Bachelorarbeit „Timber!“

Philippe Morier & Martin Weber

Betreuer: Prof. Dr. Josef Joller

Gegenleser: Prof. Dr. Markus Stolze

Experte: Matthias Lips

HSR Hochschule für Technik Rapperswil

Frühlingssemester 2011

Erklärung

Wir, Philippe Morier und Martin Weber erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, 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 haben.

Rapperswil, 17. Juni 2011

Philippe Morier

Martin Weber

Inhalt

1 Einführung

Die vorliegende Bachelorarbeit behandelt die Realisierung einer eigener Distributed Hash Table (DHT). Viele der bereits existierenden Lösungen verwenden eine ringförmige Netzstruktur. Um in einer ringförmigen Netzstruktur eine Routing-Komplexität von O(log n) zu erreichen, sind weitere nicht triviale Konzepte notwendig. Die Herausforderung dieser Bachelorarbeit besteht darin, eine DHT zu entwickeln, welche ohne zusätzlichen Aufwand eine Routing-Komplexität von O(log n) aufweist.

Die Struktur eines binären Suchbaumes (BST) erfüllt grundsätzlich eine Routing-Komplexität von O(log n). Daher wurde versucht die gestellte Anforderung mittels Einsatz einer Baumstruktur zu erfüllen. Der beschriebene Ansatz wurde weiterverfolgt und zu einem Konzept ausgearbeitet. Das Konzept wurde umgesetzt und am Ende der Bachelorarbeit entstand eine funktionsfähige DHT.

Dieses Dokument beschreibt zu Beginn das ausgearbeitete Konzept für die Erstellung der geforderten DHT. Weitere mögliche Konzepte und Ideen wurden angedacht und festgehalten. Im Anschluss wird die Architektur der entwickelten DHT erläutert. Sequenzdiagramme zeigen in einem weiteren Schritt das Zusammenspiel der einzelnen Klassen. Dabei soll die Beschreibung der Architektur das Weiterführen des Projektes erleichtern.

Gegen das Ende der Dokumentation werden die aufgetretenen Probleme und ein Tutorial aufgeführt. Das Tutorial beschreibt die Einbindung der Bibliothek sowie die Verwendung der API.

2 Vision & Motivation

In der Studienarbeit wollten wir eine einfach einsetzbare DHT verwenden. Leider liess sich keine geeignete Implementation finden. Worauf wir unsere Studienarbeit ohne DHT realisierten. Das Interesse eine DHT einzusetzen wurde dadurch noch verstärkt und wir entschieden uns, unsere Eigene zu implementieren.

Bei den gefundenen Implementierungen waren wir über die komplexe Anwendung überrascht. Wir vermissten eine DLL, welche einfach in das eigene Projekt eingebunden werden kann. Diese sollte nur gerade die wichtigsten Methoden einer DHT anbieten. Zusätzlich sollten Anforderungen wie Redundanz, Lastverteilung und Ausfallsicherheit mit dem Einsatz der DHT abgedeckt werden. Unsere Vision für diese Bachelorarbeit ist es, eine solche Komponente zu realisieren.

```
TimberService timber = new TimberService();

timber.Start();
timber.SetMinClones(3);

timber.Add("Key", "Value");
timber.Get("Key");
timber.Remove("Key");

timber.Stop();
```

Tabelle 2-1: Code - Vision

Das oben aufgeführte Code-Fragment zeigt, wie die Komponente nach unseren Vorstellungen zu verwenden wäre. Der Benutzer sollte die Möglichkeit haben den Service nach seinen Bedürfnissen zu konfigurieren. Eine dieser Konfigurationen könnte das Setzen einer minimalen Redundanz sein. Im Beispiel wäre nun eine doppelte Redundanz garantiert. Es soll jedoch nicht die Aufgabe des Benutzers sein, diese Redundanz aufrecht zu erhalten. Solche Anforderungen sollten vom System erfüllt werden. Der Benutzer hätte so lediglich eine einfache und saubere Schnittstelle zur Verfügung, über welche er den Service nutzen könnte.

In den uns bekannten DHT-Implementationen wurde eine Ringanordnung der Nodes gewählt. Um eine Routing-Komplexität von O(log n) zu erreichen, unterhält jeder Node eine sogenannte Fingertable. Den Unterhalt dieser Fingertables erachten wir als komplex und aufwendig. Ein Binärbaum hingegen hat die Eigenschaft, dass der Suchalgorithmus grundsätzlich die Komplexität von O(log n) besitzt. Diese Komplexität ist ohne weiteren Verwaltungsaufwand garantiert. Des Weiteren lassen sich die Methoden, welche auf einen Binärbaum angewendet werden können, sehr schön auf die Methoden einer DHT abbilden. Ein Binärbaum hat den weiteren Vorteil, ein bekanntes Konstrukt in der Informatik zu sein und somit gut dokumentiert ist. Darum überzeugte uns die Idee, die Nodes in einem Binärbaum statt einem Ring anzuordnen und wir wählten dieses Thema als Bachelorarbeit.

3 Ziele

3.1 Einfaches Verständnis

Das Konzept unserer DHT soll einfach zu verstehen sein. So sollen der Aufbau und die Funktionsweise vom Leser gut nachvollziehbar sein.

3.2 Einfache Anwendung

Die Software soll als eine komplette Komponente zur Verfügung gestellt werden. Diese sollte sich einfach in andere Projekte einbinden lassen. Vorstellbar wäre eine Bibliothek in Form einer DLL-Datei. Dem Benutzer sollte eine einfache API angeboten werden.

3.3 Hohe Effizienz

Der Verwaltungsaufwand innerhalb der DHT ist klein zu halten. So sollte die Kommunikation zwischen den benachbarten Nodes möglichst effizient sein. Das Routing sollte die Komplexität von O(log n) nicht übersteigen.

3.4 Redundanz

Für das System sollte sich eine minimale Redundanz definieren lassen. D.h. die Werte, welche die DHT verwaltet, müssen auf mehreren Peers gespeichert werden. So dürfen weder bei einem kontrollierten noch bei einem unkontrollierten Ausfall eines Peers Daten verloren gehen.

