
        <name>Multiple Youtube-Link-Converter</name>
        <parameter>
            <link>http://www.youtube.com/watch?v=x2rprKs15A8</link>
            <link>http://www.youtube.com/watch?v=y4kdwke12E8</link>
            <link>http://www.youtube.com/watch?v=z6ruqos1910</link>
        </parameter>
    </job>
</body>
```

Abbildung 6-2: REST-Nachricht

Das Protokoll soll so aufgebaut werden, dass es ohne grossen Aufwand erweitert werden kann.

6.1.2 SON – Clustering aufgrund von Systemanalyse durch Client

Jeder Node hat unterschiedliche Hardware und einen unterschiedlich schnellen Internetzugang. Daher sind gewisse Nodes besser für eine Art von Aufträgen geeignet als andere. Diese Gegebenheiten sollen berücksichtigt werden. Alle Nodes werden Aufgrund ihrer Systemanalyse in unterschiedliche Cluster eingeteilt. So werden zum Beispiel Nodes mit starker Rechenleistung in das eine Cluster und Nodes mit einem schnellen Internetzugang in ein zweites Cluster eingeteilt. Die Information in welchem Cluster sich ein Node befindet, wird ebenfalls in der DHT gespeichert.

6.1.3 Generischer Task

Das System soll verschiedenste Aufträge (Tasks) bearbeiten können. Jeder Benutzer soll in der Lage sein, einen eigenen Task zu erstellen und diesen mit Hilfe des P2P-Netzes bearbeiten zu lassen. Die Tasklogik muss als Quellcode-Datei vom Benutzer zur Verfügung gestellt werden. Diese wird im Task als weitere Ressource verlinkt.

Ein Task soll aufteilbar sein, damit auf verschiedenen Nodes gleichzeitig Taskteile abgearbeitet werden können. Diese Taskteile sind so genannte Jobs. Diese Jobs unterscheiden sich in den zu bearbeitenden Parametern. Ein Benutzer kann eine Liste von parallel bearbeitbaren Parametern im Task definieren. Diese müssen alle vom gleichen Typ sein damit diese dann vom System automatisch aufgeteilt und an die verschiedenen Nodes gesendet werden können. Natürlich können auch Taskparameter angegeben werden, die für jeden Job gleich gelten.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Task xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>TestTaskName</Name>
  <Description>Dieser Task zeigt das Deserialisieren auf.</Description>
  <ConfigParameters>
    <anyType xsi:type="xsd:string">MyString</anyType>
    <anyType xsi:type="xsd:boolean">true</anyType>
    <anyType xsi:type="xsd:int">42</anyType>
  </ConfigParameters>
  <ParallelParameters>
    <KeysAndValues>
      <Key>Jöstin.mp3</Key>
      <Value xsi:type="xsd:string">\\10.10.10.41\Debug\Incoming\0.0.mp3</Value>
    </KeysAndValues>
    <KeysAndValues>
      <Key>Jöstin.mp3</Key><!-- An identical key will not be added!!! -->
      <Value xsi:type="xsd:string">\\10.10.10.41\Debug\Incoming\0.1.mp3</Value>
    </KeysAndValues>
    <KeysAndValues>
      <Key>Siemens.mp3</Key>
      <Value xsi:type="xsd:string">\\10.10.10.41\Debug\Incoming\0.2.mp3</Value>
    </KeysAndValues>
    <KeysAndValues>
      <Key>Fujitsu.mp3</Key>
      <Value xsi:type="xsd:string">\\10.10.10.41\Debug\Incoming\0.3.mp3</Value>
    </KeysAndValues>
  </ParallelParameters>
</Task>
```

Abbildung 6-3: Taskdefinition mit den Config- und Parallel-Parameter

```xml
<?xml version="1.0" encoding="utf-8"?>
<Source xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Version>1</Version>
  <Name>YouTube converter</Name>
  <Description>Convert YouTube-Links into *.mp3 files</Description>
  <Author>Philippe Morier</Author>
  <Code>
    <![CDATA[
      using System;
      using System.Collections.Generic;
      using System.Net;

      namespace P2PTaskbag
      {
          class JobRunable
          {
              public byte[] Run(List<object> configParameters, object param)
              {
                  WebClient client = new WebClient();
                  Console.WriteLine(param.ToString());

                  byte[] res = client.DownloadData(param.ToString());
                  Console.WriteLine("Finish: " + res.LongLength);

                  return res;
              }
          }
      }
    ]]>
    </Code>
</Source>
```

Abbildung 6-4: Source Datei

6.1.4 Konkrete Taskbeispiele

Die konkreten Beispiele sollen die Skalierbarkeit und Effizienz des Netzes aufzeigen. Zudem sollen sie die Machbarkeit unserer Konzepte aufzeigen. Dabei stellen die Beispiele jeweils einen Cluster in unserem SON dar.

6.1.4.1 Multiple YouTube-Link-Converter

Dieser Task wird nur von Nodes im Cluster für schnelle Internetverbindung verarbeitet.

Dieses Beispiel wurde gewählt, um einen Task zu simulieren, der eine grosse Internetbandbreite benötigt. Oft möchte man von einem Musik-Video auf YouTube.com nur die Tonspur. Um dies zu bewerkstelligen stehen diverse online Konverter zur Verfügung (z.B. www.youtube-mp3.org). Diese bieten jedoch immer nur das Konvertieren eines einzelnen YouTube-Links auf einmal an. Möchte man nun mehrere Links konvertieren, muss dies sequenziell geschehen.

Der Multiple YouTube-Link-Converter erlaubt es eine ganze Linkliste anzugeben, die dann im Netz verteilt wird und somit von verschiedenen Nodes parallel verarbeitet werden kann. Dabei ist zu erwähnen, dass die angegebene Linkliste von dem Ausgangs-Node zuerst in atomare Jobs unterteilt wird. Folgende Darstellung soll dies illustrieren.

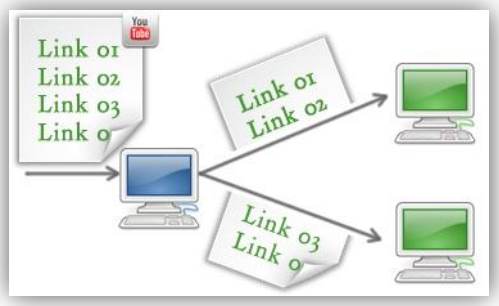

Abbildung 6-5: YouTube-Link-Splitting

6.1.4.2 Password Security Checker

Dieser Task wird nur von Nodes im Cluster für starke CPU-Leistung verarbeitet.

Dieses Beispiel wurde gewählt, um einen rechenintensiven Task zu simulieren. Dabei kann vom Ausgangs-Node, der den Task startet, eine beliebige Zeichenkette (String) angegeben werden. Die Nodes verwenden einen Brute-Force Algorithmus, um die gewünschte Zeichenkette zu generieren. So kann geprüft werden, ob die angegebene Zeichenkette sicher ist. Ausserdem soll dieses Beispiel die Macht resp. die Leistung unseres P2P-Netzes aufzeigen.

Dabei wird der Task in Jobs unterteilt. Jeder Job enthält einen unterschiedlichen Generierungsbereich. Zum Beispiel generiert Node 1 Zeichenketten von A bis J und Node 2 von K bis Z. So wird der gesamte Generierungsbereich in kleinere Bereiche aufgeteilt und von den einzelnen Nodes als Job abgearbeitet.

6.1.4.3 File-Saver

Dieser Task ist für Nodes im Cluster für schnelle Internetverbindungen geeignet und soll zeigen, dass die parallel bearbeitbaren Parameter nicht in jedem Fall benötigt werden.

Unser Netz könnte auch für das Speichern einer bestimmten Datei aus dem Internet sein. So könnte als Konfigurationsparameter eine URL angeben, hinter der eine beliebige Ressource steht. Jeder Node der den Task erhält, lädt sich die gewünschte Datei runter. Da dies mehrere Nodes bewerkstelligen, wird die Datei redundant d.h. sicher im Netz gespeichert.

Hier ist ersichtlich, dass dieser Task nur einen und immer gleichbleibenden Parameter hat. Also keine gleichartige Links, die aufgeteilt werden müssen. Somit verwendet dieser Task die aufteilbaren Parameter nicht.

6.1.5 Voting-Algorithmus

Ein Wahl-Algorithmus könnte für folgende drei Situationen eingesetzt werden.

6.1.5.1 Wahl für Eintrittsnode

Ein neuer Node, der sich in das Netz einklinken möchte, kennt zu Beginn keine andere Nodes. Dies lässt sich entweder durch manuelles Mitteilen vom Benutzer oder durch das zur Verfügung stellen eines Eintrittsnode resp. Bootstrappingnode ändern.

Denkbar wäre, dass der DynDNS-Dienst www.dyndns.org verwendet werden würden. So könnte jeder Node von Zeit zu Zeit nachschauen, ob hinter der DynDNS noch eine gültige resp. aktive IP steht. Ist dies nicht der Fall löst dieser Node eine neue Wahl für den Eintrittsnode aus. Ist dieser gewählt, trägt er seine IP beim DynDNS-Dienst ein. Bei der Wahl des Eintrittsnodes sollte darauf geachtet werden, dass dieser eine hohe Verfügbarkeit aufweist. Die Syntax für das Eintragen einer neuen IP bei DynDNS ist unter folgender Adresse angegeben:

http://www.dyndns.com/developers/specs/syntax.html

6.1.5.2 Wahl eines Garbage-Collector-Nodes

Um zu verhindern, dass immer mehr falsche oder nicht mehr gültige Ressourcen im Netz vorhanden sind, sollten diese von Zeit zu Zeit gelöscht werden. Diese Aufgabe soll von einem oder ev. auch mehrere ausgewählte Nodes durchgeführt werden.

6.1.5.3 Wahl von Publisher für neue Ergebnisse

Je nach Task kann es von Vorteil sein, dass der Task gebende Node benachrichtigt wird, sobald ein neues Ergebnis zu einem bestimmten Task zur Verfügung steht. Es wäre vorstellbar, dass ein Node als Publisher gewählt wird. Bei diesem können sich die an neuen Ergebnissen interessierten Nodes subscriben. Der Publisher teilt, sobald ein neues Ergebnis eingetragen wurde, allen bei ihm eingetragenen Nodes mit, dass neue Ergebnisse zu einem Task X vorliegen.

6.2 Kann-Anforderungen

6.2.1 Sicherheit

Prinzipiell ist jeder Benutzer berechtigt alle Nachrichttypen auf beliebige Tasks zu versenden. Vorstellbar wäre aber, dass nur gewisse IP-Bereiche Zugriff auf das Netzwerk haben. Eine andere Möglichkeit besteht darin, jeweils ein Passwort in den REST-Nachrichten zu verlangen. Dadurch könnte man z.B. nur bestimmten Clients die Erlaubnis zur Verteilung von Tasks geben. So könnte man unterschiedliche Sicherheitslevel einbauen.

6.2.2 Ressourcen-Routing mittels DHT

Ressourcen eines Tasks müssen gespeichert, abgefragt, verändert und gelöscht werden können. Um diese Aktionen durchführen zu können, müssen zuerst die URIs der Ressourcen bekannt sein.

6.2.2.1 Speicherung / Publikation

Sobald eine neue Ressource erstellt wird, muss diese resp. deren URI in der DHT publiziert werden. So wird zum Beispiel nach dem erfolgreichen Abarbeiten eines Jobs eine neue Result-Ressource erstellt. Hat der Client die entsprechende Ressource erstellt, trägt er dessen URI in das Key-Value Pair des entsprechenden Tasks ein. Somit wird die neue Ressource den anderen Nodes bekannt gemacht.

6.2.2.2 Abfragung

Um alle Job-Results zu erhalten muss eine Anfrage an die DHT mit dem passendem Key des Tasks gesendet werden. Daraufhin werden alle URIs der Ressourcen empfangen und können abgefragt werden.

6.2.2.3 Löschung

In einem P2P-Netz ändert sich die Netzwerkstruktur fortlaufend. Neue Nodes kommen hinzu und andere wiederum verlassen das Netz. Durch diesen ständigen Wechsel gehen Ressourcen verloren. Die Verluste führen dazu, dass in der DHT ungültige resp. nicht mehr erreichbare URIs gespeichert sind. Diese müssen gelöscht werden, um ungültige Requests zu verhindern. Das Aufräumen der DHT kann unterschiedlich realisiert werden. Eine Möglichkeit besteht darin, dass ein Node gewählt wird,

dessen Aufgabe darin besteht, ungültige URIs in der DHT zu identifizieren und zu löschen. Er fungiert als eine Art Garbage-Collector Node. Jeder Node, der eine Ressource nicht erreichen kann, teilt dies dem Garbage-Collector Node mit. Diese führt Buch, welche Ressource zum wievielten Mal bereits als „nicht erreichbar“ angegeben wurde. Wurde eine bestimmte Ressource zu oft gemeldet, löscht der Garbage-Collector Node daraufhin den Ressource-URI in der DHT effektiv.

6.3 Use-Case Überblick

6.3.1 Use-Case Diagramm

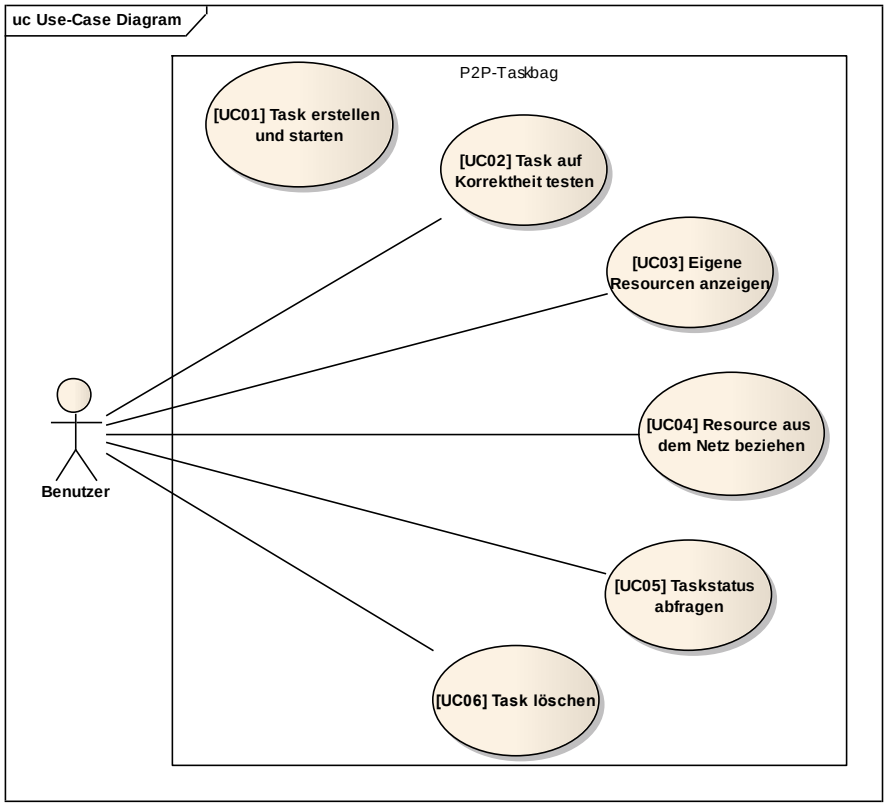

Abbildung 6-6: Use-Case Diagramm

6.3.2 Akteure & Stakeholders

6.3.2.1 Primär Akteur

Benutzer: Der primäre Akteur ist der Benutzer welcher sich hinter einem Node befindet. Jeder Benutzer ist gleichwertig und besitzt somit die gleichen Rechte wie alle andern. Sie alle können jeden Use-Case durchführen.

6.3.3 [UC01] Task erstellen und starten

6.3.3.1 Brief

Ein Benutzer kann einen Task erstellen und diesen im P2P-Netz verteilen und somit die Ausführung des Tasks starten.

6.3.3.2 Fully Dressed

Vorbedingungen: Es muss sowohl eine XML-Datei für den Task wie auch für die Source vorliegen. Diese beschreiben verschiedene Input-Werte und die Verarbeitung dieser. Siehe Abschnitt [6.1.3 Generischer Task].

Standardablauf:

1. Der Benutzer kann unter der Angabe der beiden XML-Dateien und einem vordefinierten Startbefehl den Task starten.
2. Das System testet die beiden XMLs auf Korrektheit und kompiliert den im Source-XML geschrieben Code.
3. Das System generiert eindeutige IDs für die Source und den Task.
4. Das System lässt diese beiden Ressourcen beim Server registrieren.
5. Das System ermittelt geeignete Nodes und teilt anhand dieser den Task in Jobs auf.
6. Das System verteilt die Jobs auf die ermittelten Nodes.
7. Das System erhält die Resultate der einzelnen Jobs, publiziert diese auf dem Server und zeigt sie dem Benutzer an.

6.3.4 [UC02] Task auf Korrektheit testen

6.3.4.1 Brief

Der Benutzer kann die Task-und Source-XML-Datei auf Korrektheit prüfen.

6.3.4.2 Fully Dressed

Vorbedingungen: Es muss sowohl eine XML-Datei für den Task wie auch für die Source vorliegen.

Standardablauf:

1. Der Benutzer gibt den Befehl und die beiden zu prüfenden XML-Dateien an.
2. Das System überprüft die beiden XML-Dateien auf ihre Korrektheit. Dies beinhaltet die Überprüfung der Syntax des XMLs und die Kompilation des Quellcodes.

6.3.5 [UC03] Eigene Ressourcen anzeigen

6.3.5.1 Brief

Der Benutzer kann durch die Eingabe eines Befehles alle Ressourcen, welche sich auf seinem Rechner befinden anzeigen lassen.

6.3.5.2 Fully Dressed

Standardablauf:

1. Der Benutzer gibt den für die Anzeige der Ressourcen erforderten Befehl ein.
2. Das System ermittelt alle auf dem Rechner und zum P2P-Taskbag gehörenden Ressourcen.
3. Das System zeigt die ermittelten Ressourcen an.

6.3.6 [UC04] Ressource aus dem Netz beziehen

6.3.6.1 Brief

Der Benutzer kann unter der Angabe einer Ressource-ID und dem dafür vorgesehenen Befehl die entsprechende Ressource aus dem P2P-Netz beziehen.

6.3.6.2 Fully Dressed

Standardablauf:

1. Der Benutzer gibt den entsprechenden Befehl und die ID der gewünschten Ressource ein.
2. Das System ermittelt alle Nodes, welche im Besitz dieser Ressource sind.
3. Das System beginnt die Ressource von einem ermittelten Node zu beziehen.
4. Sobald das System die Ressource erfolgreich bezogen hat, wird diese auf dem Server publiziert.

6.3.7 [UC05] Taskstatus abfragen

6.3.7.1 Brief

Der Benutzer kann den Status eines Tasks abfragen.

6.3.7.2 Fully Dressed

Standardablauf:

1. Der Benutzer gibt den Befehl und die ID des Tasks im System ein.
2. Das System führt *[UC04] Ressource aus dem Netz beziehen* unter der Angabe der Task-ID aus.
3. Das System führt für jeden Job des Tasks *[UC04] Ressource aus dem Netz beziehen* aus.
4. Das System bestimmt anhand der Job-Status den aktuellen Task-Status.
5. Das System teilt den Task-Status dem Benutzer mit.

6.3.8 [UC06] Task löschen

6.3.8.1 Brief

Eine Task mit allen auf ihn bezogenen Ressourcen (z.B. Job, Result, Source, etc.) kann zu jedem Zeitpunkt gelöscht werden.

6.3.8.2 Fully Dressed

Standardablauf:

1. Der Benutzer gibt den Befehl für die Löschung und die ID des zu löschenden Tasks ein.
2. Das System ermittelt alle Nodes, welche im Besitz dieses Tasks oder einer zu diesem Task dazugehörigen Ressource sind.
3. Das System schickt jedem ermittelten Node den Befehl zur Löschung der entsprechenden Ressourcen.

7 Grob-Architektur

Das folgende Kapitel beschreibt den Aufbau des P2P-Taskbags. Das Klassendiagramm sowie die Sequenzdiagramme und Zustandsdiagramme sollten für eine graphische Darstellung und einem besseren Verständnis des P2P-Taskbags dienen.

7.1 Klassendiagramm

Das unten dargestellte Klassendiagramm zeigt die wichtigsten Klassen und deren Zusammenhänge auf. Es ist nicht vollständig, und auf die Darstellung der Attribute und Methoden wurde bewusst verzichtet, um die Übersicht zu bewahren. Einerseits sind die Services, deren Implementierungsklassen und die übergeordneten Interfaces ersichtlich (Blau markiert). Grundsätzlich besteht jeder Service aus einem Interface, welches von `IService` ableitet, und mindestens einer Klasse, welche die Logik des Services implementiert. Andererseits sind die Ressourcen und deren Verwaltungs-, Verarbeitungs- und Verteilungsklassen ersichtlich (Rot markiert). Auf die Darstellung einiger Klassen wurde jedoch gänzlich verzichtet. Es handelt sich dabei hauptsächlich um Klassen, welche nicht direkt in Beziehung mit den anderen stehen. Die meisten dieser nicht dargestellten Klassen sind sogenannte „Helper“-Klassen und bieten Funktionen, die jederzeit verwendet werden können.

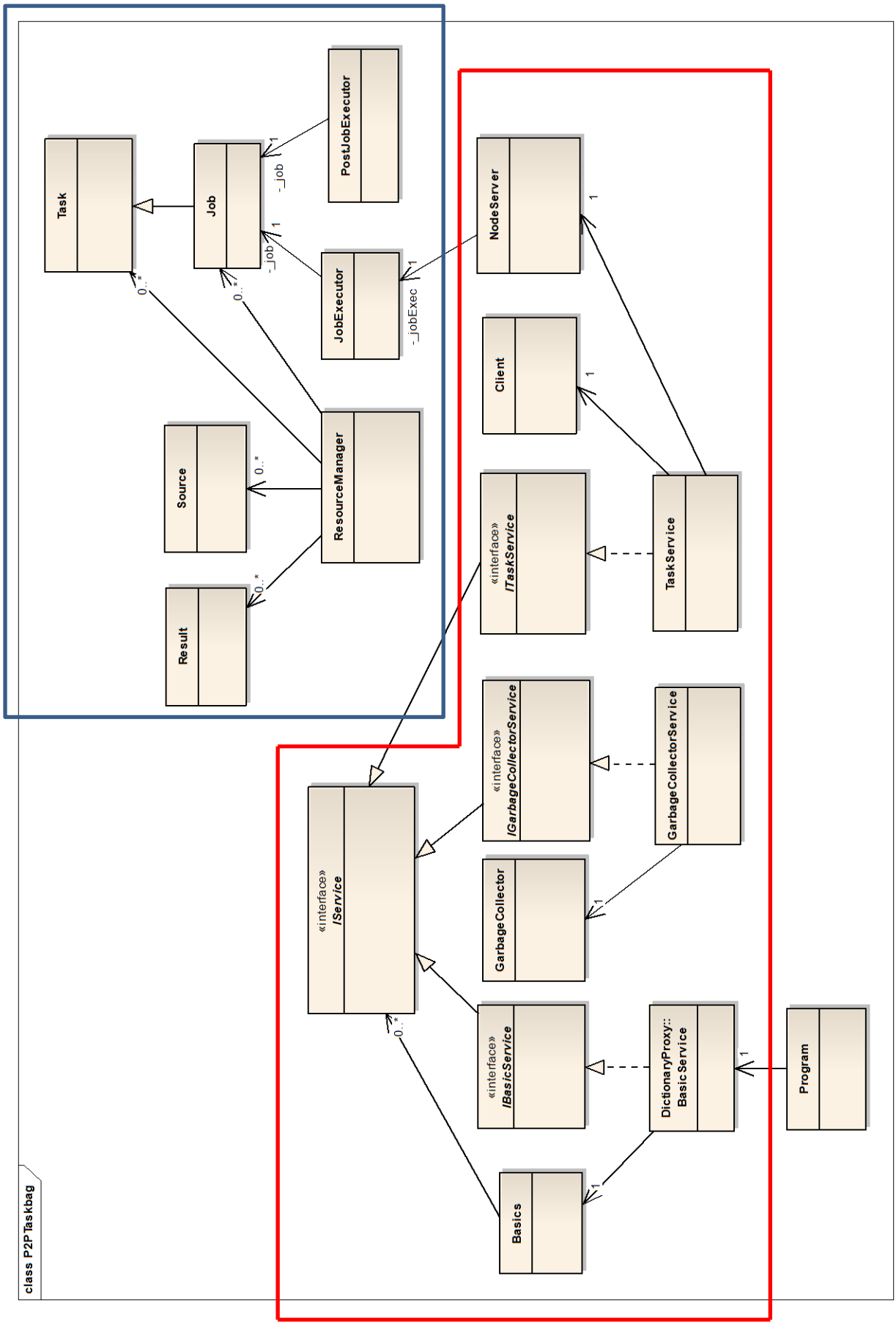

Abbildung 7-1: P2P-Taskbag Klassendiagramm

7.2 REST

Da ein Task in ein Peer-To-Peer Netz gesendet wird, muss dieser aufgeteilt werden. Durch diese Aufteilung ist es möglich, dass mehrere Nodes gleichzeitig an dem Task arbeiten können. REST muss jedoch den Task nicht nur auf mehrere Nodes teilen, sondern muss die entstandenen Results wieder zusammenführen. Das beschriebene Konzept identifiziert folgende Objekte als Ressourcen.

7.2.1 Taskdefinition

Der Benutzer muss als erster Schritt eine Taskdefinitionsdatei (Task.xml) erstellen. Dieses wird vom System verarbeitet und als eine neue Task-Ressource auf den lokalen Node erstellt. Eine Taskdefinition beschreibt einen Task. Sie beinhaltet die Config- sowie auch die Parallel-Parameter. Hinzu kommen bestimmte Taskeigenschaften, wie zum Beispiel der Name und eine kurze Beschreibung des Tasks. Ein Beispiel einer Task-XML ist in der Abbildung [Abbildung 6-3] ersichtlich. Die meisten verwendeten Tags sollten selbstbeschreibend sein. Die wichtigsten Tags sind in den folgenden Abschnitten noch genauer beschrieben

7.2.1.1 Config-Parameters

In den Config-Parameter können diverse Werte angegeben werden. Diese sind über den ganzen Task hinweg eindeutig. Alle Jobs besitzen somit die gleichen Config-Parameter. Diese Parameter können zwar im Quellcode verwendet werden, aber sie können nicht parallelisiert werden.

7.2.1.2 Parallel-Parameters

Zwischen diesen Tags können viele aber einheitliche Parameter angegeben werden. Diese Parameter werden dem Quellcode übergeben und stellen die Parameter für den Task dar. Sie werden auf die Jobs verteilt und können somit parallelisiert werden. Deshalb ist es wichtig, dass ein Paralleler-Parameter eine möglichst kleine Einheit darstellt, dadurch können sie besser auf die Jobs verteilt werden und zudem ist eine bessere Parallelisierung möglich.

7.2.2 Source

Die Source-Datei (Source.xml) enthält den effektiven C# Code. Der Task sowie auch die einzelnen Jobs verweisen auf die Source-Datei. Dabei ist zu beachten, dass der Namespace „P2PTaskbag“ und der Klassenname „JobRunable“ lauten muss. Die Methodensignatur muss der im folgenden Bild gezeigten Signatur entsprechen.

```
namespace P2PTaskbag
{
	class JobRunable
	{
		public byte[] Run(List<object> configParameters, object parallelParam)
		{
			…
		}
	}
}
```

Table 7-1: Struktur der Source.xml

In der Abbildung [Abbildung 6-4] ist ein Beispiel-XML ersichtlich.

7.2.3 Task

Beim Start eines Tasks werden die beiden oben beschriebenen Ressourcen deserialisiert und auf die entsprechenden Klassen abgebildet. Nach der Deserialisierung hat der Task einen Verweis auf die Source. Sobald diese korrekt eingelesen wurden, wird der Task aufgrund verschiedener Kriterien in mehrere Jobs aufgeteilt. Ein Job beinhaltet somit nur eine Teilmenge der zu bearbeitenden Parallel-Parameter. Die Jobs werden anschliessend auf mehrere verschiedene Nodes verteilt und dort als Job-Ressource abgespeichert.

7.2.4 Job

Eine Job-Ressource beschreibt jeweils ein Teil des zu erledigenden Tasks und beinhaltet wie oben beschrieben nur eine Teilmenge der zu bearbeitenden Parallel-Parameter. Ein Job verweist immer auf seinen übergeordneten Task. Ein Node der eine Job-Ressource erfolgreich erhalten hat, beginnt anschliessend mit dem Abarbeiten. Dabei wird zuerst die verlinkte Source-Ressource vom entsprechendem Node geladen und anschliessend kompiliert. Nach dem Kompilieren beginnt der Node mit dem effektiven Abarbeiten des Jobs. Unter abarbeiten wird das einzelne Bearbeiten der parallel bearbeitbaren Parametern verstanden.

7.2.5 Result

Eine Result-Ressource beschreibt und beinhaltet das Ergebnis, das als Output eines Jobs resultiert. Damit grosse Dateien nicht immer mitübertragen werden müssen, sobald eine Ergebnisanfrage eintrifft, kann eine Result-Ressource auf weitere Ressourcen, wie zum Beispiel MP3- oder JPG-Ressourcen, verweisen. Die jeweiligen Nodes senden dem Node, der die Task-Ressource besitzt, die

`Result`-Ressourcen. Somit schliesst sich der Kreis und der Ausgangs-Node erhält das gewünschte Ergebnis. Die folgende Abbildung sollte die soeben beschriebene Architektur aufzeigen.

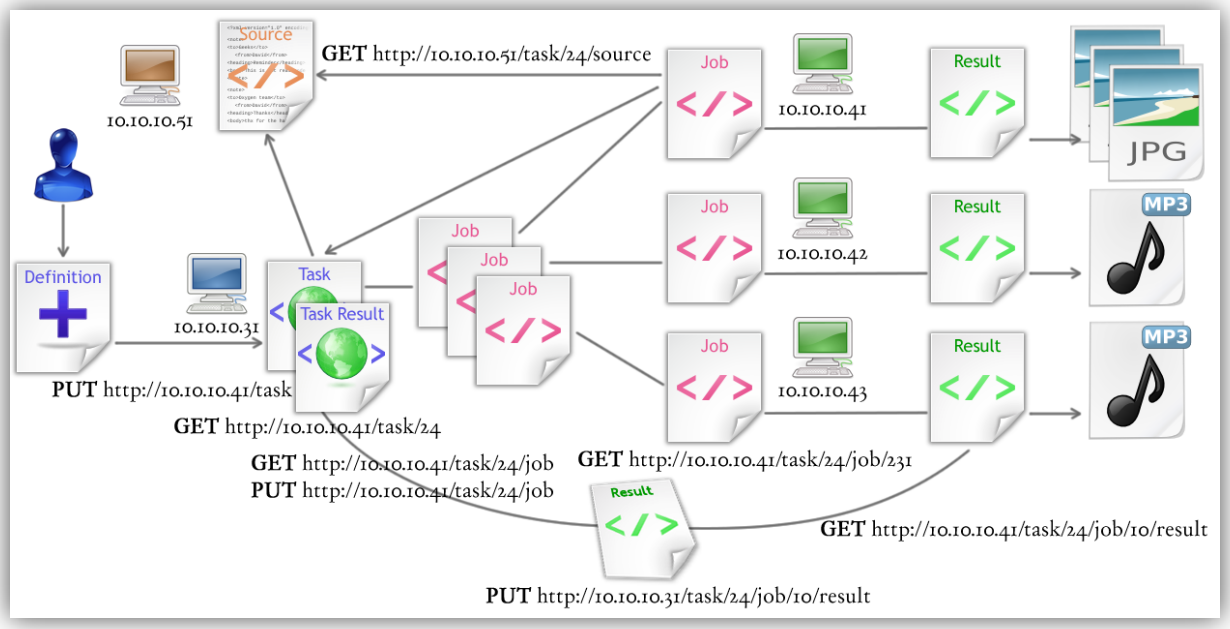

Abbildung 7-2: REST Übersicht

7.3 Zentraler Server

Eine zentrale Frage lautet: Wie findet ein Node, der die `Task`-Ressource nicht selbst besitzt, die zugehörigen `Result`-Ressourcen? Ursprünglich war geplant, die Lösung dieses Problems durch den Einsatz einer DHT zu realisieren. Aus verschiedenen Gründen entschieden wir uns allerdings das Problem mit dem Einsatz eines zentralen Servers zu lösen. Gründe für diesen Entscheid sind im Abschnitt [12.1 NChord Implementation] genauer beschrieben. Jeder Node muss nun alle erstellten oder erhaltenen Ressourcen beim zentralen `Server` registrieren. Der zentrale `Server` fügt dabei die IP des Nodes unter dem entsprechenden Key hinzu. Als Key wird jeweils die ID der zu registrierenden Ressource gewählt. Sobald ein Node eine Ressource benötigt, kann er den zentralen `Server` mithilfe des Ressourcen-Keys abfragen und erhält als Ergebnis alle IP-Adressen, welche im Besitz dieser Ressource sind. Fügt der fragende Node nun die erhaltene IPs mit den jeweiligen Keys zusammen, ist dieser nun im Besitz der kompletten IDs resp. URIs der Ressourcen. Je nach Situation kann er nun einen oder mehrere dieser Nodes nach der Ressource abfragen. Folgendes Beispiel zeigt der Inhalt der Key-Value Pairs wie sie auf dem Server gespeichert werden. Für die anderen Ressourcen, wie zum Beispiel `Job` oder `Task`, wird nach demselben Prinzip verfahren.

<Key>	**<Value>**
</task/24/job/231/result>	<10.10.10.41:6789; 10.10.10.42:9876; … >
</task/25/job/962/result>	<10.10.10.42:9876; 10.10.10.43:9988; … >

Tabelle 7-1: Key-Value Pair Aufbau beim zentralen Server

Der zentrale Server sollte natürlich wenn möglich immer verfügbar sein. Muss der Server trotzdem, z.B. wegen Wartungsarbeiten, heruntergefahren werden, so werden alle Ressourcen in entsprechende XML-Dateien gespeichert. Die Ressourcen werden beim Neustart aus der XML-Datei geladen und sind somit gleich wieder verfügbar. Zu einem späteren Zeitpunkt sollte die Verfügbarkeit z.B. durch den Einsatz eines skalierbaren Server oder einer DHT verbessert werden [10.1 Verbesserung des zentralen Servers].