3.5 Gleichmässige Lastverteilung

Der Rechenaufwand soll über das ganze Netz verteilt werden, um eine Überlastung eines einzelnen Peers zu verhindern. Auch sollten die zu speichernden Daten gleichmässig im Netz verteilt werden.

3.6 Ausfallsicherheit

Der Ausfall eines Peers darf die Funktionsfähigkeit des Systems nicht beeinträchtigen. Eine Selbstreorganisation des Netzes wird erwartet.

3.7 Vollständige Verwaltung der Einträge

Das Löschen, Hinzufügen und Abfragen der Einträge soll unterstützt werden.

3.8 Unterstützung des Ein-/Austritts eines Peers

Ein Peer welcher sich nicht im Netz befindet, soll sich jederzeit im Netz einbinden können. Zugleich sollte auch ein kontrollierter Austritt eines Peers unterstützt werden.

3.9 Grafisch darstellbare Netzstruktur

Ein existierendes Netz soll grafisch aufgezeigt werden können. Die grafische Darstellung soll Informationen enthalten, welche Aussagen über die korrekte Arbeitsweise des Netzes machen.

3.10 Exportierbare Netzdaten

Die wichtigen Daten des Netzes sollten in einer gebräuchlichen Form exportiert werden können. Die exportierten Daten sollen beispielsweise Informationen über den Peer wie dessen Beziehungen und gespeicherten Einträge enthalten.

3.11 Nicht blockierende Aufrufe

Jegliche Aufrufe auf den Peers sollen asynchron ablaufen. D.h. ein Peer wird nicht durch einen einzelnen Befehl blockiert.

3.12 Netzwerkkommunikation mit WCF

Die Kommunikation im Tree soll über WCF realisiert werden. D.h. Methoden eines entfernten Systems sollen lokal über ein Stub- bzw. Proxy-Objekt aufgerufen werden.

4 Konzepte

4.1 Struktur des Netzes

Wir haben entschieden, unsere DHT in Form eines binären Suchbaums bzw. Binary-Search-Tree (BST) aufzubauen. Ein BST ist eine spezielle Form eines Graphen. Jeder Knoten bzw. Node im Tree besitzt höchstens zwei Kinder bzw. Child-Nodes. Somit ist jeder Node mit seinem Elternknoten bzw. Parent-Node und den eigenen Child-Nodes verbunden. Ausnahmen bilden hier der Wurzelknoten bzw. Root-Node und die Blätterknoten bzw. Leaf-Nodes. Der Root-Node hat als einziger kein Parent-Node. Es existiert genau einen Root-Node in einem Tree. Die zweite Ausnahme bilden jeweils die Leaf-Nodes. Diese besitzen keine Child-Nodes. So muss jeder Node jeweils nur drei Verbindungen verwalten. Um der Komplexität von O(h) gerecht zu werden, wobei h der Höhe des Trees entspricht, starten einige Algorithmen, welche auf dem Tree ausgeführt werden, jeweils vom Root-Node aus. Um dies in unserem Netzwerk umzusetzen, muss zu den drei Verbindungen eine zusätzliche vierte Verbindung zum Root-Node aufgebaut werden. Jeder Node kennt also nebst seinem Parent- und Child-Nodes noch den Root-Node.

In einem BST hat jeder Node einen eindeutigen Identifikationsschlüssel bzw. Key. Durch was genau dieser Key repräsentiert wird, steht nicht im Vordergrund. Viel wichtiger ist, dass die Keys untereinander vergleichbar sind. D.h. es ist klar definiert, ob ein Key kleiner, grösser oder gleich ist als ein anderer Key. Diese Bedingungen führen dazu, dass jeder Node durch seinen Key, einen eindeutigen Platz im Tree besitzt. Der Tree ist somit sortiert. Die folgende Abbildung zeigt einen BST mit der Höhe h von 4. Wobei 8 der Root-Node repräsentiert und die Nodes 2, 4, 7, 13 Leaf-Nodes sind.

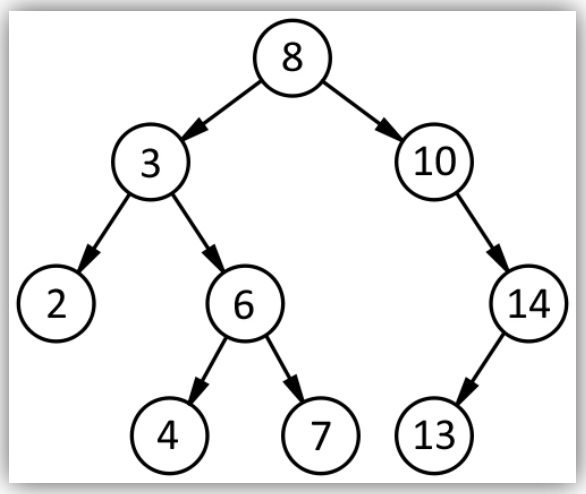


Abbildung 4-1: Binary-Search-Tree [1]

Durch die gewählte Netzstruktur entstehen Vor- bzw. Nachteile, die im Folgenden aufgezeigt werden sollen.

Vorteile

- Durch den Einsatz eines BST als Netzstruktur wird die Komplexität sehr klein gehalten. Alle Operationen, wie zum Beispiel das Hinzufügen oder Entfernen eines Nodes aus dem Tree, sind klar definiert und gut dokumentiert. So kann das Wissen über die verschiedenen Operationen, welche auf einen BST angewendet werden können, schnell erworben werden. Dadurch kann auch das allgemeine Verständnis für unsere DHT gut gewonnen werden.
- Durch die Verwendung eines BST ist automatisch die Komplexität O(h) resp. O(log n) für den Suchalgorithmus gegeben. Dies stellt eine sehr wichtige Eigenschaft für eine DHT dar.

Nachteile

- Ein BST stellt automatisch eine hierarchische Architektur dar. So ist ein Node je weiter oben er sich in Richtung Root-Node befindet, wichtiger, als ein Node der weiter unten im Tree ist. Der Root-Node wird am meisten beansprucht und ist somit am wichtigsten. Denn jede Suchanfrage startet immer als erstes beim Root-Node. Es ist ersichtlich, dass je weiter unten ein Node im Tree ist, seine Wichtigkeit logarithmisch abnimmt. In einem P2P-System sollte aber keine hierarchische Struktur herrschen. Der Umgang mit diesem Problem und dessen Lösung wird im Kapitel [7.1 Hierarchische Struktur] behandelt.
- Wie bereits erwähnt, kennt jeder Node den Root-Node des Trees. Dies bedeutet jeder Node muss zusätzlich eine Verbindung mehr verwalten. Es entsteht also einen gewissen Mehraufwand.