7.4 SON mittels Systemanalyse und ADS

Ein SON wird in zwei Formen umgesetzt. Einerseits werden die Nodes anhand Ihrer Systemressourcen eingeteilt und anderseits werden die durch das Abarbeiten der Tasks generierten Result-Files mit Eigenschaften versehen. Durch das Setzen von Eigenschaften der Result-Files lassen sich diese einteilen und somit clustern.

7.4.1 Clustering durch Systemanalyse

Um einen Node einem Cluster anhand seiner Systemressourcen zuzuteilen, müssen diese zuerst ausgelesen werden. Das von Microsoft stammende .Net-Framework bietet dafür die Klasse ManagementObjectSearcher an. Diese Klasse befindet sich im Namensraum System.Management. Damit ist es möglich mit einer SQL-ähnlicher Abfrage praktisch jede beliebige Systeminformation auszulesen.

```
ManagementObjectSearcher s = new ManagementObjectSearcher("SELECT * FROM
Win32_Service");

foreach (ManagementObject service in s.Get())
{
        // show the instance
        Console.WriteLine(service.ToString());
}
```

Tabelle 7-2: Systeminformationen mittels .Net auslesen

Die benötigten Informationen werden in die entsprechende Klasse NodeInfo abgefüllt und werden auf dem Server abgespeichert. Einerseits werden die so gesammelten Informationen für das Einteilen der Nodes in Clusters verwendet und anderseits werden Sie für das Voting benötigt. Weitere Informationen betreffend dem Ablauf eines Votings sind im Kapitel [7.5 Voting] ersichtlich

7.4.2 Clustering durch Setzen von Dateieigenschaften

Das Dateisystem NTFS von Microsoft unterstützt so genannte alternative Datenströme resp. Alternate Data Stream (ADS). ADS sind separate Streams, die neben dem Haupt-Stream der Datei existieren. D.h. eine Datei die auf einer NTFS formatiertem Speichermedium liegt kann mehrere Datenströme besitzen.

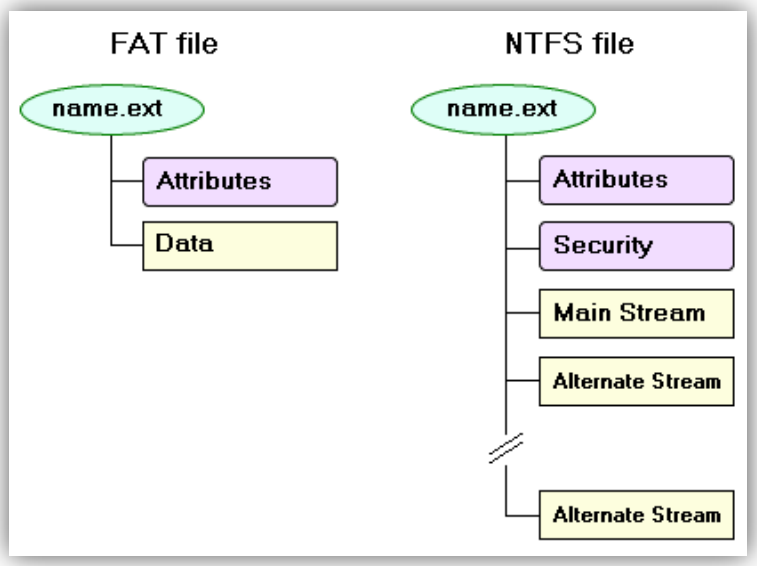

Abbildung 7-3: Alternative Datenströme im NTFS

Diese alternativen Datenströme werden hier benutzt um Metadaten abzuspeichern. Diese zusätzlichen Informationen werden für das Kategorisieren und Einteilen in die entsprechenden Clusters verwendet. Alternative Datenströme sind so in der oben beschriebenen Form und in der verwendeter Art und Weise nur auf einem NTFS formatierten Speichermedium verwendbar. FAT zum Beispiel bietet keine alternativen Datenströme. Darüber hinaus, gehen die in den alternativen Datenströmen gespeicherten Informationen verloren, sobald die betroffene Datei beispielsweise über FTP oder HTTP übertragen wird. Diese Eigenschaft beeinflusst unsere Verwendung negativ und bedingt eine spezielle Übertragung. In unserem Fall wird einfach der verwendete Alternativstrom zusätzlich zum Hauptstrom mitübertragen und auf der Gegenseite entsprechend deserialisiert. Interessant ist, dass der Windows Explorer weder die Möglichkeit bietet solche Datenströme zu bearbeiten noch zeigt er dessen Grösse in den Dateieigenschaften an. Ohne weitere Tools zu verwenden, ist das Benutzen der alternativen Datenströme nur über die Konsole möglich. [13]

Abbildung 7-4: Zugriff auf ADS per Konsole

Für das Auslesen der alternativen Datenströme per Software wird eine bereits existierende Library verwendet. Diese ermöglicht ein einfaches Handling für das Ein-/Auslesen und Bearbeiten der Datenströme. [14]

```
AlternateDataStreamInfo adsInfo = FileSystem.GetAlternateDataStream(_filePath,
       "P2PTaskbag");
using (FileStream file = adsInfo.OpenWrite())
{
       _serializer.WriteObject(file, fileInfo);
}
```

Tabelle 7-3: Setzen von alternativem Datenstrom

Der oben aufgezeigte Code zeigt wie einfach es ist, ein alternativer Datenstrom, hier mit dem Namen „P2PTaskbag“, zu setzen. Die Klassen `FileSystem` und `AlternateDataStreamInfo` erlauben in vergleichbar einfacher Art und Weise das Auslesen und Löschen der Datenströme.

7.5 Voting

Grundsätzlich sind alle Nodes in einem P2P-System gleichwertig. Sie können die gleichen Aufgaben tätigen und besitzen die gleichen Rechte. Trotzdem gibt es gewisse Aufgaben, welche nicht von allen Nodes übernommen oder zu jeder Zeit durchgeführt werden dürfen. Jeder Node wäre theoretisch für die Ausführung dieser Tätigkeit in der Lage. Oftmals eignen sich jedoch nicht alle Nodes gleichermassen dafür. Aufgrund besserer Hardwareleistung, höherer Bandbreite oder längerer Uptime eignen sich bestimmte Nodes besonders gut für diese Tätigkeiten. Um den für eine Aufgabe am besten geeignete Node zu finden, wurde ein Algorithmus für das Voting implementiert.

7.5.1 Algorithmus

1. Ein Node im P2P-System möchte, dass eine spezielle Tätigkeit durchgeführt wird. Er fragt den Server also nach der IP des dafür zuständigen Nodes ab. Ist kein Node vorhanden oder der aktuell eingetragene nicht erreichbar, so wird ein Voting ausgelöst.
2. Es werden sämtliche `NodeInfo`s vom Server bezogen.
3. Aufgrund dieser `NodeInfo`s kann nun der am besten geeignete Node bestimmt werden.
4. Nach der Bestimmung dieses Nodes wird ihm eine Nachricht zugestellt.
5. Nach Empfang der Nachricht startet der Node einen weiteren Service, welcher zur Erfüllung der geforderten Aufgaben benötigt wird.

7.5.2 Wahl eines Garbage-Collector

Der Garbage-Collector ist einer dieser „speziellen“ Nodes. Seine Aufgabe ist es, die registrierten Ressourcen auf dem Server aktuell zu halten. Siehe dazu Kapitel [6.2.2.3 Löschung]. Für die Wahl eines geeigneten Garbage-Collector-Nodes wird nach dem im Abschnitt [7.5.1 Algorithmus] beschriebenen Algorithmus vorgegangen. Das Voting wird ausgelöst sobald die Response eines Requests eine Exception wirft, sich der zurückgegebene Status einen ConnectFailure anzeigt und noch kein anderer Node als Garbage-Collector gewählt wurde oder dieser nicht erreichbar ist.

```
try
{
	…
	response = (HttpWebResponse)request.GetResponse();
	…
}
catch (WebException e)
{
	…
	if(e.Status==WebExceptionStatus.ConnectFailure)
	{
		//Server nach aktuellem Garbage-Collector anfragen
		//Falls kein Garbage-Collector vorhanden oder eingetragener nicht
		//erreichbar
		//Voting auslösen
		…
```

Tabelle 7-4: Auslösung eines Votings

Ein geeigneter Garbage-Collector wird anhand einer Liste aller NodeInfos gewählt. Mögliche Kriterien für die Wahl sind unter anderem die CPU-Leistung, Uptime und Bandbreite des Nodes. In der Implementation wird der Garbage-Collector Node anhand der CPU-Leistung des entsprechenden Computers bestimmt. Dies darum, weil die anderen Kriterien noch nicht erfasst werden oder weniger Sinn machen würden. Um die Liste aller NodeInfos vom Server anzufordern, wird folgender Code verwendet.

```
List<NodeInfo> nodeInfos;
…
new RestWebRequest().MakeGet(new Uri("http://" + serverIp + "/nodeinfo"), out
nodeInfos)
…
```

Tabelle 7-5: Alle NodeInfos beim Server beziehen

Diese Liste wird aufgrund der CPU-Taktrate der NodeInfo sortiert. Dies ist lediglich eine beispielhafte Implementation. Je nach Voting kann die Liste anhand anderer Eigenschaften sortiert und somit einen für die Aufgabe geeigneten Node bestimmt werden.

```
nodeInfos.Sort(CompareNodeInfoByCpu);
```

Tabelle 7-6: Sortierung der NodeInfo-Liste

```
private static int CompareNodeInfoByCpu(NodeInfo nodeInfo1, NodeInfo nodeInfo2)
{
	return nodeInfo1.MaxClockSpeed.CompareTo(nodeInfo2.MaxClockSpeed)*-1;
}
```

Tabelle 7-7: Algorithmus zur Sortierung der Liste

Anschliessend wird jeweils der Reihe nach, beginnend beim ersten Element in der Liste, versucht einen Garbage-Collector zu wählen.

```
for (int i = 0; i < nodeInfos.Count; i++)
{
	if (new RestWebRequest().MakePost(new Uri(nodeInfos[i].Ip + "/basic/vote/gc"),
	    "") == HttpStatusCode.OK)
	{
		break;
	}
	Logger.LogInfo("No answere from voted node! Try Next!");
}
```

Tabelle 7-8: Nachricht an den potenziellen Garbage-Collector senden

Sobald ein Node einen POST-Request auf das Uri-Template "/vote/{role}" erhält, startet dieser je nach mitgegebener {role} den entsprechenden Service und registriert diesen beim Server.

```
switch (role)
{
	case "gc":
		Add<IGarbageCollectorService>(nodePort, new GarbageCollectorService(),
		"gc");
		…
		//Register this node as GarbageCollector on Server
		new RestWebRequest().MakePost(new Uri("http://" + serverIp +
		"/voted/gc"), port)
	…
```

Tabelle 7-9: Neuen Service starten und registrieren

Wird der HTTP-Statuscode „200 OK“ zurückgeliefert, ist das Voting abgeschlossen.

7.6 Sequenzdiagramme

Im folgenden Abschnitt sollen die wichtigsten Abläufe dargestellt werden. Dabei ist zu erwähnen, dass einige komplexe Teilschritte vereinfacht und nicht komplett dargestellt sind. Dies soll eine bessere Übersicht bieten und so zu mehr Verständnis beitragen. Zu Beginn wird die Startsequenz eines Nodes beschrieben. Danach wird der Ablauf des Nodes in der Rolle als Client und Server bei einer Taskbearbeitung aufgezeigt. Zum Schluss wird das Verhalten eines Nodes, der einen Garbage-Collector Service in Ausführung hat, beschrieben.

7.6.1 Startup

Jeder Node startet jeweils den `BasicService`. Dieser Service stellt die Grundfunktionalität zur Verfügung und läuft somit auf jedem Node. Nach der Systemanalyse werden die ausgelesenen Daten, die IP und der verwendete Port auf dem Server registriert.

Als zweiter Service wird standardmässig ein `TaskService` geladen. Dieser Service ist für das Geschehen rund um den `Task` zuständig. Ressourcen-Anfragen oder POST-Requests welche einen `Task` betreffen, werden über den `TaskService` abgewickelt. Wurde dieser mit all seinen Daten erfolgreich gestartet, beginnt der Node auf die Benutzereingaben zu warten.

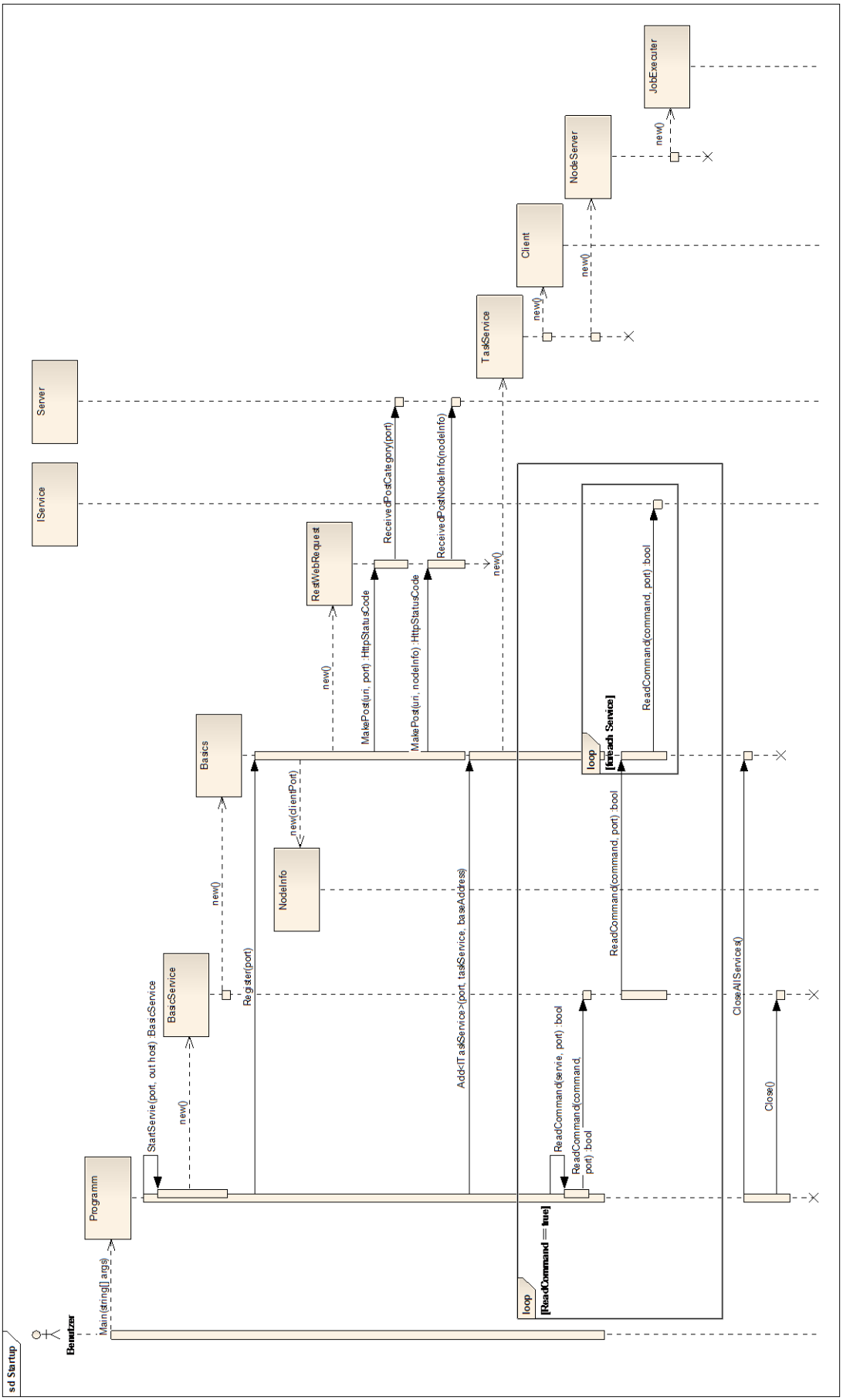

Abbildung 7-5: Start eines Nodes

7.6.2 Client

Sobald der Benutzer den Startbefehl per Konsole auslöst, testet das System die angegebene XML-Datei auf Korrektheit. Dabei ist zu erwähnen, dass der Prüfvorgang mit nur einer Methode im Diagramm zusammengefasst wurde.

Sobald die Inputdaten korrekt sind, werden die entsprechenden Daten im `ResourceManager` eingetragen. Dieser wiederum registriert die Daten beim zentralen `Server`. Nach erfolgreicher Registration wird der `Task` gestartet und dabei in mehreren `Job`s aufgeteilt. Diese müssen auf die jeweiligen Nodes verteilt werden, damit diese dann den einzelnen `Job` ausführen können. Zum Schluss erhält der Auftrag gebende Node alle Ergebnisse der einzelnen Nodes.

Man beachte, dass ganz rechts eine Lebenslinie NodeServer dargestellt ist, welche, die komplette Rolle des Nodes als Server repräsentiert resp. abstrahiert. Dieser ist im nächsten Kapitel detailliert beschrieben.