4.2 Begriffsdefinition

Im Zusammenhang mit einem BST werden diverse Begriffe verwendet. Zum Teil werden mehrere Begriffe benutzt, um dasselbe Tree-Element zu beschreiben. Um eine Begriffsverwirrung zu vermeiden, wird im Folgenden jedem Tree-Element genau ein Begriff zugeordnet.

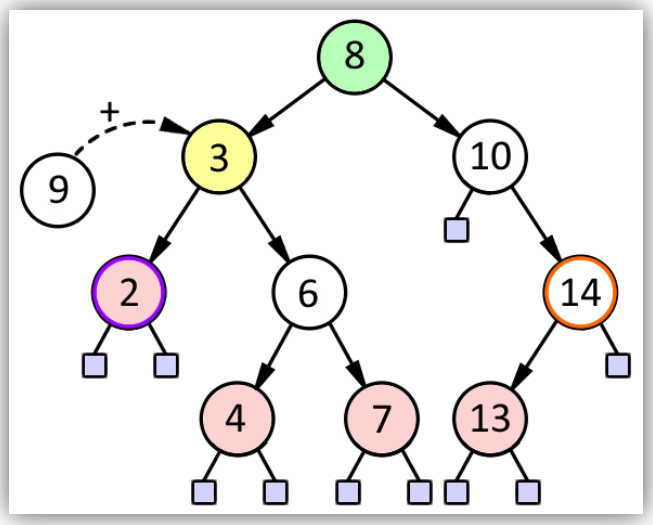


Abbildung 4-2: Begriffsdefinition

6	**Node:**	Ein Node im BST wird durch einen Kreis dargestellt.
	**Parent-, Child-, Left- & Right-Node:**	Der Node, welcher sich eine Stufe höher im Tree als der Eigene befindet, wird als Parent-Node bezeichnet. Der Eigene gilt dabei als Child-Node des Parent-Nodes. Maximal haben zwei Nodes denselben Node als Parent-Node. Jeder Node besitzt genau einen Parent-Node. Eine Ausnahme bildet hier der Root-Node. Der linke Child-Node wird als Left-Node und der rechte Child-Node als Right-Node bezeichnet.
8	**Root-Node:**	Pro BST gibt es immer genau einen Node, welcher als einziger kein Parent-Node besitzt. Dieser Node ist einzigartig und wird als Root-Node bezeichnet.
	NodeNull:	Ein NodeNull stellt einen Platzhalter für einen nicht existierenden Child-Node dar.
4	**Leaf-Node:**	Ein Leaf-Node ist ein Node, bei welchem sowohl der linke wie auch der rechte Child-Node ein NodeNull ist.
2	**First-Node:**	Der Node mit dem kleinsten Key wird als First-Node bezeichnet und befindet sich ganz links im BST. Der Inorder-Vorgänger dieses Nodes ist der Last-Node.

14	**Last-Node:**	Der Node mit dem grössten Key wird als Last-Node bezeichnet und befindet sich ganz rechts im BST. Der Inorder-Nachfolger dieses Nodes ist der First-Node.
3	**Bootstrapping-Node:**	Ein Boostrapping-Node fungiert als Einstiegspunkt für einen Node, welcher sich dem Tree hinzufügen möchte. Jeder Node im Tree kann die Aufgaben eines Bootstrapping-Nodes übernehmen.
	**Predecessor-Node**	Der Predecessor-Node ist der direkte Vorgänger eines Nodes einer Inorder-Traversierung.
6	**Inner-Node**	Als Inner-Node werden Nodes bezeichnet, welche mindestens ein Child-Node besitzen. Es sind also alle Inner-Nodes bis auf die Leaf-Nodes.

Tabelle 4-1: Begriffsdefinition

4.3 Zuständigkeitsbereich eines Nodes

In einer DHT muss ein Wert bzw. Value genau einem Node zugewiesen werden können. D.h. ein Value wird von genau einem Node verwaltet. Ein Node kann jedoch mehrere Values verwalten. Dies zeigt die unten stehende Abbildung. Um eine Zuweisung eines Values an einen Node zu ermöglichen, muss sowohl von den Values wie auch von den Nodes einen Key generiert werden. Damit diese Keys im selben Wertebereich liegen, müssen die Keys mit einem identischen Hash-Algorithmus berechnet werden.

Jeder Node besitzt einen klar definierten Bereich für welche Keys von Values er zuständig ist. Der Zuständigkeitsbereich eines Nodes erstreckt sich vom eigenen Key bis zum Key des Nachfolger-Nodes gemäss Inorder-Traversierung. Die folgende Abbildung soll dies illustrieren.

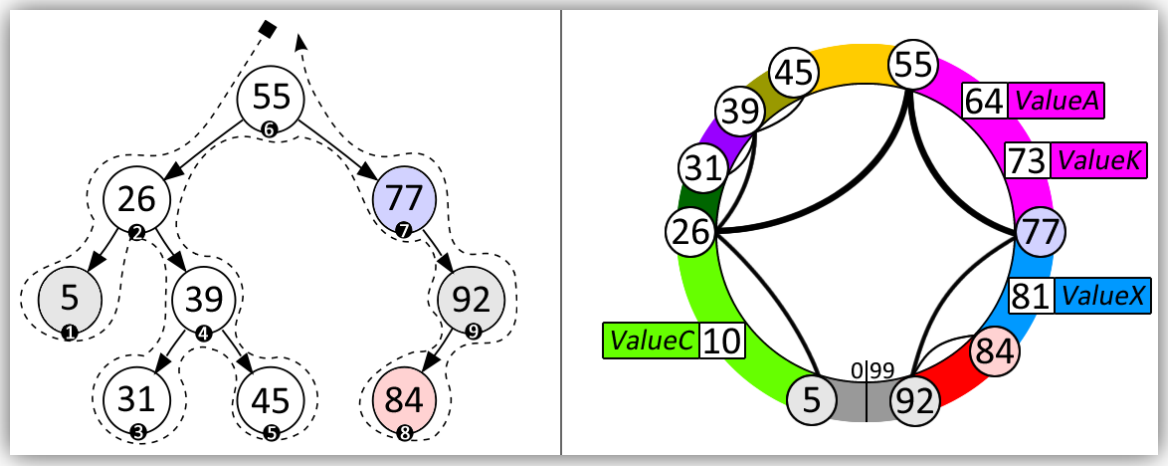


Abbildung 4-3: Zuständigkeitsbereich eines Nodes [2]

Im oben gezeigten BST würde das Resultat einer Inorder-Traversierung wie folgt aussehen: 5, 26, 31, 39, 45, 55, 77, 84, 92. Betrachtet man den blau hinterlegten Node mit dem Key 77, so stellt man fest, dass dessen Inorder-Nachfolger-Node der rot hinterlegte Node mit dem Key 84 ist. Somit ist der Node mit dem Key 77 für die Values mit den Keys zwischen 77 und 84 zuständig. In diesen Zuständigkeitsbereich fällt z.B. das Value mit dem Key 81.

Eine Ausnahme betreffendem Zuständigkeitsbereich bildet jeweils der First- und Last-Node Der Last-Node ist für die Keys, von seinem eigenen Key bis zum maximalen Wert, welcher ein Key annehmen kann, zuständig. Hinzu kommt nun der Bereich, welcher sich vom kleinstmöglichen Key-Wert bis zum Key des First-Nodes erstreckt. Soll nun als Beispiel ein neues Value mit dem Key 2 im Tree abgespeichert werden, so wird dieses nicht auf dem Node mit dem Key 5 sondern auf dem Node mit dem Key 92, welcher für die Key-Wert 92 bis 5 zuständig ist, gespeichert. Somit ist der Node mit dem Key 5 der Nachfolger vom Node mit dem Key 92. Durch das Hinzufügen dieser Ausnahme hat nun jeder Node einen klar definierten Vorgänger.

4.4 Hashbildung von Nodes und Values

Für die Generierung der Keys von Nodes und den gespeicherten Values kommt eine konsistente Hash-Funktion zum Einsatz. Eine solche Hash-Funktion grenzt die möglichen Keys auf einen linearen Wertebereich ein. Dadurch entsteht eine zufällige Verteilung der Keys innerhalb dieses Wertebereichs, was wiederum eine gleichmässige Verteilung der Values auf den Nodes zur Folge hat. Als Hash-Funktion wurde der 160-Bit lange SHA-1 Algorithmus gewählt. Der Key eines Nodes wird aus dessen URL gebildet. So wird beispielsweise der Node mit der Adresse http://192.168.1.6:8000 auf den Key „6898fd42707b7bef8c398920cfa40936abb12343“ abgebildet. Anhand dieses Keys kann sich der Node nun im Tree einordnen und sein Zuständigkeitsbereich ist klar definiert. Um die Keys der Values zu generieren kann der Benutzer jeweils eine Identifikation für das Value angeben. Diese Identifikation ist vom Benutzer frei wählbar und wird für die Berechnung des Hash verwendet. Will ein Benutzer beispielsweise ein Value abspeichern und wählt für dessen Identifikation das Wort „MyValue“ wird dieses Value mit dem Key „dfffea589fcea0a4571169942 2c7a20435b8b9a5“ versehen. Das Abspeichern, Suchen und Löschen dieses Values im System erfolgt nun über den generierten Key, wobei nur ein einziger Node im Tree für die Verwaltung dieses Values zuständig ist.

4.5 Clones

Beim bis jetzt beschrieben Konzept werden Aspekte wie zum Beispiel Redundanz, Lastverteilung und Ausfallsicherheit nicht unterstützt. So würden beispielsweise beim Ausfall eines Nodes bzw. Rechners alle von ihm verwalteten Values verloren gehen. Um solche Aspekte bzw. Anforderungen erfüllen zu können, ist das Einführen eines neuen Konzeptes notwendig.

Die Idee des Konzeptes besteht darin statt einen Node durch nur einen Rechner darzustellen, diesen durch mehrere Rechner, sogenannte Peers, zu repräsentieren. Folgende Abbildung soll dies verdeutlichen.

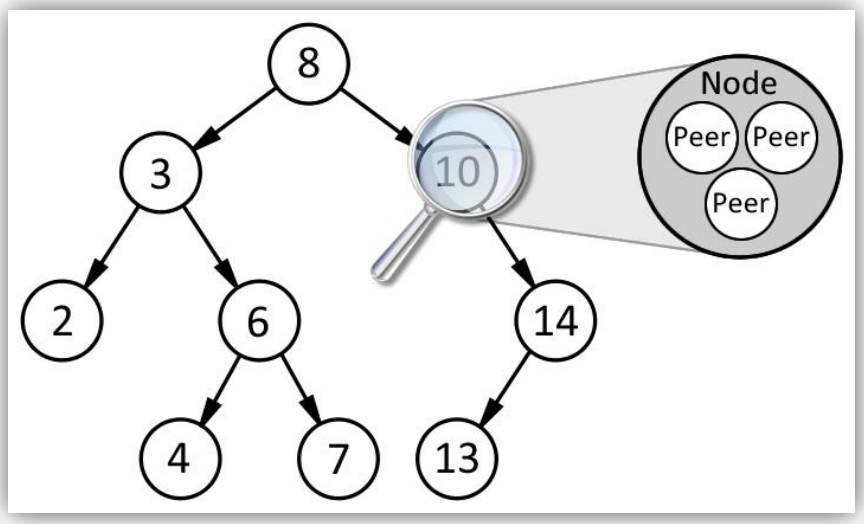


Abbildung 4-4: Repräsentation eines Nodes durch Peers

Da nun mehrere Peers denselben Node repräsentieren, müssen die Peers untereinander identisch gehalten werden. D.h. die Peers sind 1:1 Kopien voneinander und werden daher auch Clones genannt. Eine Anfrage an den Node wird dabei nur von einem Clone des Nodes entgegen genommen und bearbeitet. Der Einsatz von Clones bringt Vorteile wie Redundanz, Lastverteilung und Ausfallsicherheit mit sich. Die untenstehende Abbildung verdeutlicht den Zusammenhang zwischen Node, Peer und Clones.