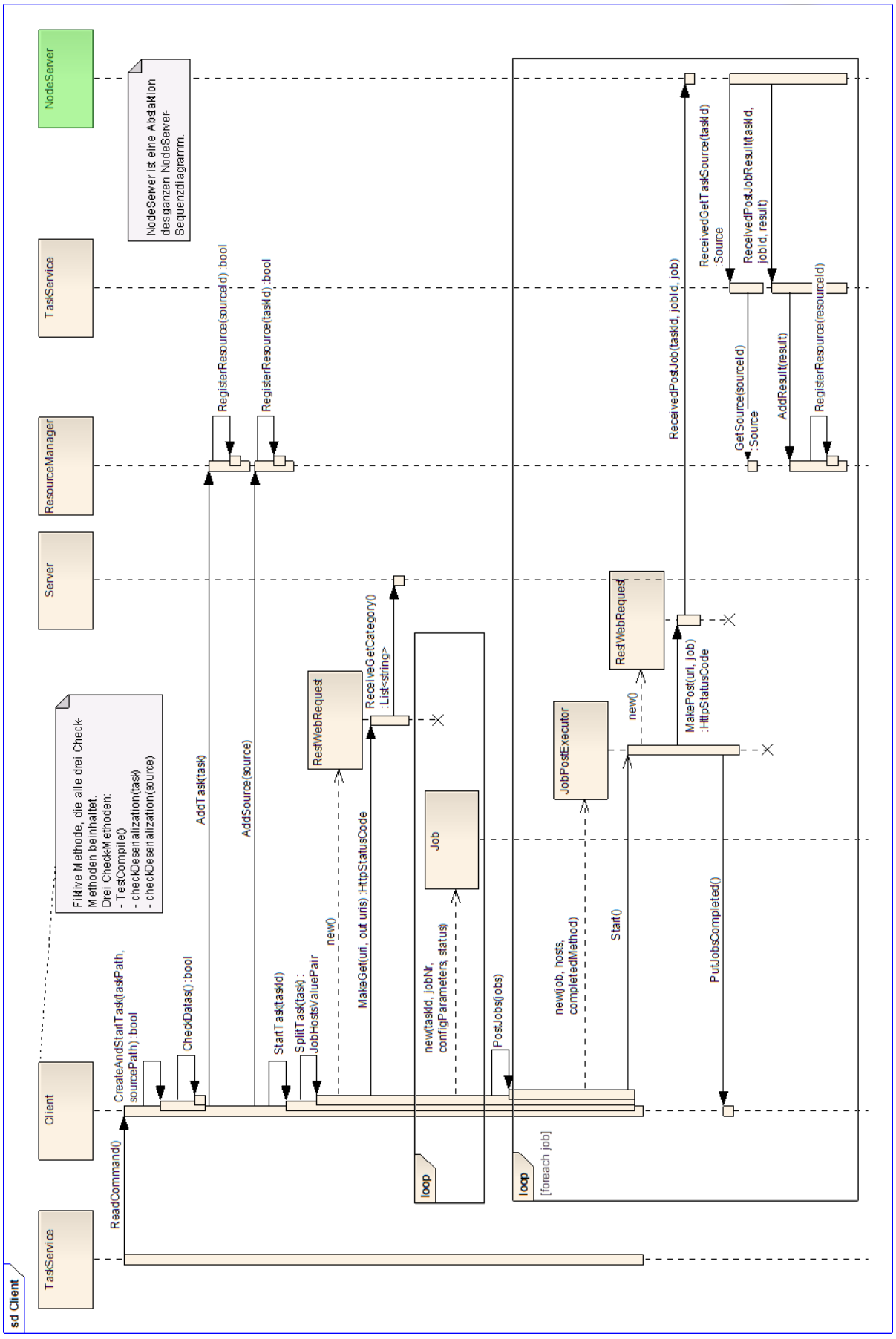

Abbildung 7-6: Ablauf des Clients während der Verteilung eines Tasks

7.6.3 NodeServer

Im letzten Kapitel wurde beschrieben, dass die einzelnen `Job`s auf verschiedene Nodes aufgeteilt werden. Nun wird die Reaktion eines Nodes auf den Erhalt eines solchen `Job`s beschrieben. Wir befinden uns also im Server-Teil des `TaskService` eines Nodes.

Sobald der `NodeServer` einen `Job` erhält, wird dieser im `ResourceManager` eingetragen. Anschliessend wird ein neuer `JobExecuter`, der die wirkliche Arbeit erledigt, gestartet. Der neu erstellte `JobExecuter` muss als erstes den Quellcode beschaffen. Nach dessen Erhalt wird der Quellcode kompiliert und ausgeführt.

Das entstandene Ergebnis wird im `ResourceManager` abgelegt. Danach wird der `Server` nach dem Auftraggeber gefragt. Sobald der entsprechende Node bekannt ist, wird diesem das `Result` gesendet.

Wieder ist zu beachten, dass ganz links eine Lebenslinie Client aufgeführt ist, die das ganze Sequenzdiagramm vom vorherigem Kapitel abstrahiert.

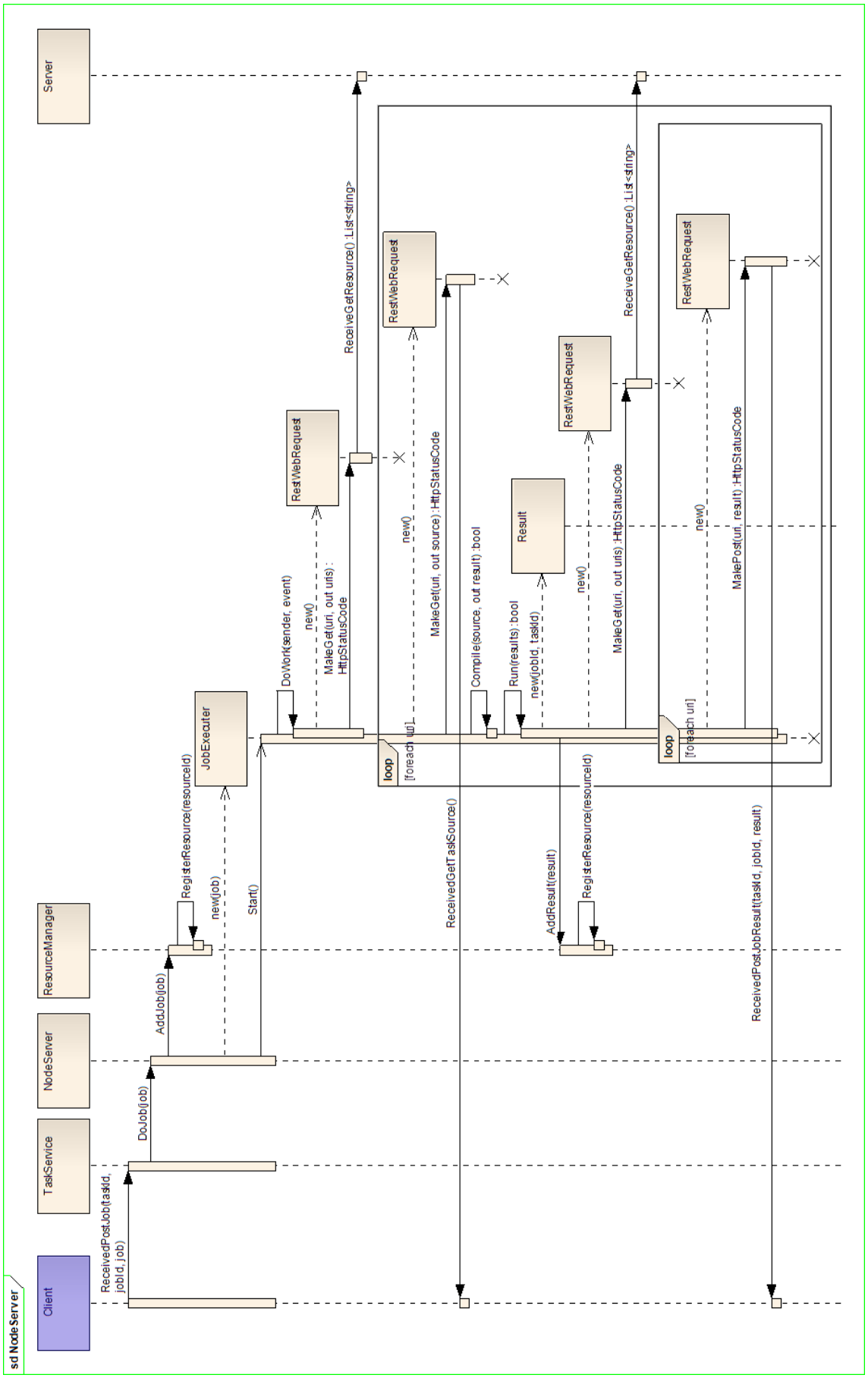

Abbildung 7-7: Ablauf des NodeServer beim Erhalt eines Jobs

7.6.4 Garbage-Collector und Voting

Erteilt der Benutzer den Befehl zum Aufräumen, wird der zentrale `Server` nach der IP des aktuell gewählten Garbage-Collector Node gefragt. Nun wird getestet, ob hinter der erhaltenen IP wirklich ein aktiver Node steht. Ist dies der Fall wird ihm der Befehl zum Aufräumen erteilt. Dieser geht anschliessend durch alle Ressourcen auf dem `Server` durch und prüft diese auf dessen Erreichbarkeit. Ist eine Ressource nicht erreichbar, wird diese vom `Server` gelöscht

Ist die vom `Server` erhaltene IP nicht erreichbar, löst der Node den Voting-Algorithmus aus. Dabei wird die erhaltene Liste von `NodeInfo`s nach der passender Eigenschaft absteigend sortiert. Anschliessend wird so lange die Schleife vorgesetzt, bis der erste Node erreichbar ist und ihm die neue Rolle als Garbage-Collector erfolgreich aufgetragen wurde. Siehe [7.5 Voting].

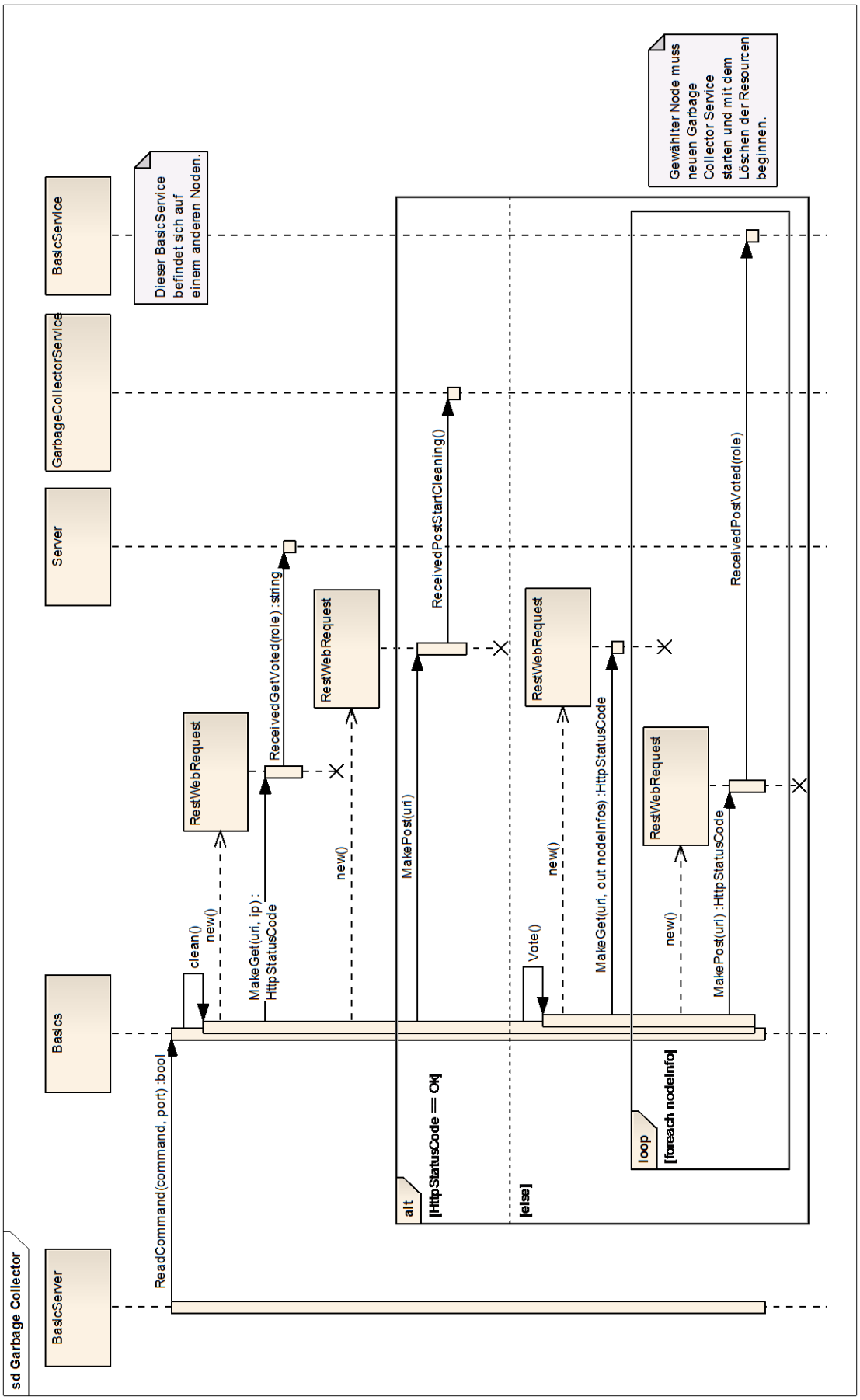

Abbildung 7-8: Ablaufdiagramm des Garbage-Collector

7.7 Zustandsdiagramme

Die Zustandsdiagramme im folgenden Kapitel zeigen grob auf wie der P2P-Taskbag funktioniert und sollten für das Verständnis des P2P-Taskbags von Nutzten sein. Sie sind teilweise etwas vereinfacht dargestellt aber zeigen die wichtigsten Zusammenhänge auf.

7.7.1 Client

Das erste Zustandsdiagramm zeigt einen Node auf, welcher sich in der Rolle des Clients befindet. Er ist also derjenige, der einen `Task` in Auftrag gibt oder andere Ressourcen abfragt. Nach dem Erstellen eines `Task`s, wird dieser im Zustand „Aufteilend“ in mehrere Jobs aufgeteilt. Diese werden anschliessend im Zustand „Job verteilend“ an diverse `NodeServer` verschickt. Der Zustand „Job verteilend“ wird hierbei für jeden Job in einem eigenen Thread aufgerufen und ist in [Abbildung 7-10] detaillierter dargestellt.

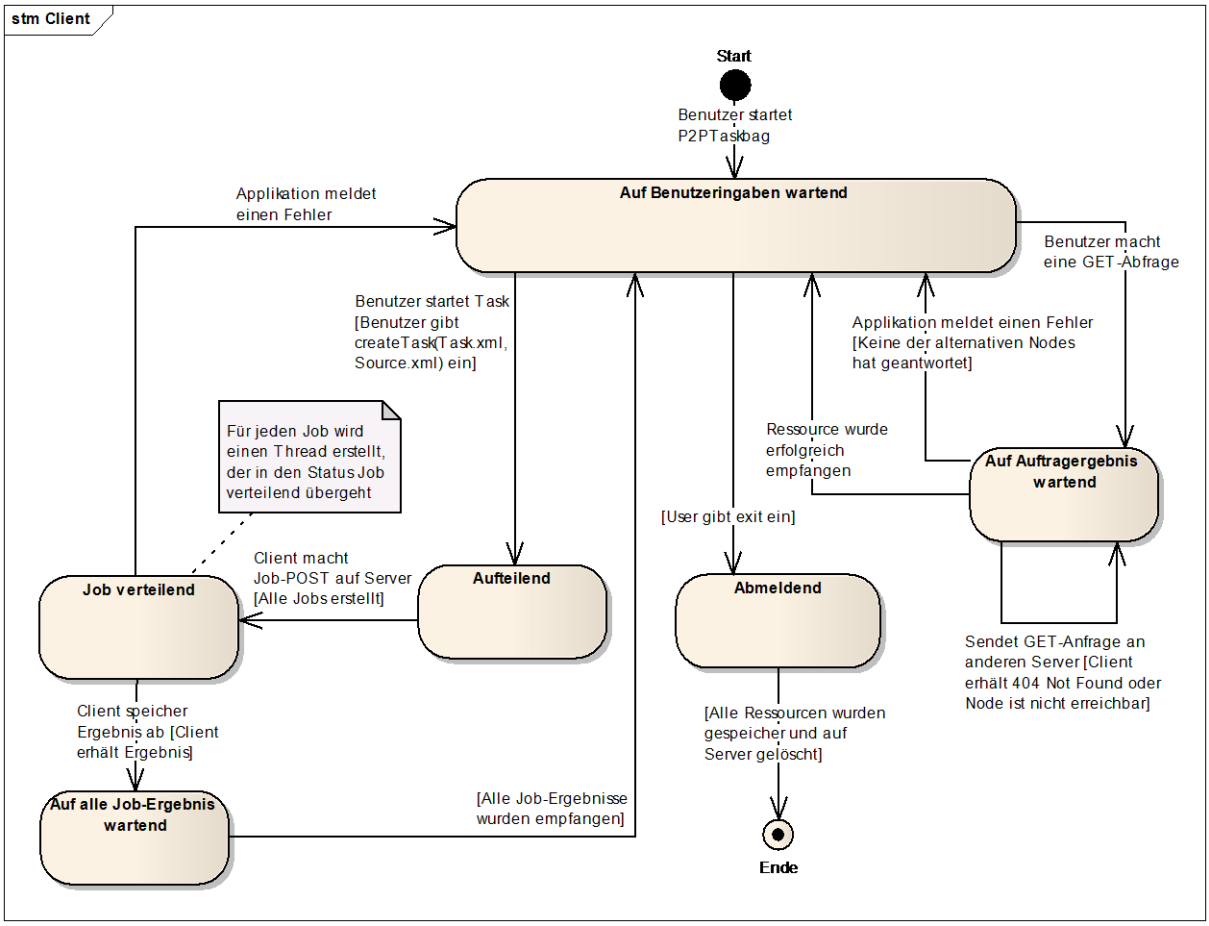

Abbildung 7-9: Ein Node in der Rolle als Client

Das nächste Zustandsdiagramm ist eine Detailansicht des Zustands „Job verteilend“ und stellt einen Thread dar, welcher einen `Job` an einen `NodeServer` sendet und auf das Ergebnis wartet. Ebenfalls ersichtlich ist auch das Vorgehen des Systems, wenn es einen `Job` nicht korrekt verschickt kann. Je

nach Fehlermeldung muss es anders reagieren und den `Job` nochmals versuchen zu versenden, weiter warten oder den Vorgang abbrechen und dem Benutzer eine Fehlermeldung zurück liefern.

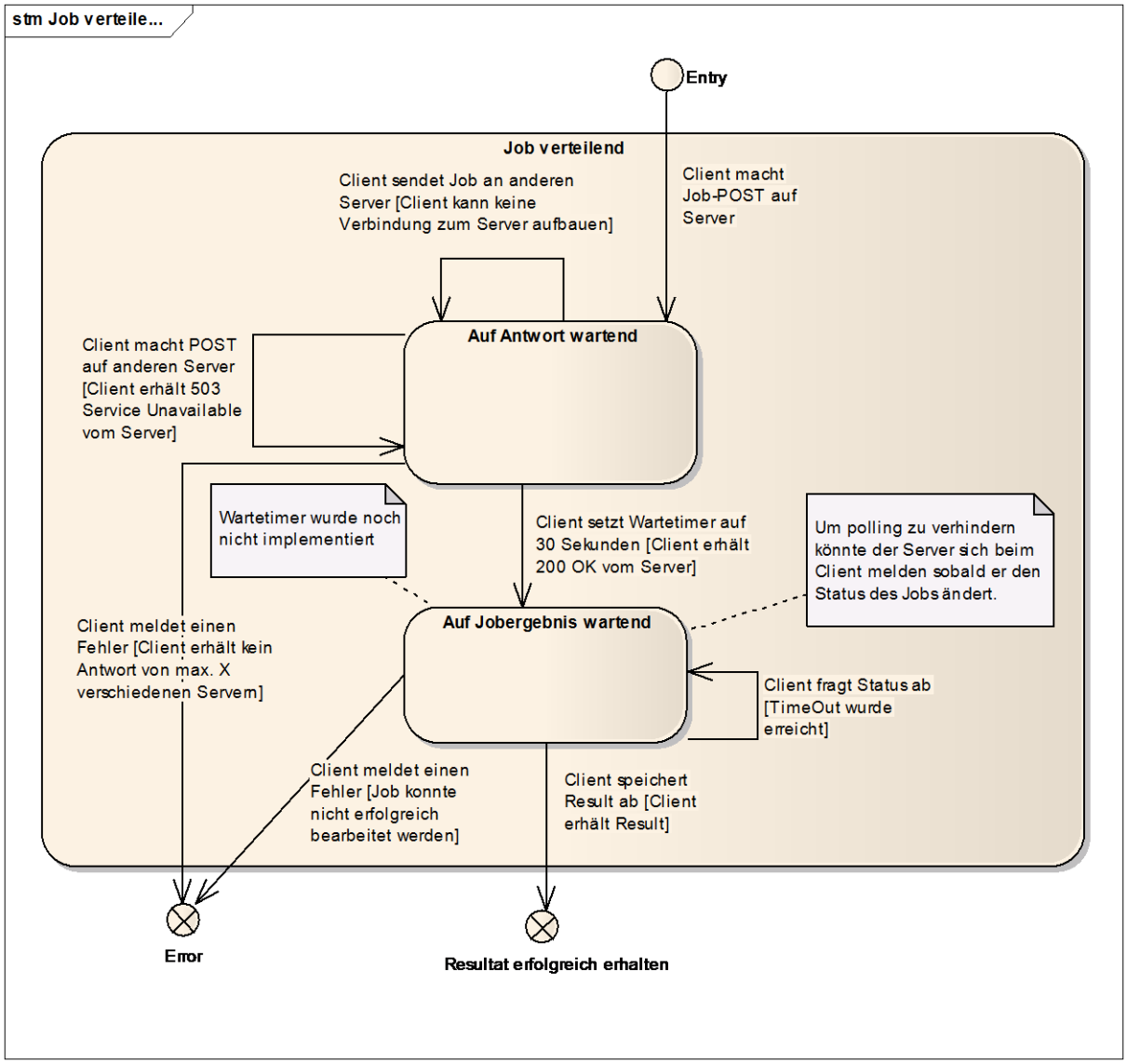

Abbildung 7-10: Ein Node während der Verteilung eines Jobs

7.7.2 NodeServer

Im nächsten Diagramm wird ein Node in der Rolle des Servers dargestellt. Das Diagramm zeigt auf was ein Node tun muss wenn er einen `Job`, eine GET-Anfrage oder eine DELETE-Anfrage erhält. Des Weiteren sind die verschiedenen Verhalten im Falle eines Fehlers ersichtlich.

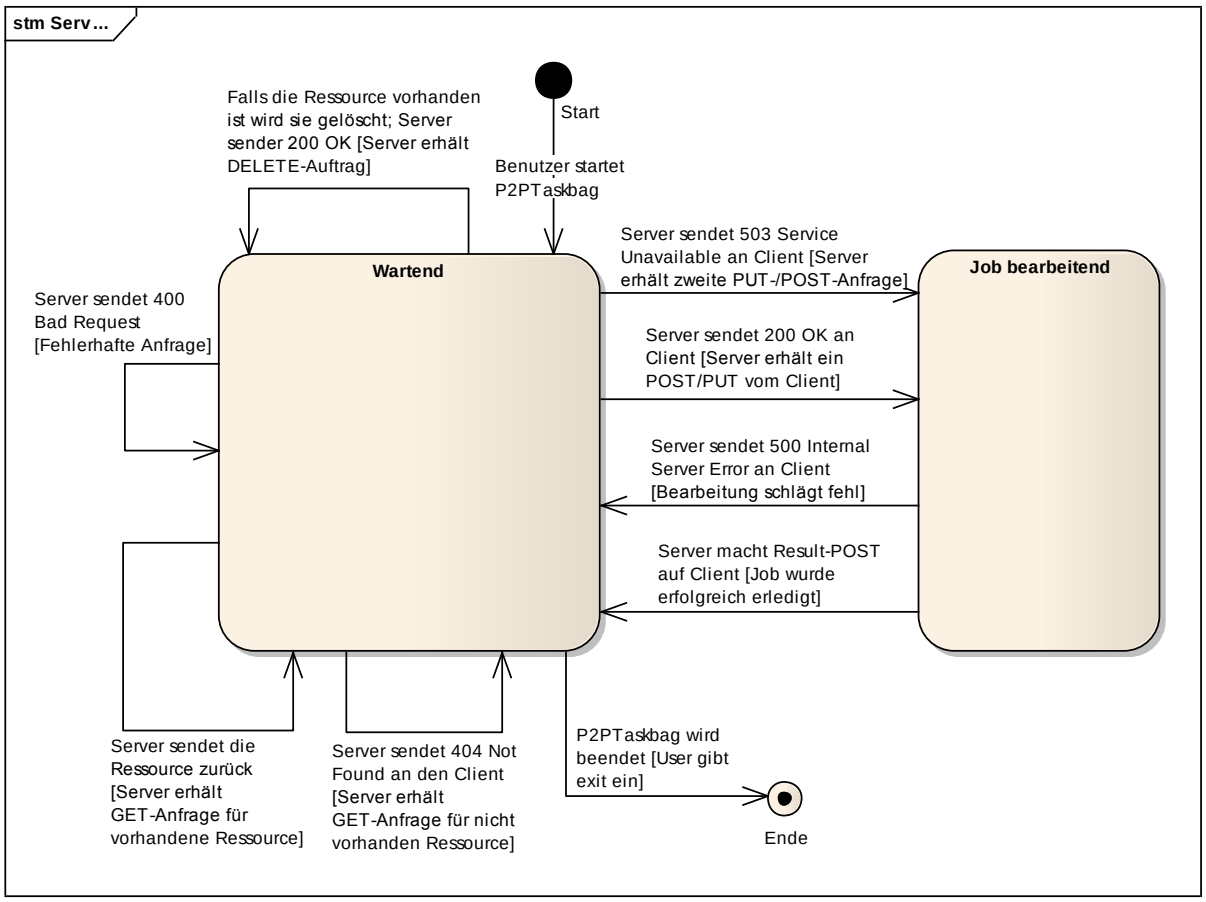

Abbildung 7-11: Ein Node in der Rolle als NodeServer

8 Fehlertoleranz

8.1 Redundanz

Redundanz ist für ein P2P-System von entscheidender Bedeutung. Damit nach dem Austritt oder Ausfall eines Nodes die bei ihm gespeicherten Ressourcen weiterhin zur Verfügung stehen muss zwingendermassen Redundanz eingeführt werden. Bei jedem Request speichert der `Client` alle Ressourcen, die er bezieht, bei sich ab. Diese werden automatisch beim zentralen `Server` registriert. Die Ressource ist nun redundant im P2P-Netz vorhanden. Auf eine automatische Erstellung von Redundanz, also der automatischen Verteilung der Ressourcen, wurde bewusst verzichtet. Redundanz wird also nur erzeugt wenn sich verschiedene Nodes für ein und dieselbe Ressource interessieren.

8.2 Ein- und Austritt eines Nodes

Der Ein- und Austritt eines Nodes in bzw. aus dem P2P-Netz ist ein häufig auftretendes Ereignis. Wenn diese Ereignisse sauber erfolgen, können Fehlverhalten wie [8.3.2 Node nicht verfügbar] minimiert werden. Zusätzlich trägt es zu einer besseren Verfügbarkeit der Ressourcen bei. Um dies zu erreichen hat ein Node in beiden Fällen gewisse Aufgaben die er erledigen muss.

8.2.1 Eintritt

- Sofern bereits eine XML-Datei mit den Ressourcen des `Client`s erstellt wurde, so wird diese ausgelesen und alle Ressourcen auf dem zentralen `Server` veröffentlichen.
- Der Node registriert sich beim zentralen `Server` und schickt ihm seine `NodInfo`s.

8.2.2 Austritt

- Alle verwendeten Ressourcen werden in eine entsprechende XML-Datei abgespeichert.
- Der Node meldet sich beim zentralen `Server` ab. Er gibt also den Löschbefehl für alle auf dem zentralen `Server` über ihn gespeicherten Daten. Alle Verweise über den abmeldenden Node werden somit auf dem zentralen `Server` gelöscht und die Gefahr, dass der Server nicht verfügbare Ressourcen gespeichert, hat wird stark minimiert.

8.3 Fehlerszenarios

Bei jedem System muss prinzipiell mit einem Fehlverhalten gerechnet werden. P2P-Systeme bilden hier keine Ausnahme. Im Gegenteil, sie sind oftmals fehleranfälliger als andere Systeme. Dies kommt daher, dass sie auf die Funktionalität vieler anderer Technologien angewiesen sind. Besonders nennenswert sind hier die Netzwerkverbindung und die einzelnen Nodes. Diese beiden

Komponenten sind öfters für Fehlverhalten zuständig. So kann beispielsweise die Netzwerkverbindung eines Nodes unterbrochen sein, versendete Pakete von einem Router weggeworfen werden oder ein Node bewusst abgeschaltet worden sein. Es gilt nun die Fehler so gut wie möglich zu identifizieren und in den folgenden Fehlerszenarios zu beschreiben. Es sollte dabei auf eine möglichst vollständige Fehlerabdeckung geachtet werden.

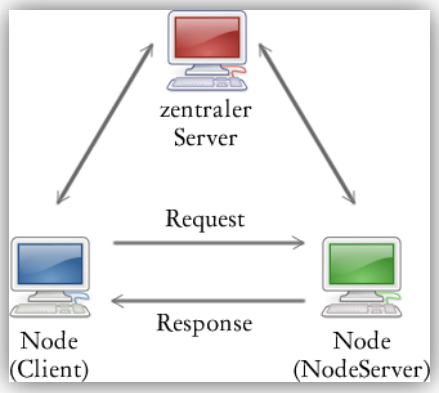

Abbildung 8-1: Grundarchitektur

In einem P2P-Netz gibt es prinzipiell keine Unterscheidung der Nodes. Für mehr Informationen siehe [5.3 Peer-To-Peer]. Ein Node kann sowohl als Server wie auch als Client fungieren nimmt allerdings je nach Situation eine andere Rolle ein. Sobald er einen Request versendet nimmt er die Rolle eines Client ein. Der Empfängernode hingegen nimmt die Rolle des Servers ein und antwortet dem Client mit einer Response. Siehe [Abbildung 8-1: Grundarchitektur]. In diesem Kapitel wird daher oft von einer Kommunikation zwischen einem Server und Client gesprochen. Dabei wird jeweils auf einen Node, der gerade die Rolle eines Server bzw. Client einnimmt verwiesen. Manchmal findet jedoch eine Kommunikation zwischen einem Node und dem effektiven `Server` statt. In diesen Fällen wird explizit von einem zentralen `Server` gesprochen.

8.3.1 Verlust einer Nachricht

8.3.1.1 Problem

Bei der Kommunikation über eine Netzwerkverbindung ist der Verlust oder die Verfälschung einer Nachricht nicht auszuschliessen. Diese Eigenschaft könnte zu Problemen bei der Kommunikation bzw. der Verarbeitung einer Nachricht führen.

8.3.1.2 Vorgehen

Um das Problem des Nachrichtenverlusts zu umgehen, erfolgt sämtliche Kommunikation, die über das Netzwerk stattfinden, über TCP. Mit dem Einsatz von TCP wird eine fehlerfreie Kommunikation zwischen den Nodes gewährleistet.

8.3.2 Node nicht verfügbar

8.3.2.1 Problem

Die Nichtverfügbarkeit eines Nodes ist eines der wohl am meisten auftretenden Probleme. Dies kann durch das gezielte Austreten, sowie einem ungewollten Ausfall verursacht werden. Es muss zu jeder Zeit mit dem Ausfall eines Nodes oder gar dem Ausfall eines grösseren Teils des Systems gerechnet werden. Diese Fehlerquelle ist an die grundlegende Struktur des P2P-Netzes gebunden und lässt sich kaum verhindern. Beim Ausfall eines Nodes werden seine Ressourcen nicht automatisch beim zentralen `Server` entfernt. Solange die Ressourcen nicht gelöscht wurden, verweist der zentrale `Server` weiterhin auf den ausgefallenen Node. Dadurch können unnötige Wartezeiten und Netzwerkbelastungen entstehen.

8.3.2.2 Vorgehen

Eine Abfrage auf den zentralen `Server` liefert eine URI, welche auf einen nicht mehr verfügbaren Node verweist, zurück. Beim vergeblichen Versuch einen Request an diesen Node zu schicken kann keine Verbindung aufgebaut werden. Der Client meldet dem Garbage-Collector die IP des nicht verfügbaren Nodes. Die Aufgabe des Garbage-Collector ist es nun zu prüfen, ob der entsprechende Node verfügbar ist. Ist dieser nicht mehr verfügbar, so beginnt der Garbage-Collector alle Einträge, welche auf diesen Node verweisen, vom zentralen `Server` zu löschen.

8.3.3 Der JobExecuter ist beschäftigt

8.3.3.1 Problem

Der Client sendet einen `Job` an einen Server. Der `JobExecuter` des Servers ist jedoch bereits mit der Abarbeitung eines anderen `Job`s beschäftigt bzw. die Bearbeitungs-Warteschlange ist voll. Daher nimmt der `JobExecuter` keine weiteren `Job`s entgegen.

8.3.3.2 Vorgehen

Server:

Der `JobExecutor` des Servers ist mit der Abarbeitung eines anderen `Job`s beschäftig und sendet den HTTP-Statuscode „503 Service Unavailable“ dem Client zurück.

Client:

Für jeden `Job` ist eine Liste von potenziellen Servern verfügbar. Der `Job` wird nun der Reihe nach an die potenziellen Nodes versendet bis einer den HTTP-Statuscode „200 OK“ zurück sendet. Erhält er von keinem einzigen Node den HTTP-Statuscode „200 OK“ so wird eine Warnung mit dem entsprechenden Text ausgegeben.