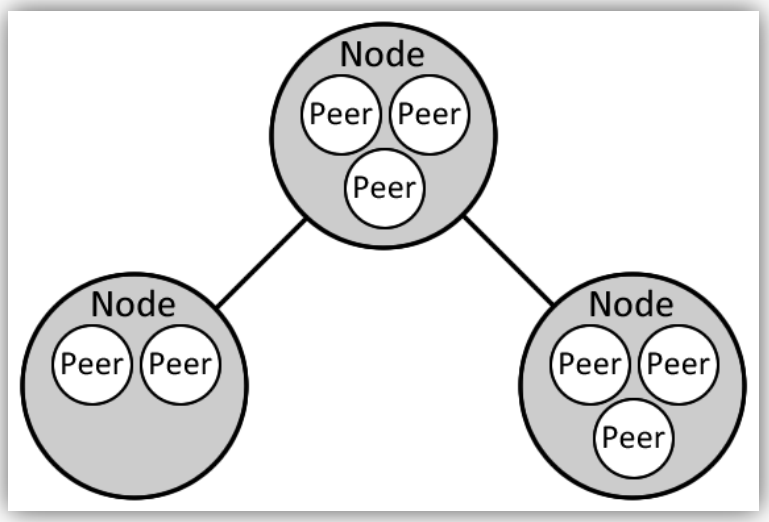


Abbildung 4-5: Node-Details

Jeder Node im Tree wird wie bereits erwähnt durch mehrere Peers dargestellt. Die Peers eines Nodes sind untereinander identisch und somit gleichwertig. Nachfolgend werden diese Peers je nach Kontext auch Clones genannt.

Um die Stärke der Redundanz, Lastverteilung und Ausfallsicherheit konfigurierbar zu halten, ist eine minimale Anzahl Clones pro Node vom Administrator frei wählbar. Das Konzept sieht auch eine maximal erlaubte Anzahl Clones pro Node vor. Die maximale erlaubte Anzahl Clones pro Node ist das Doppelte der minimalen Anzahl Clones. In der oben gezeigten Abbildung ist die minimale Anzahl Peers bzw. Clones pro Node auf zwei gesetzt.

4.5.1 Synchronisationsmechanismus

Das eingeführte Konzept bedingt eine ständige Synchronisation der Clones, für das korrekte Verhalten des Trees. D.h. zu jedem Zeitpunkt muss sichergestellt werden, dass die Clones, welche zusammen einen Node bilden, identisch sind. Wird dies garantiert, so spielt es keine Rolle welcher Peer die Anfrage an den entsprechenden Node entgegennimmt und bearbeitet. Lastverteilung, Redundanz und Ausfallsicherheit wird dadurch ermöglicht. Damit die Clones stets eine 1:1 Kopie darstellen, müssen diese untereinander synchronisiert werden. Sobald ein Peer eine Änderung erfährt, teilt dieser die Änderung über den Synchronisationsmechanismus seinen Clones mit. Eine solche Änderung ist beispielsweise die Anpassung einer Node-Beziehung oder das Hinzukommen bzw. Entfernen eines Key-Value-Pairs. Ein solcher Synchronisationsvorgang soll anhand folgenden Beispiels erläutert werden.

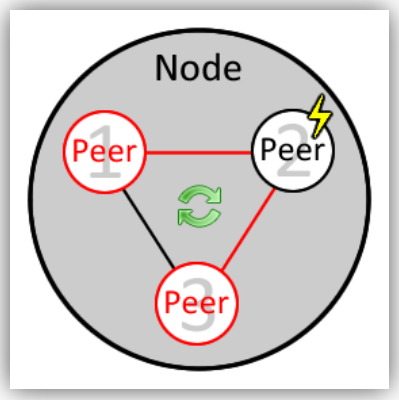


Abbildung 4-6: Synchronisations-Detail

Zu Beginn findet eine Änderung einer Eigenschaft auf dem Peer 2 statt. Somit entsteht eine Ungleichheit zwischen den Clones. Um die Clones identisch zu halten, werden Peer 1 und 3 über die Änderung benachrichtigt und aktualisiert. Nach diesem Vorgang besteht wieder eine Gleichheit unter den Clones.

4.5.2 Keep-Alive-Mechanismus

Es wird ein Mechanismus benötigt, welcher einen unkontrollierten Ausfall eines Clones feststellen kann. Hierzu müssen die Clones untereinander sogenannte Keep-Alive-Nachrichten austauschen. Die Nachrichten werden in konstanten Zeitabständen versendet. Wird eine Keep-Alive-Anfrage nicht bestätigt, wird der entsprechende Clone als nicht erreichbar markiert. Daraufhin wird den erreichbaren Clones die unterbrochene Verbindung mitgeteilt. Die Beziehungen zum ausgefallenen Clone können somit entfernt werden. Das beschriebene Vorgehen entspricht dem Polling-Prinzip.

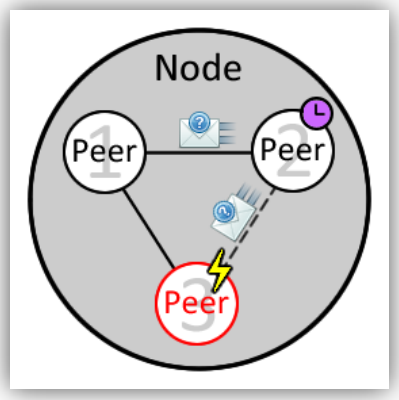


Abbildung 4-7: Keep-Alive-Mechanismus

Obige Abbildung zeigt ein Beispiel in welchem Peer 3 unkontrolliert ausfällt. Eine Zeitperiode ist auf dem Peer 2 abgelaufenen. Er sendet darauf eine Keep-Alive-Nachricht an Peer 1 und Peer 3. Da Peer 3 ausgefallen ist, erhält Peer 2 keine Bestätigungsnachricht von Peer 3. Peer 1 wird durch das im

Kapitel [4.5.1 Synchronisationsmechanismus] beschriebene Konzept über den Ausfall von Peer 3 in Kenntnis gesetzt.