8.3.4 Client erhält JobResultat nicht

8.3.4.1 Problem

Ein `Job` wurde vom Client verschickt und die Ankunft vom Server bestätigt („200 OK“). Obwohl der `Job` erfolgreich verteilt wurde, erhält der Client nie ein `Result` des Servers.

8.3.4.2 Vorgehen

Gründe für das Fehlverhalten könnten Fehler während der Kompilation oder der Ausführung des `Job`s sein, sowie das Beenden des Nodes während der Bearbeitung oder der Unterbrunch der Netzwerkverbindung. Tritt ein Fehler bei der Kompilation oder während der Ausführung auf so wird der Status des `Job`s dementsprechend gesetzt.

Der Client hat nun die Möglichkeit den Status eines `Task`s und somit auch die Status der einzelnen `Job`s abzufragen. Aufgrund der dargestellten Informationen kann der Benutzer Schlussfolgerungen über den Verbleib der `Results` treffen.

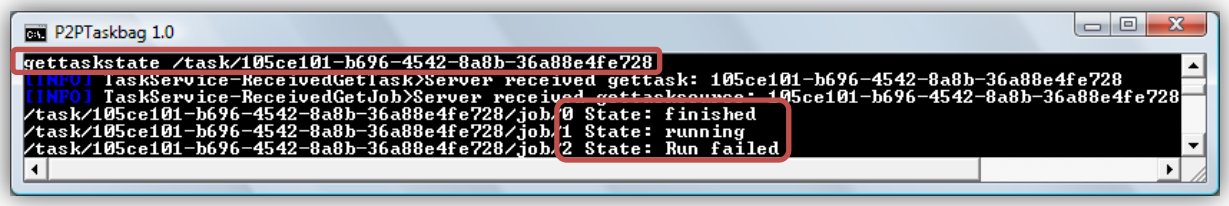

Abbildung 8-2: Abfrage des Taskstatus

Eine Automatisierung dieses Prozesses sollte ohne grosse Probleme implementiert werden können. Der Server müsste hierfür lediglich eine, dem Fehler entsprechende, Nachricht dem Client schicken. Diese Nachricht könnte beispielsweise in Form des HTTP-Statuscode „500 Internal Server Error“ erfolgen. Eine manuelle Abfrage des Status wäre dadurch überflüssig. In einem weiteren Schritte könnte der `Job` automatisch an einen anderen Node gesendet werden. Dieses Verhalten wurde aus zeitlichen Gründen noch nicht umgesetzt.

8.3.5 Ressource ist nicht verfügbar

8.3.5.1 Problem

Der Client erhält vom zentralen `Server` einen URI, welcher auf eine nicht mehr verfügbare Ressource verweist. Ein Request an den entsprechenden Node liefert den HTTP-Statuscode „404 Not Found“ zurück. Dieses Problem tritt dann auf wenn ein Node eine Ressource löscht aber der zentrale `Server` nie darüber informiert wird.

8.3.5.2 Vorgehen

Erhält ein Node einen Request auf eine nicht verfügbare Ressource, meldet dieser den HTTP-Statuscode „404 Not Found“ zurück. Der Empfänger des Statuscodes wird versuchen den Request an einen anderen Node im Netz zu schicken. Bevor der nächste Request allerdings erfolgen kann teilt der Node das Fehlverhalten dem Garbage-Collector-Node mit. Dieser ist nun für die Überprüfung der Ressource sowie einer allfälligen Löschung des Verweises auf dem zentralen `Server` zuständig.

8.4 Enterprise Library 5.0

Bei dieser Library handelt es sich um eine Sammlung von Softwarekomponenten (Application Blocks), welche bei der Entwicklung einer Software bei diversen Problemen und Aufgaben wie z.B. Logging, Validation, Caching oder Exception Handling unterstützend verwendet werden können. Sie wird mit dem Source Code, Testfällen und einer Dokumentation zur Verfügung gestellt. Die Library wurde von uns etwas genauer untersucht und für gut und hilfreich befunden. Trotzdem entschieden wir uns gegen einen Einsatz dieser Library, da sie für unsere Einsatzzwecke zu umfangreich gewesen wäre. Es war für uns einfacher die gewünschten Funktionen auf eine simple Art selber zu implementieren und dadurch Overengineering zu verhindern. [15]

9 Tests zur Sicherstellung der Grundfunktionalitäten

In verteilten Systemen sind Unit Tests nur mit grossem Aufwand realisierbar. Um sinnvolle Tests durchführen zu können, müssten Kommunikationspartner simuliert und mit zusätzlichem Fehlverhalten implementiert werden. Als weiterer Nachteil für das Testing eines verteilten Systems ist die Vielzahl an Interaktionen, Fehlerfälle, Konfigurationen, verwendete Plattformen und sonstige Konstellationen. Schlägt beispielsweise der Verbindungsaufbau zu einem Node fehl, so ist dies nicht zwingend ein Fehler in der Applikation. Dies könnte auf einen Fehler hinweisen, aber es könnte auch das korrekte Verhalten des Systems sein.

Aus diesem Grund werden nur die Grundfunktionalitäten des Systems getestet. Dazu zählen die Zusammenarbeit mit dem zentralen Server und die Kommunikation mit anderen Nodes.

Die implementierten Tests sind, wie bereits gesagt, nicht in gewohnter Form von Unit Test anzutreffen. Sie stellen viel mehr eine Art Hilfefunktion dar. So können die Tests nebst den Entwicklern auch von den Benutzern zur Entdeckung der Fehlerursache verwendet werden.

9.1 Tests für zentralen Server

Der zentrale Server stellt eine sehr wichtige Komponente dar. Jeder Node ist auf die korrekte Arbeitsweise vom Server angewiesen. Folgende Funktionen werden getestet:

- Erreichbarkeit des Server
- Ressource-Speicherung auf dem Server
- Ressource-Anfrage an den Server
- Ressource-Löschung auf dem Server

9.2 Test für Node-Kommunikation

In einem P2P-Netz ist die Kommunikation zwischen den Nodes etwas vom Zentralsten. Im Test werden alle Nodes, welche sich beim zentralen Server registriert haben, der Reihe nach auf dessen Erreichbarkeit getestet. So ist schnell ersichtlich ob sich der testende Node im Netzwerk befindet und mit dessen Teilnehmer kommunizieren kann.

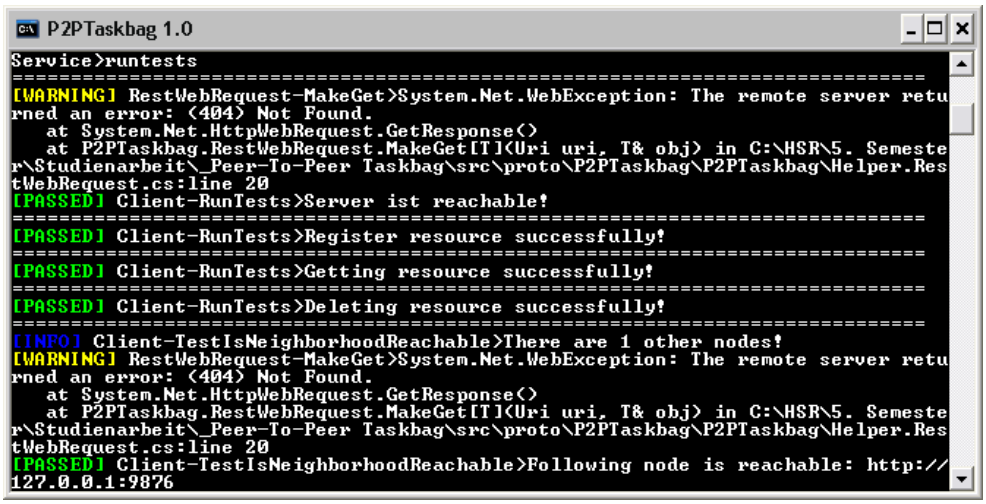

Abbildung 9-1: Tests für die Grundfunktionalitäten

10 Erweiterungsmöglichkeiten

10.1 Verbesserung des zentralen Servers

Da die verwendete Library „NChord“ nicht wie erhofft funktionierte, wurde auf die Verwendung einer DHT verzichtet. Stattdessen wurde ein zentraler `Server` implementiert. Im Moment werden beim Beenden des `Server`s zwar alle Ressourcen gespeichert und ermöglichen somit beispielsweise Wartungsarbeiten durchzuführen, ohne den Verlust der bereits registrierten Ressourcen. Der `Server` stellt aber trotzdem noch einen „Single-Point-Of-Failure“ dar. Nun gibt es zwei Erweiterungsmöglichkeiten, um die Verfügbarkeit und Skalierbarkeit des Systems hochzuhalten.

10.1.1 Einsatz einer DHT

Der Einsatz einer DHT ist grundsätzlich keine schlechte Idee und würde die Verfügbarkeit und Skalierbarkeit des Systems erhöhen. Das Ersetzen des zentralen `Server`s durch eine DHT wäre also eine durchaus realistische und sinnvolle Erweiterung. Mehr Informationen zum Einsatz einer DHT können im Kapitel [5.4 DHT] gefunden werden.

10.1.2 Einsatz eines skalierbaren Servers

Eine alternative Erweiterung zur DHT wäre der Einsatz eines skalierbaren `Server`s. Auch dieser würde die Verfügbarkeit und Skalierbarkeit des Systems erhöhen. Hierzu müssten mehrere `Server` im P2P-Netz verfügbar sein und die Ressourcen gemeinsam von ihnen verwaltet werden. Vorstellbar wäre, dass jeder dieser `Server` für einen gewissen Schlüssel-Bereich verantwortlich wäre. So wäre zum Beispiel bei zwei `Server` der erste für den Bereich mit den Schlüsseln, die von A-L gehen, zuständig und der Zweite für den Bereich von M-Z. Die Anzahl resp. die Grösser der Bereiche wird natürlich dynamisch anhand der Anzahl der vorhanden `Server`n angepasst. Somit hätte man einen skalierbaren `Server`.

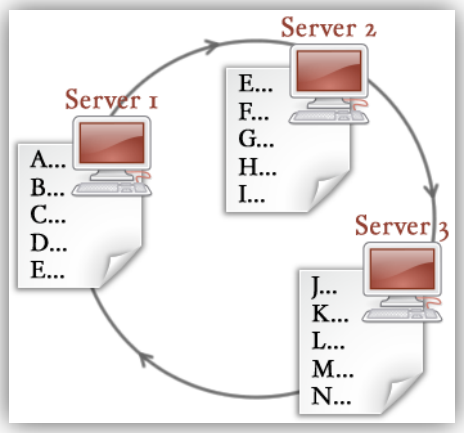

Abbildung 10-1: Skalierbarer Server

10.2 Security

Sicherheit ist ein sehr wichtiges Thema in der heutigen Zeit und stellt bei der Kommunikation über ein Netzwerk einen enorm wichtigen Aspekt dar. Im P2P-Taskbag wurde allerdings keine Form der Sicherheit implementiert. Die Sicherheit könnte und sollte zu einem späteren Zeitpunkt in den folgenden zwei Bereichen umgesetzt werde.

10.2.1 Authentifizierung

Damit nicht Unbefugte irgendwelche Tasks in das Netzwerk einbringen oder dessen Status abfragen können, wäre ein Login zur Authentifizierung des Auftraggebers vorstellbar. Somit wäre die Authentifizierung der Benutzer sichergestellt. Dazu wären QoS-Anforderungen umsetzbar. Die Gefahr, dass schädliche Software über das Netzwerk verteilt werden könnte, wird dadurch massiv reduziert.

10.2.2 Verschlüsselte Kommunikation

Um genügend grosse Vertraulichkeit und Privatsphäre garantieren zu können, wäre das Verschlüsseln der Kommunikation ein erster Schritt. Bevor eine eigene Implementierung umgesetzt wird, muss abgeklärt werden, ob nicht bereits das WCF von Microsoft Verschlüsselungsmöglichkeiten bietet.

10.3 Unterstützung von „multiple source download“ und „swarming“

Es ist damit zu rechnen, dass einige Tasks grosse `ResultFiles` von bis zu mehreren GigaBytes erstellen. Diese sollten trotz ihrer Grösse effizient im Netz verteilt werden können. In einem P2P-Netz ist es möglich die Ressourcen der einzelnen Nodes zu summieren. Ein einzelner Node hat tendenziell eher eine kleine Upload-Geschwindigkeit. Verwendet man aber gleichzeitig mehrere solche Verbindungen ergibt sich fast eine nach oben unbegrenzt grosse Geschwindigkeit je nach Anzahl der Nodes im Netz. Um diese Eigenschaft auszunutzen, ist es nötig, dass ein `ResultFile` (resp. Teile davon), von mehreren Nodes gleichzeitig heruntergeladen werden kann. Diese Möglichkeit ist auch unter dem Begriff „swarming“ bekannt. Sucht sich ein Node nach einem unerwarteten Download-Abbruch eigenständig eine nächste Quelle für den Download, so spricht man von „multiple source download“. Dieses Feature sollte auch unterstützt werden. [16]

10.4 Maintenance GUI

Das ganze System resp. Netzwerk soll auch zur Laufzeit noch beinflussbar sein. So sollen zum Beispiel ungünstig gewählte Timer neu gesetzt werden können. Oder irgendwelche Ressourcen manuell gelöscht werden können.

Ziel des Maintenance GUI ist es, das Netzwerk zu steuern und bei Bedarf neu zu konfigurieren, ohne dass das ganze Netzwerk neu aufgebaut resp. neugestartet werden muss. Auch irgendwelche gravierende Fehler könnten so ohne grossen Aufruhr behoben werden. Natürlich ist das GUI den Netzwerk-Administratoren vorbehalten.

Vorstellbar wäre das GUI mit einem zusätzliches Feature zu erweitern, welches das ganze Netzwerk, dessen Datenverkehr und Ereignisse grafisch darstellen würde. Dieses Feature müsste so umgesetzt werden, dass es die Wirkung als „Eye-Catcher“ besitzen würde.

10.5 Anbindung an diverse Online-APIs

Wie bereits beschrieben, werden die ADS vom NTFS genutzt um Metadaten abzuspeichern. Damit die Daten nicht manuell erfasst werden müssen, sollen Online-Dienste in Anspruch genommen werden. Diese sollen unterstützend wirken indem sie sinnvolle Vorschläge dem Benutzer liefern. Folgend sind einige mögliche Dienste aufgelistet.

- http://www.allcdcovers.com
- http://www.deanclatworthy.com/imdb/
- http://www.lastfm.de/api/rest
- http://developer.yahoo.com/music/

11 Performance Test

Dieses Kapitel soll die Performanz und Skalierbarkeit des Systems aufzeigen. Zu Beginn wird der Task beschrieben, welcher zur Performance-Messung verwendet wurde. Anschliessend wird die Testumgebung resp. Testhardware aufgezeigt. Zum Schluss werden die Messresultate und Erkenntnisse aufgelistet.

11.1 Performance-Task

Der Performance-Task verfolgt ein ähnliches Ziel wie der bereits beschriebene „Password Security Checker“. Es soll ebenfalls ein gegebenes Passwort überprüft werden. Dabei werden aber nicht bruteforce-mässig Vergleichspasswörter generiert. Stattdessen wird das zu prüfende Passwort in einem Wörterbuch gesucht. Hinzukommt, dass das gesuchte Passwort in Form eines MD5-Hashs vorliegt. Daher werden zuerst die Hashs der Wörter vom Wörterbuch generiert und anschliessend mit dem gegebenen Passwort-Hash verglichen. Das Wörterbuch umfasst ca. 400‘000 Wörter. Dieses wird gleichmässig auf die einzelnen Nodes verteilt und anschliessend als `Job` verarbeitet.

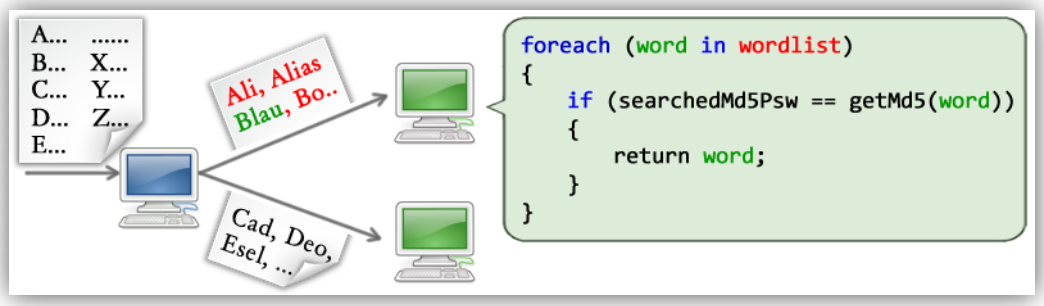

Abbildung 11-1: Performance-Test Aufbau

11.2 Testumgebung

Für die Testdurchführung standen uns vier Test-Rechner zur Verfügung. Zwei davon waren Übungsrechner der HSR und die anderen beiden Rechner unsere privaten Laptops. Leider konnten wir auf Grund ungenügender Rechte auf den HSR Rechner nicht die benötigte .Net Version installieren. Darum verwendeten wir jeweils ein entsprechendes VMware-Image mit dem Betriebssystem Windows XP Professional.