4.5.3 Redundanz

Damit bei einem Ausfall eines Peers die gespeicherten Daten nicht verloren gehen, ist der Einsatz von Redundanz unentbehrlich. Um diese Redundanz zu erzielen kommt das Konzept der Clones zum Einsatz. Alle Clones, die zusammen einen Node darstellen, speichern die Daten jeweils lokal ab. Eine Kopie der Daten wird somit gewährleistet. Dabei ist die Stärke der Redundanz direkt von der Anzahl Clones abhängig.

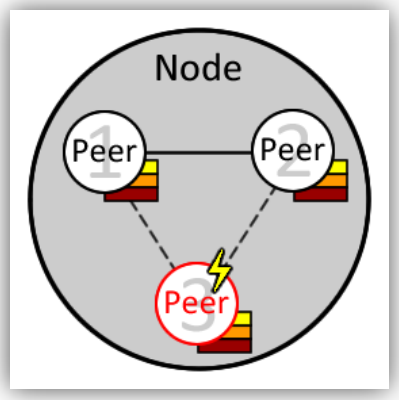


Abbildung 4-8: Redundanz

Die Abbildung zeigt, dass jeder Peer dieselben drei Key-Value-Pairs verwaltet. Diese werden somit redundant im Netz gehalten. Der Ausfall von Peer 3 verursacht kein Datenverlust, da Peer 1 und Peer 2 jeweils Clones von Peer 3 sind. D.h. die Daten sind weiterhin im Netz vorhanden.

4.5.4 Lastverteilung

Durch die Verwendung einer hierarchischen Netzstruktur werden gewisse Nodes stärker als andere belastet. Eine solche Belastung kann durch gezielte Lastverteilung minimiert werden. Die Lastverteilung kann ebenfalls unter Verwendung von Clones erzielt werden. Wird eine Anfrage an einen Node gestellt, so wird zuerst ein Clone des Nodes gewählt, an welchen die Anfrage effektiv gestellt wird. Durch die unterschiedliche Wahl eines Clones entsteht eine gleichmässige Auslastung der Clones. Die Anzahl Clones hat somit einen linearen Einfluss auf die Anzahl der zu bearbeitenden Anfragen pro Peer. Wird beispielsweise die Anzahl Peers pro Node verdoppelt, so hat der einzelne Peer nur halb so viele Anfragen zu bearbeiten.

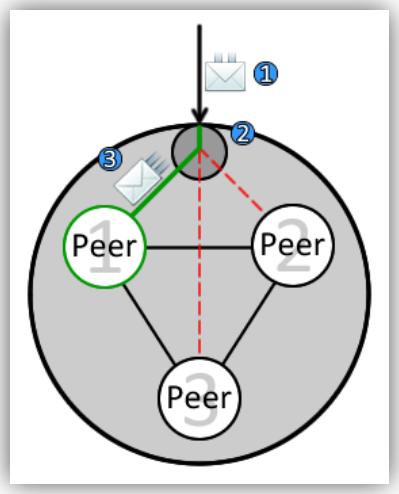


Abbildung 4-9: Lastverteilung

Die Wahl eines Peers innerhalb eines Nodes kann in folgende drei Schritte unterteilt werden.

1. Eine Nachricht wird an den Node gesendet.
2. Der Node wählt abwechslungsweise einen Peer.
3. Die Nachricht wird an den gewählten Peer weitergeleitet und von diesem bearbeitet.

4.5.5 Ausfallsicherheit

Der Pfad von einem Node zu einem anderen führt jeweils durch klar definierte Nodes. Es existiert somit nur eine einzige Verbindung zwischen einem Node und einem Subtree. Würde nun ein Node innerhalb einer solchen Verbindung ausfallen, würde der Pfad unterbrochen und der Tree gespaltet werden. Eine solche Spaltung des Trees darf unter keinen Umständen auftreten da ansonsten getrennte Trees entstehen, die untereinander nicht kommunizieren können.

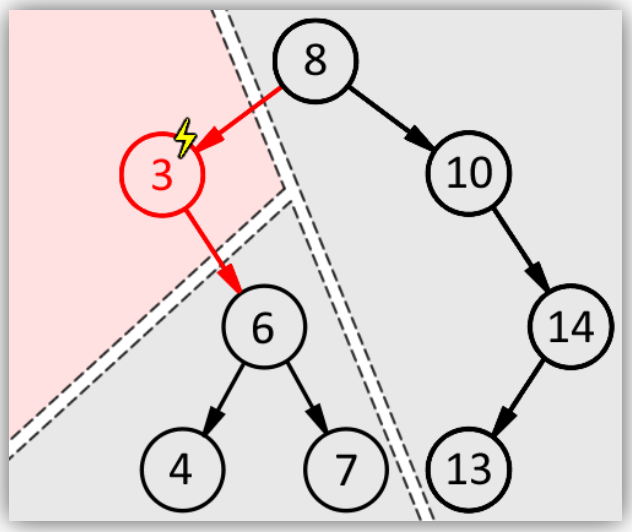


Abbildung 4-10: Tree-Aufspaltung

Die Abbildung zeigt die Trees, die entstehen wenn der Node mit dem Key 3 ausfällt. Durch den Ausfall bilden sich zwei getrennte Trees. Der eine Tree hat neu den Node mit dem Key 6 als Root-Node und der Andere behält den Node mit dem Key 8 als Root-Node. Würde ein Node im Tree nur durch einen Peer repräsentieren werden, so würde der Wegfall eines Peers unmittelbar zum Ausfall eines Nodes führen. Die Gefahr eines Ausfalls resp. einer Spaltung wird durch die Verwendung von Clones verhindert.

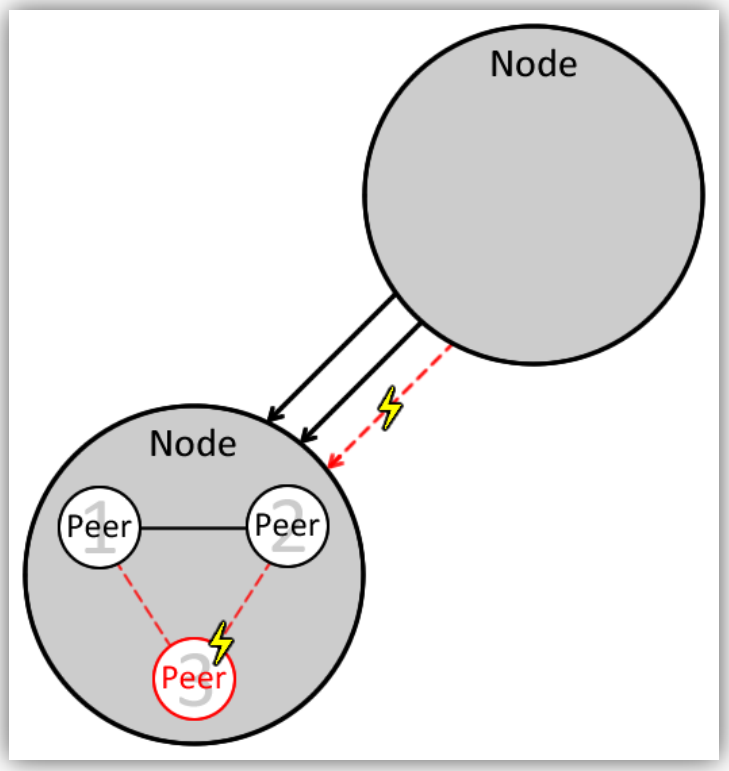


Abbildung 4-11: Ausfallsicherheit eines Node

Da jeder Node durch mehrere identische Clones dargestellt wird, ist der Ausfall eines einzelnen Peers unkritisch. Dies darum, weil der Node weiterhin durch die restlichen Peers repräsentiert wird und somit nicht ausfällt. In der gezeigten Abbildung bleiben Peer 1 und 2 nach dem Ausfall von Peer 3 weiterhin bestehen. Die einzige Änderung besteht darin, dass der Node statt wie zu Beginn durch 3 Peers nur noch durch 2 Peers repräsentiert wird. D.h. der Node kann nur noch über zwei Verbindungskanäle angesprochen werden.

Da sich die Anzahl Clones auf einem Node verringern kann, muss darauf geachtet werden, dass eine untere Grenze der Anzahl Clones nicht unterschritten wird. Darum wird eine minimale Anzahl Clones pro Node definiert. Durch folgendes Konzept wird garantiert, dass dieses Minimum eingehalten wird.

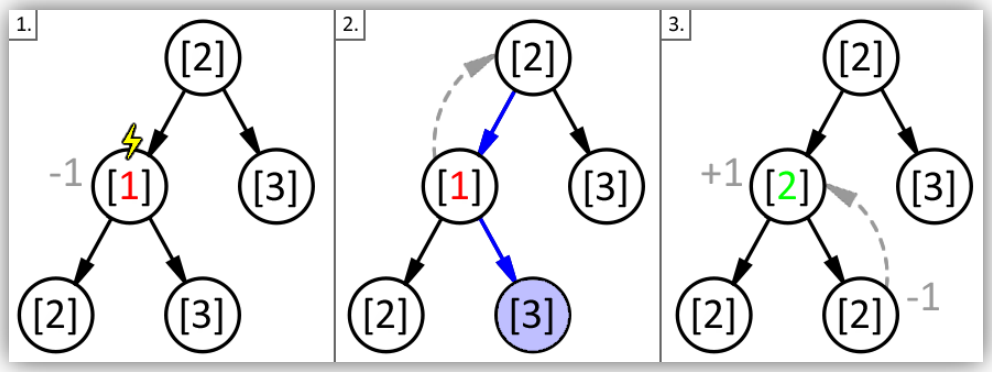


Abbildung 4-12: Peer-Ausfall auf Inner-Node

Wird die untere Grenze auf einem Inner-Node unterschritten, muss sofort ein neuer Clone diesem Node hinzugefügt werden. Um dies zu erreichen wird jeweils ein Clone eines zufällig gewählten Leaf-Node des Subtrees als neuer Clone des Nodes gewählt. Dieser Sachverhalt wird im nachfolgenden Beispiel schrittweise erläutert. Dabei stellen die Zahlen in den eckigen Klammern die Anzahl Clones pro Node dar.

1. Ein Clone fällt auf einem Inner-Node aus. Dadurch wird die minimale Anzahl Clones von beispielsweise 2 unterschritten.
2. Der betroffene Inner-Node startet auf dem Root-Node eine Suche nach einem Leaf-Node. Von diesem fordert er später einen Peer an.
3. Der Node erhält einen Peer vom gefundenen Leaf-Node und klont diesen. Somit erreicht er die geforderte minimale Anzahl Clones.

Es existiert ein unterschiedliches Verfahren, falls die minimale Anzahl Clones auf einem Leaf-Node unterschritten wird. In diesem Fall wird kein weiterer Peer angefordert. Der betroffene Leaf-Node löst sich stattdessen auf. Die Peers des sich auflösenden Leaf-Nodes werden anderen Nodes neu zugeordnet.

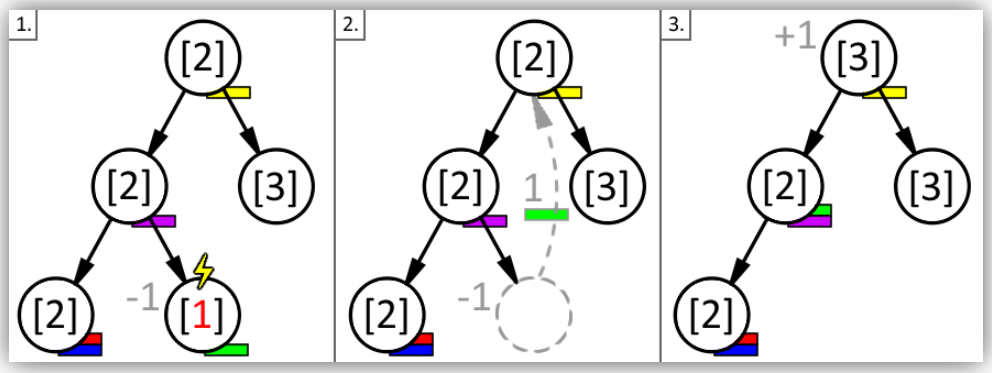


Abbildung 4-13: Peer-Ausfall auf Leaf-Node

Der Ablauf eines Ausfalls eines Clones auf einem Leaf-Node wird anhand der oben gezeigten Abbildung erklärt. Er kann in folgende drei Schritte unterteilt werden.