	HSR Image	Martins-PC	Philippes-PC
OS:	Win. XP Pro. SP3	Win. Vista Premium SP2	Win. XP Pro. SP3
CPU	2.7 GHz	Intel Core Duo 1.87 GHz	Intel Core Duo 2.5 GHz
RAM	2 GB	3 GB	2 GB

Tabelle 11-1: Testumgebung

11.3 Messresultate

Wie aus der obenstehenden Tabelle hervorgeht, waren drei unterschiedliche PC-Typen im Einsatz. Daher sind die Test nur beschränkt repräsentativ. Um dennoch sinnvolle Ergebnisse aufzeigen zu können, wurden gewisse Testszenarios mehrmals aber jeweils auf unterschiedlichen Rechner durchgeführt. Dadurch ist man in der Lage Mittelwerde abzuschätzen.

11.3.1 Mit einem Rechner

Da unterschiedliche Rechner für die Tests verwendet worden sind, wurde dieses Testszenario auf allen drei unterschiedlichen Rechnertypen durchgeführt. Dabei entsprechen die Farben den jeweiligen Rechnertypen aus der [Tabelle 11-1: Testumgebung].

Ort des Wortes in der Wortliste	Erstes Ergebnis nach x Sek.	Letztes Ergebnis nach x Sek.
Am Anfang	1.1, 2.4, 1.3	35.0, 177.8, 56.9
In der Mitte	16.7, 88.1, 27.5	31.3, 177.8, 56.9
Am Schluss	33.0, 177.8, 56.6	33.0, 177.8, 56.9

Tabelle 11-2: Performance-Test mit einem Rechner

11.3.2 Mit zwei Rechner

Dieses Szenario wurde mit den beiden HSR-Rechnern durchgeführt, da diese identische Hardware besitzen.

Ort des gesuchten Wortes in der Wortliste	Erstes Ergebnis	Letztes Ergebnis
Am Anfang	nach 2.1 Sekunden	nach 33.1 Sekunden
In der Mitte	nach 11.9 Sekunden	nach 25.8 Sekunden
Am Schluss	nach 21.1 Sekunden	nach 21.1 Sekunden

Tabelle 11-3: Performance-Test mit zwei Rechner

11.3.3 Mit drei Rechner

Martins Laptop wurde bei diesem Szenario nicht mit eingebunden.

Ort des gesuchten Wortes in der Wortliste	Erstes Ergebnis	Letztes Ergebnis
Am Anfang	nach ca. 4.2 Sekunden	nach ca. 21.1 Sekunden
In der Mitte	nach ca. 10.4 Sekunden	nach ca. 20.1 Sekunden
Am Schluss	nach ca. 13.6 Sekunden	nach ca. 19.4 Sekunden

Tabelle 11-4: Performance-Test mit drei Rechnern

11.3.4 Mit vier Rechner

Ort des gesuchten Wortes in der Wortliste	Erstes Ergebnis	Letztes Ergebnis
Am Anfang	nach ca. 2.1 Sekunden	nach ca. 53.2 Sekunden
In der Mitte	nach ca. 5.1 Sekunden	nach ca. 53.2 Sekunden
Am Schluss	nach ca. 10.2 Sekunden	nach ca. 53.2 Sekunden

Tabelle 11-5: Performance-Test mit vier Rechnern

11.3.5 Erkenntnisse

Die erste Ergebnisantwort kam unterschiedlich schnell. Denn wird gerade dem schwächsten PC der `Job` mit dem gesuchten MD5-Hash zugeteilt, so dauert natürlich die benötigte Zeit enorm viel länger als wenn dieser dem schnellsten Rechner zugeteilt worden wäre.

Klar ist auch, dass die Dauer bis zum Eintreffen des letzten Ergebnisses jeweils vom langsamsten Rechner abhängt. Aus diesem Grund sind die Zeitangaben für den Empfang des letzten Ergebnisses pro Test dieselben. Da der Einsatz eines, für einen spezifischen Task, schlechten Rechners das System verlangsamen kann, ist es sinnvoll die einzelnen Nodes zu clustern. So würde zum Beispiel ein CPU-schwacher Rechner gar nie unseren Performance-Task erhalten und würde daher das Netz auch nicht unnötig bremsen.

12 Aufgetretene Probleme

12.1 NChord Implementation

Während der Recherche über DHT stiessen wir auf die C#-Implementation NChord auf http://nchord.sourceforge.net/. Nach längerer Testzeit stellten wir fest, dass der jetzige Stand von NChord nicht wie erhofft arbeitet. Um nicht unnötig Zeit zu verlieren, entschieden wir uns NChord nicht zu verwenden.

13 Entscheide

13.1 Zentraler Server statt DHT

Da wie beschrieben Probleme mit der DHT-Implementation auftraten, waren wir gezwungen eine alternative Lösung zu finden. Wir entschieden uns selber einen zentralen Server zu implementieren. Dies sparte Zeit und ermöglichte uns, unsere Nodes so zu entwickeln, dass sie dynamisch die URLs der Ressourcen ermitteln. Dazu kommt, dass das Ersetzen des Servers durch eine DHT für unsere Nodes keine grosse Auswirkung haben sollte. Schlussendlich wird eine global verfügbare Datenstruktur benötigt. Ob nun diese zentral oder verteilt organisiert ist, interessiert die Nodes in diesem Sinne nicht.

13.2 ADS statt eigener Daten-Container

Zuerst wurde vorgesehen, dass für die von den Jobs erstellten ResultFiles einen eigener Daten-Container umgesetzt würde. Nach weiteren Überlegungen und Diskussionen wurde festgestellt, dass dieser Ansatz den Nachteil hat, dass das originale Format verloren geht. Dies würde bedeuten, dass zum Beispiel eine MP3-Datei nicht direkt abspielbar wäre. Sie müsste stattdessen zuerst mit unserer Software „entpackt“ werden.

Verständlicherweise würde ein Benutzer nach dem Entpacken unsere Container-Datei löschen um so Speicherplatz zu sparen. Dies würde aber bedeutet, dass auch die Metadaten verloren gehen. Dieser Effekt ist natürlich nicht gewünscht und veranlasste uns nach einer anderen Lösung zu suchen, die wir in den ADS gefunden haben.

14 Erfahrungsberichte

14.1 Martin Weber

Unser 5. Semester begann und mit ihm auch der Start unserer Studienarbeit. Leider hatten wir zu Beginn ein paar organisatorische Probleme. So war beispielsweise lange Zeit nicht ganz klar wer unser offizieller Betreuer der Arbeit sein wird. Der Start eines neuen Projektes ist bekanntlich immer etwas mühsam und so hatten auch wir unsere Schwierigkeiten. Wir investierten einige Zeit in die Projektorganisation, die wir nach RUP durchführen wollten. Dieses Vorgehen erwies sich allerdings für unser Projekt als ungeeignet, weswegen wir dieses Vorgehen verwarfen und uns auf andere Dinge konzentrierten. Weiterhin bestanden einige Unklarheiten, technischer Probleme und auch die Abwesenheit von Herrn Joller spielte hier unter anderem eine entscheidende Rolle. Doch auch diese Probleme gingen vorbei und das Projekt kam ins Rollen. Die Ziele und Anforderungen wurden genauer definiert und mit der Programmiersprache C# konnte ich mich gleich anfreunden. Viele hitzige Diskussionen und Programmierstunden später nahm unser P2P-Taskbag langsam Gestalt an. Es war herrlich zu beobachte, wie durch die Eingabe einfacher Konsolenbefehle unsere Computer und diverse Virtuelle Maschinen begannen zu arbeiten und kommunizieren. Natürlich befand sich die Applikation immer noch im Anfangsstadium und es gab noch einiges zu tun. Viele „Kleinigkeiten“ oder Details erwiesen sich oftmals als erhebliche Probleme oder führten zu längeren Diskussionen im Team. Nichts desto trotz wuchs unsere Applikation ständig. Die Arbeit an unserem P2P-Taskbag war von Anfang an spannend und die Motivation für die Weiterarbeit blieb hoch. Ich war immer wieder dafür dankbar, dass unser Projekt ein für uns höchst interessantes Themengebiet behandelt und konnte mir kaum vorstellen wie es wäre an einem, für mich uninteressanten, Thema zu arbeiten. Es stand uns natürlich auch jederzeit offen eine kleine Diskussionsrunde mit Herrn Joller zu arrangieren. Über seine fachlich kompetente und angenehme Betreuung war ich oft dankbar. Er liess uns frei arbeiten aber war bei Problemen stets eine Hilfe.

Langsam aber sicher bewegte sich das Projekt auf das Ende zu. Uns war beiden bewusst, dass, obwohl die Studienarbeit ein Ende nimmt, der P2P-Taskbag noch lange keine abgeschlossene Applikation ist. Da das Interesse an einer Weiterentwicklung sehr hoch ist, hoffe ich auf eine Fortsetzung der Arbeit im nächsten Semester als Bachelorarbeit.

14.2 Philippe Morier

Der Start unserer Studienarbeit war zu Beginn nicht ganz einfach. In den ersten drei Wochen mussten wir ohne Betreuer zu Recht kommen. Wir starteten das Projekt gemäss RUP. Daher schrieben wir viel Dokumente gemäss vorhandenen Vorlagen. Nach dem ersten Meeting mit Herrn Joller war klar, dass das RUP-Model nicht ganz auf unser Projekt passte. Unser Projekt ist kein Kunden-Auftrag Projekt sondern viel mehr ein experimentelles und mit Recherchen verbundenes Projekt. Anhand der Informationen von Herrn Joller verliessen wir das RUP-Model und konzentrierten uns auf ein Gesamtdokument, das den Vorgaben von Herrn Joller entsprechen sollte.

Die Recherche nach einer geeigneten DHT-Implementation war sehr interessant und lehrreich. Auf der anderen Seite mussten wir feststellen, dass das ganze Thema DHT sehr komplex ist. Wir fanden eine fast perfekte C# .Net Implementation mit dem Namen „NChord“. Leider stellte sich nach längeren Tests heraus, dass die gefundene Implementation nicht fehlerfrei arbeitete. Daraufhin entschieden wir uns für die Umsetzung eines zentralen Servers. Das Entwickeln einer eigenen Netzwerkarchitektur und dessen Protokoll empfand ich persönlich als den spannendsten und zugleich anspruchsvollsten Teil des Projektes. Das Anwenden des REST-Prinzips auf ein P2P-Netzwerk ist und bleibt eine sehr interessante und spannende Idee. Durch das WCF von Microsoft hatten wir eine sehr gute Unterstützung für die Umsetzung des REST-Prinzips. Die Möglichkeit eigene Ideen und Vorschläge einzubringen und diese mit Herrn Joller anschliessend zu besprechen empfand ich als sehr motivierend. Ich schätzte es, dass nicht alles von Anfang an klar bis ins letzte Detail definiert war, sondern dass die Entwicklung des Systems dynamisch blieb. Dadurch konnten sich alle miteinbringen und es entstand in Zusammenarbeit eine Lösung. Allgemein verlief die Zusammenarbeit reibungslos. Dadurch dass Martin und ich uns gut verstehen, waren die Diskussionen über mögliche Lösungsvorschläge sehr effizient und direkt. Es wurde nicht lange um den heissen Brei gesprochen sondern Vorschlag um Vorschlag diskutiert. Anschliessend wurde gemeinsam eine Entscheidung getroffen. Dies schätzte ich sehr und zeigte ein weiteres Mal wie wichtig ein funktionierendes Team ist.

Zusammenfassend bin ich sehr zufrieden mit unserer Studienarbeit. Sie war spannend, interessant und lehrreich. Der Verdacht dass verteilte Systeme etwas Interessantes sind, erhärtete sich mit dieser Arbeit. Ich hoffe dass wir unsere Bachelorarbeit als eine Vorsetzung dieser Studienarbeit weiter führen können und würde mich auf diese freuen.

15 Reflektionen

15.1 Gelungene Implementationseigenschaften

15.1.1 Effiziente Umsetzung von REST

Die Umsetzung der REST-Architektur und dessen Protokoll wurden mit Hilfe des von Microsoft zur Verfügung gestellten WCF umgesetzt. Dadurch waren wir in der Lage die ganze Kommunikation im Netzwerk sehr effizient und ohne grossen Aufwand zu realisieren. Das WCF nahm uns dabei sehr viel Arbeit ab.

Es zeigte sich, dass die richtige Wahl der Technik sehr zeitsparend sein kann. Der Wahl der Technik sollte daher auch in Zukunft genügend Aufmerksamkeit geschenkt werden.

15.1.2 Nutzung vorhandenen Standards

In unserem Projekt wurden Standards resp. Prinzipien wie HTTP, TCP, REST und XML verwendet. So wurde explizit Wert darauf gelegt, dass Vorhandenes verwenden wird und nicht das Rad jeweils neu erfunden wurde.

Durch dieses Vorgehen konnten wir uns um die effektiven Probleme kümmern. Auch konnte zusätzlich Zeit gewonnen werden. Daher sollten Standards und bereits vorhandene Techniken beachtet werden und bei passender Problemstellung verwendet resp. eingesetzt werden.

15.2 Unschöne Implementationseigenschaften

15.2.1 Zu starke Konzentration auf DHT-Recherche

Zu Beginn des Projektes wurde viel Zeit in die DHT-Recherche investiert. So konnte zwar viel Wissen über dieses Gebiet aufgebaut werden, jedoch wurde der Gesamtüberblick ein wenig vernachlässigt.

Das nächste Mal soll daher explizit darauf geachtet werden, dass zu Beginn alle Möglichkeiten und Varianten beachtet und abklärt werden. So soll der Überblick über das Gebiet stets erhalten bleiben.

15.2.2 Mächtiger zentraler Server

Nach dem Entscheid, dass keine DHT eingesetzt werden soll, wurde einen zentraler Server implementiert. Um möglichst viel Zeit zu sparen wurde der Server möglichst passend für unsere Bedürfnisse erstellt. D.h. der Server kennt zum Teil die genauen Datentypen, die er in seinen Listen abspeichert.

Wir sind der Meinung, dass der Server dadurch zu mächtig resp. zu viel Logik enthält. Er sollte vielmehr eine neutrale und möglichst einfache Datenstruktur zur Verfügung stellen. Dabei sollte er kein Wissen über die bei ihm abgespeicherten Daten besitzen. Auf diese Eigenschaft soll in Zukunft geachtet werden.

15.2.3 Serialisierbare HashMap-Implementierung

Im Projekt wurde oft mit einer HashMap als Datenstruktur gearbeitet. Leider ist die von .Net zur Verfügung gestellte Implementation nicht auf XML Basis serialisierbar. Da die Daten resp. Ressourcen, die zur Laufzeit angesammelt werden, irgendwann aber abgespeichert werden müssen, benötigten wir eine Datenstruktur, die serialisierbar ist. Nach kurzer Suche im Internet fanden wir eine entsprechende Implementierung mit der Klasse `DictionaryProxy`. Im Laufe der Entwicklung stellten wir das Fehlen von benötigten Funktionalitäten fest. Wir implementierten darauf die gewünschten Features selber. Dabei wurde für jedes spezifische Problem kurzer Hand eine passende Lösung realisiert. Leider wurden nicht viel Gedanken über das Design der stetig wachsenden Datenstruktur gemacht. So wurden einzelne Unschönheiten wie das nach aussen Bekanntmachen der internen Datenhaltung, implementiert.

Wird in Zukunft wieder eine fremde Implementation verwendet, soll diese zu erst richtig untersucht werden. So soll abgeschätzt werden, ob die vorliegende Implementation auch wirklich den gestellten Anforderungen gerecht wird. Dadurch sollen allfällige Erweiterungen zum grössten Teil vermieden werden.

16 Aufbau eines eigenen Netzwerkes

In diesem Kapitel soll beschrieben werden, wie ein neues Netzwerk aufgesetzt und verwendet werden kann.

16.1 Installation des .Net Frameworks von Microsoft

Als erstes muss festgestellt werden, ob das .Net Framework in der Version 4.0 auf dem Rechner installiert ist. Es gibt verschiedene Arten dies herauszufinden. Im Folgenden sind drei Varianten aufgelistet:

- In den Systemsteuerung unter „Software“ resp. „Programm und Funktion“ (Vista/Win7) prüfen welche .Net Version installiert ist.
- In der Windows Registry gemäss folgendem Artikel die Version herauslesen: http://support.microsoft.com/kb/318785/de
- Das Global Assembly Cache-Tool Tool (Gacutil.exe) verwenden: http://msdn.microsoft.com/de-de/library/ex0ss12c%28VS.80%29.aspx

Ist nicht die geforderte Version installiert, kann diese unter folgendem Link heruntergeladen und anschliessend installiert werden:
http://www.microsoft.com/downloads/details.aspx?FamilyID=0a391abd-25c1-4fc0-919f-b21f31ab88b7&displaylang=de

16.2 Firewall & Netzwerkeinstellungen

In der Firewall muss sowohl der Client (P2PTaskbag.exe) als auch der zentrale Server (P2PTaskbagServer.exe) freigegeben werden. Je nachdem ob das System innerhalb eines LANs oder über das Internet verwendet wird, müssen die folgenden Ports weitergeleitet werden:

- P2PTaskbag.exe: 9876 (Default)
- P2PTaskbagServer.exe: 7777 (Default)

16.3 Applikationsstart und Grundfunktionalitäten überprüfen

Als erstes sollte der zentrale Server (P2PTaskbagServer.exe) auf einem ausgewählten Computer gestartet werden. Dieser sollte keine Fehler- oder Warnungs-Hinweise zeigen.

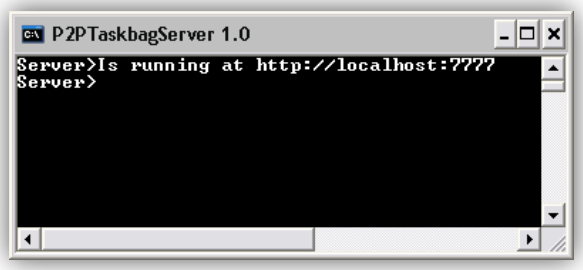

Abbildung 16-1: Erfolgreicher Server-Start

Nach erfolgreichem Serverstart kann nun der erste Client (P2PTaskbag.exe) gestartet werden. Dieser sollte in der Lage sein, sich beim soeben gestarteten Server zu registrieren. Beim Start des Clients kann zusätzlich der Clientport und der URI des Servers als Parameter mitangegeben werden muss. Standardmässig wird für den Clientport der Port „9876" gewählt und der URI des Servers ist auf http://10.10.10.41:7777 gesetzt.

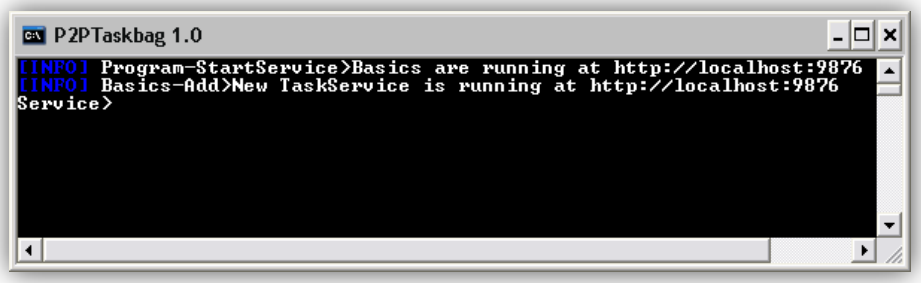

Abbildung 16-2: Erfolgreicher Client-Start

Folgendes Bild zeigt das erfolgreiche Registrieren des Client beim zentralen Server.

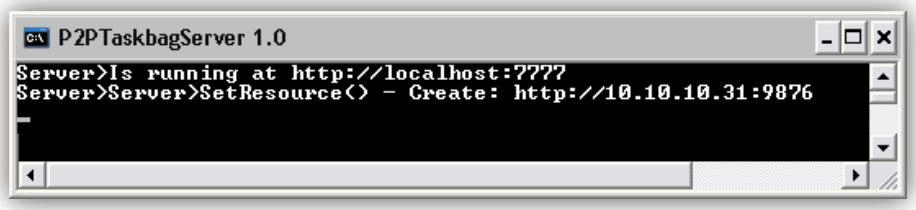

Abbildung 16-3: Erfolgreiche Clientregistration

Nun kann eine beliebige Anzahl andere Clients gestartet werden. Um ein Netz simulieren zu können ohne mehrere Computer zu benötigen, können die einzelnen Clients mit einem anderen Port als Startparameter, ausgeführt werden.

Konnte bis anhin alles erfolgreich gestartet werden, sollten nun die Grundfunktionalitäten getestet werden. Dies lässt sich von einem beliebigen Client aus mit dem Befehl „RunTests“ bewerkstelligen.

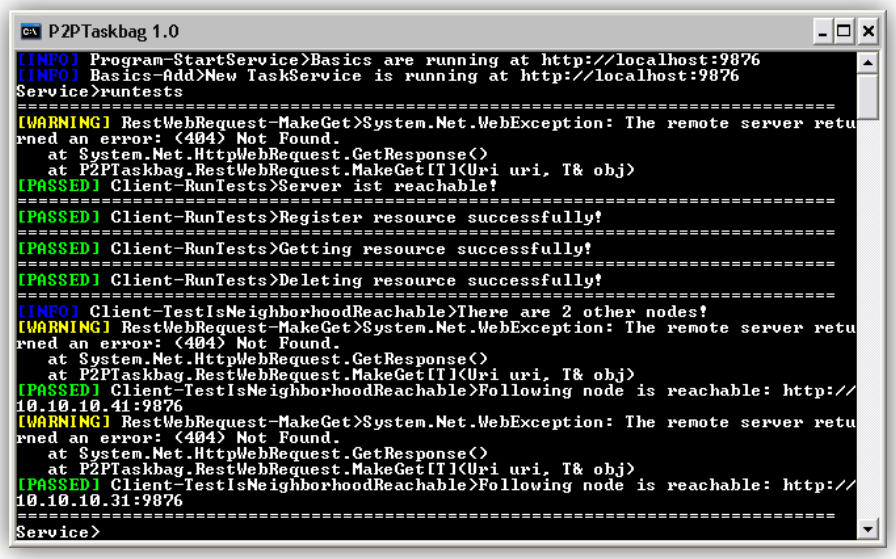

Abbildung 16-4: Grundfunktionalität-Überprüfung

16.4 Applikationseinstellungen manuell bearbeiten

Das Visual Studio 2010 unterstützt das automatische Speichern und Laden von Benutzer-Einstellungen. Diese werden in unserem Projekt verwendet. Eigentlich sieht Visual Studio das manuelle Ändern dieser Einstellungen nicht vor. Die Einstellungen sollten vielmehr durch die Applikation selbst geändert werden. Da aber erst in der Bachelorarbeit die Umsetzung einer grafischen Benutzeroberfläche geplant ist und somit die volle Unterstützung durch das Visual Studio gewährleistet wird, soll hier das manuelle Bearbeiten der Einstellung erläutert werden.

Visual Studio speichert, sobald eine Abweichung zu den Default-Einstellungen von der Datei „P2PTaskbag.exe.config“ vorliegt, pro Benutzer eine Datei mit dem Namen „user.config“ ab. Diese Datei wird unter folgendem Pfad abgelegt:

Windows XP:

- C:\Dokumente und Einstellungen\%UserName%\Lokale Einstellungen\Anwendungsdaten\P2PTaskbag\P2PTaskbag.exe_Url_z3eu...~1\1.0.0.0\user.config

Windows Vista/7:

- C:\Users\%UserName%\AppData\Local\P2PTaskbag\P2PTaskbag.exe_Url_z3eu...~1\1.0.0.0\user.config

Beim Ordner „Anwendungsdaten“ resp. „AppData“ handelt es sich um ein verstecktes Verzeichnis. Daher sollte in den Explorer-Ordneroptionen die Option „Versteckte Dateien und Ordner anzeigen“ aktiviert werden.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type="System.Config
            <section name="P2PTaskbag.Properties.Settings" ty
        </sectionGroup>
    </configSections>
    <userSettings>
        <P2PTaskbag.Properties.Settings>
            <setting name="ClientPort" serializeAs="String">
                <value>9877</value>
            </setting>
            <setting name="ServerUri" serializeAs="String">
                <value>http://10.10.10.31:7777</value>
            </setting>
        </P2PTaskbag.Properties.Settings>
    </userSettings>
</configuration>
```

Abbildung 16-5: Datei: user.config

16.5 Auftrag starten

Nun ist es dem Leser überlassen ob der Beispiel-`Task` oder einen eigenen `Task` verwendet werden soll. Sobald die entsprechenden Dateien (Task.xml & Source.xml) vorhanden sind, können diese mit dem Befehl „CheckTask“ auf Korrektheit überprüfen.

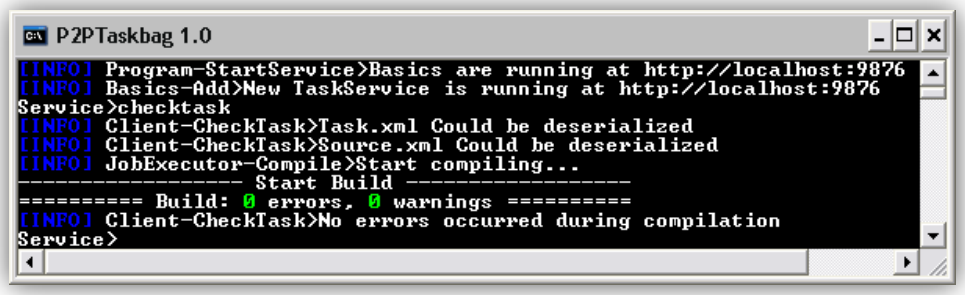

Abbildung 16-6: Task auf Korrektheit überprüfen

Durchläuft der `Task` den Test erfolgreich, kann dieser mit dem Befehl „StartTask“ gestartet werden.

16.6 Netzwerkerweiterung

Sollte das Netzwerk durch das Hinzufügen von zusätzlichen Nodes erweitert werden, können ganz einfach neue Nodes gestartet werden. Diese registrieren sich daraufhin eigenständig beim zentralen Server. Somit sind sie automatisch in das Netz miteingebunden und stehen für die Bearbeitung weiterer `Task`s zur Verfügung.

17 Geleistet Arbeitsstunden

In der unten stehenden Tabelle und Grafik sind die geforderten und die geleisteten Arbeitsstunden des Projektes ersichtlich. Die Stunden von Philippe Morier und Martin Weber werden jeweils zusammengefasst dargestellt. Dies kommt daher, weil prinzipiell nur zusammen gearbeitet wurde und die Differenz daher marginal ist.

Woche	Soll Zeit	Geleistete Stunden
1	34.25	17.5
2	34.25	39.5
3	34.25	38.75
4	34.25	41.5
5	34.25	42.5
6	34.25	40.5
7	34.25	42.5
8	34.25	44
9	34.25	41
10	34.25	42
11	34.25	44.5
12	34.25	33.75
13	34.25	40
14	34.25	20
Total	**480**	**528**

Tabelle 17-1: Geleistet Arbeitsstunden

Das folgende Diagramm zeigt die oben dargestellten Daten grafisch dar.

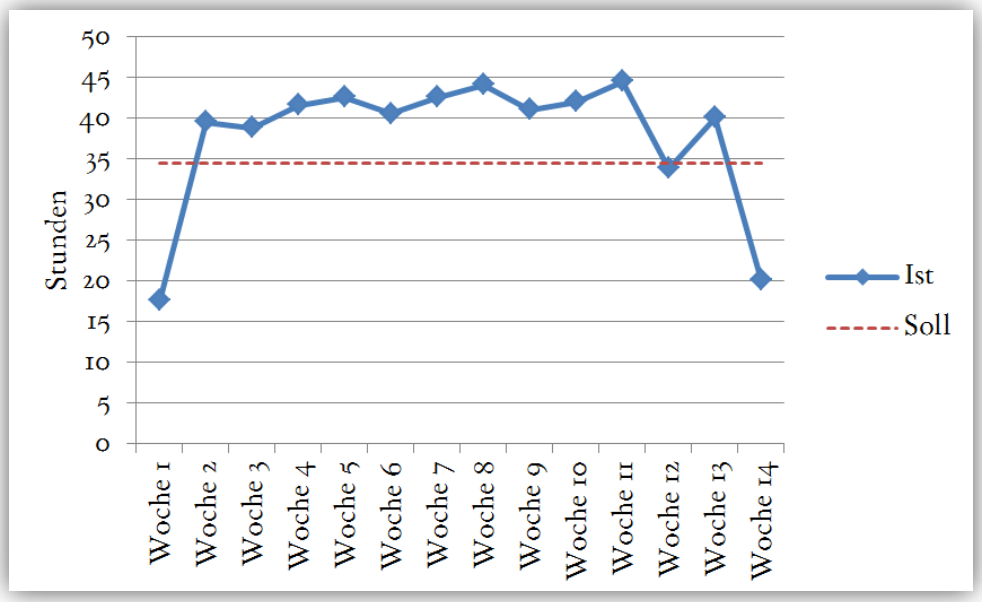

Abbildung 17-1: Geleistete Arbeitsstunden

18 Glossar

HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
P2P	Peer-To-Peer
DHT	Distributed Hash Table (verteilte Hashtabelle), eine Datenstruktur in der Informatik
URI	Uniform Resource Identifier
CSS	Client-Stateless-Server
XML	Extensible Markup Language
WCF	Windows Communication Foundation
IP	Internet Protocol
DynDNS	Ist ein System, das in Echtzeit Domain-Name-Einträge aktualisieren kann. Für das Projekt könnte der Provider www.dyndns.com verwendet werden.
Bootstrapping-Node	Auch bekannt als Rendezvous Point. Stellt der Eintrittspunkt für neuhinzukommende Nodes dar. Neue Nodes finden über ihn den Eingang in das P2P-Netz.
Peer/Node	Ein Endpunkt einer Kommunikation in einem Computernetzwerk. Jeder Peer bietet dabei seine Dienste an und nutzt die Dienste der anderen Peers. Ein Peer kann sowohl Aufträge erteilen wie auch erledigen.
ADS	Alternative Datenströme sind eine Datenstruktur zur Aufnahme strukturierter Daten innerhalb einer Datei neben den unstrukturierten (Haupt-)Daten.
NTFS	New Technology Files System
QoS	Quality of Service
RUP	Rational Unified Process
VMware	Mit VMware lassen sich verschieden Rechner virtualisieren. Somit kann ein kompletter Rechner auf einem Gast-System simuliert werden.

19 Abbildungsverzeichnis

20 Tabellenverzeichnis

21 Recherche

21.1 DHT

Anfangs Oktober 2010 wurden folgende Berichte, die das Thema DHT behandeln, studiert:

- http://cs-www.cs.yale.edu/homes/arvind/cs425/assignments/proj/p133-manku.pdf, 01.10.2010
- http://www.cs.princeton.edu/courses/archive/fall06/cos561/papers/opendht.pdf, 01.10.2010
- http://sarwiki.informatik.hu-berlin.de/DHT, 06.10.2010
- http://dks.sics.se/jdht/, 07.10.2010
- http://portal.acm.org/ft_gateway.cfm?id=872054&type=pdf&CFID=115711760&CFTOKEN=80232821, 07.10.2010
- http://pdos.csail.mit.edu/papers/dhtcomparison:infocom05/paper.pdf, 08.10.2010

Folgende Beiträge über Chord wurden bearbeitet:

- http://www.sigcomm.org/sigcomm2001/p12-stoica.pdf, 01.10.2010
- http://sarwiki.informatik.hu-berlin.de/Chord, 06.10.2010
- http://www.inf.fu-berlin.de/lehre/SS07/VS/folien/vs5.5.pdf, 07.10.2010
- http://www.gaulke.net/werner/arbeiten/Vortrag_Seminar_P2P_Chord.pdf, 08.10.2010

21.2 P2P

Informationen zu P2P wurden an folgenden Stellen nachgelesen:

- http://www.infosys.tuwien.ac.at/staff/sd/papers/DBS-P2P.pdf, 15.10.2010
- http://de.wikipedia.org/wiki/Peer-to-Peer, 18.10.2010
- http://en.wikipedia.org/wiki/Peer-to-peer, 18.10.2010

21.3 REST

Um REST zu vertiefen wurden folgende Beiträge bezogen:

- http://de.wikipedia.org/wiki/Representational_State_Transfer, 12.10.2010
- http://en.wikipedia.org/wiki/Representational_State_Transfer, 12.10.2010
- http://www.oio.de/public/xml/rest-webservices.htm, 12.10.2010
- http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, 13.10.2010
- http://www.jugs.ch/html/events/slides/090929_RestJugs.pdf, 14.10.2010
- http://www.xfront.com/REST.ppt, 15.10.2010

22 Referenzen

[1] http://www.oio.de/public/xml/rest-webservices.htm, 12.10.2010
[2] http://de.wikipedia.org/wiki/HTTP-Statuscode, 13.10.2010
[3] http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, 13.10.2010
[4] Von Josef Joller erhaltene Präsentation „090929_RestJugs.pdf“, 19.10.2010
[5] http://en.wikipedia.org/wiki/Representational_State_Transfer, 12.10.2010
[6] http://www.oio.de/public/xml/rest-webservices.htm, 12.10.2010
[7] http://www.infosys.tuwien.ac.at/staff/sd/papers/DBS-P2P.pdf, 15.10.2010
[8] http://www.gaulke.net/werner/arbeiten/Vortrag_Seminar_P2P_Chord.pdf, 08.10.2010
[9] http://cs-www.cs.yale.edu/homes/arvind/cs425/assignments/proj/p133-manku.pdf
[10] http://sarwiki.informatik.hu-berlin.de/Chord, 06.10.2010
[11] http://www.inf.fu-berlin.de/lehre/SS07/VS/folien/vs5.5.pdf, 07.10.2010
[12] http://www.gaulke.net/werner/arbeiten/Vortrag_Seminar_P2P_Chord.pdf, 08.10.2010
[13] http://www.flexhex.com/docs/articles/alternate-streams.phtml, 17.11.2010
[14] http://www.codeproject.com/KB/cs/ntfsstreams.aspx, 17.11.2010
[15] http://it-republik.de/dotnet/news/Enterprise-Library-5.0-ist-da-055049.html, 18.11.2010
[16] http://www.filesharingzone.de/download-techniken.php, 11.11.2010