1. Auf einem Leaf-Node wird die minimale Anzahl von 2 Clones unterschritten.
2. Der betroffene Leaf-Node löst sich auf. Die restlichen Peers und alle Key-Value-Pairs werden über den Root-Node dem Tree neu hinzugefügt. In der Abbildung sind nur ein Peer und ein Key-Value-Pair davon betroffen.
3. Der betroffene Peer wurde ein Clone des Root-Nodes und das Key-Value-Pair wurde auf den Predecessor-Node verschoben.

Durch die beschriebenen Konzepte werden immer nur Leaf-Nodes und nie Inner-Nodes vom Tree entfernt. D.h. wenn Peers ausfallen, wird der gesamte Tree nur von unten her abgebaut.

4.6 Peer-Eintritt

Tritt ein Peer dem Netz bei, so sind zwei Szenarios denkbar. Entweder wird der Peer zu einem Clone eines bereits existierenden Nodes oder es wird ein neuer Node erstellt.

Wurde die maximale Anzahl Clones auf einem Node noch nicht erreicht, wird der neue Peer zu einem Clone dieses Nodes. Der Peer wird an den entsprechenden Child-Node weitergeleitet, falls die maximale Anzahl Clones erreicht wurde. Ist kein Child-Node vorhanden, wird ein neuer Node mit der minimalen Anzahl Clones erstellt. Dabei werden die benötigten Peers und die betroffenen Key-Value-Pairs mitgezogen.

Zuerst wird gezeigt, wie der neue Peer zu einem Clone wird.

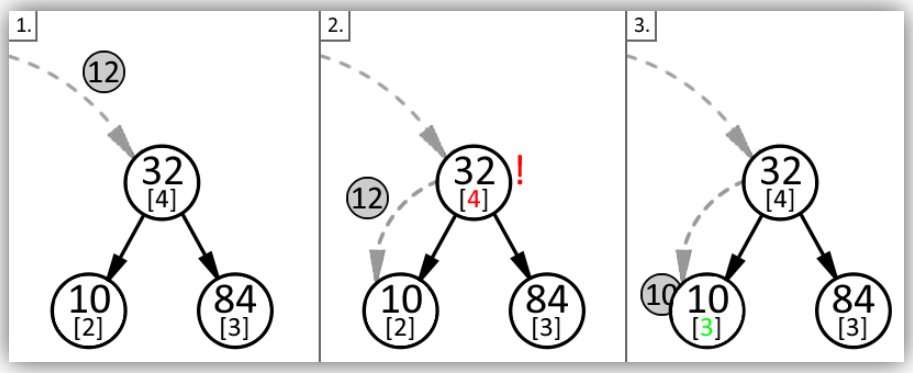


Abbildung 4-14: Hinzufügen eines Peers

Der in der Abbildung gezeigte Ablauf wird in folgende drei Schritte unterteilt.

1. Der Vorgang für das Hinzufügen des neuen Peers wird beim Root-Node gestartet.
2. Die in der Abbildung verwendete minimale Anzahl Clones beträgt zwei. Somit ist das erlaubte Maximum vier Clones pro Node. Da die maximale Anzahl auf dem Root-Node bereits erreicht ist, wird der neue Peer an ein Child-Node weitergereicht. Der Key des Peers beträgt 12 und ist somit kleiner als der Key des Root-Nodes. Deshalb wird der Peer dem Left-Node übergeben.
3. Auf dem Node mit dem Key 10 wurde die maximale Anzahl Clones noch nicht erreicht. Der neue Peer wird zu einem weiteren Clone des Nodes. Im Klonvorgang wird auch der Key übernommen. D.h. der Key 12 wird mit dem Key 10 überschrieben.

In der folgenden Abbildung wird der Ablauf, in welchem ein neuer Node entsteht, in vier Schritte gezeigt.

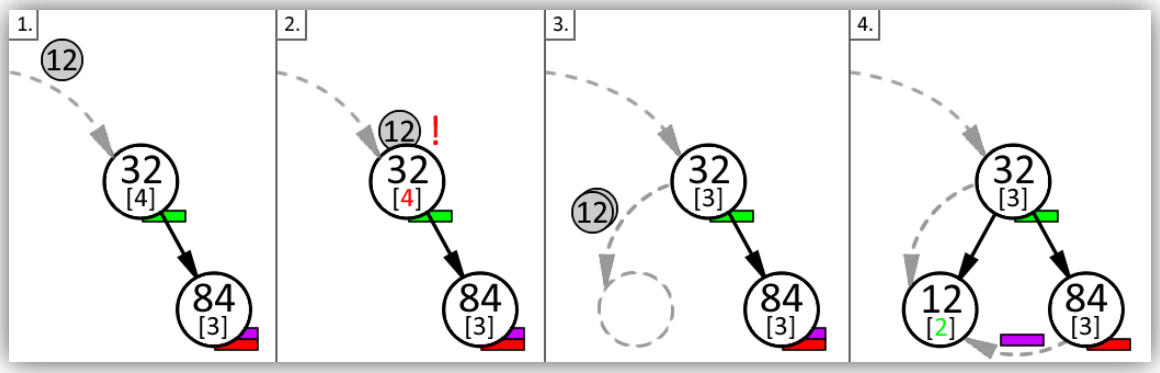


Abbildung 4-15: Hinzufügen eines Peers und Erstellen eines Nodes

1. Der Vorgang für das Hinzufügen des neuen Peer wird auch hier beim Root-Node gestartet.
2. Eine Prüfung der Anzahl Clones findet auf dem Root-Node statt. Da auf dem Root-Node die maximale Anzahl erreicht ist, kann der neue Peer nicht vom Root-Node aufgenommen werden.
3. Der Key 12 ist kleiner als der Key des Root-Nodes und es ist kein Left-Node vorhanden. Auf Grund dessen muss ein neuer Left-Node erstellt werden. Damit auf dem neuen Node die minimale Anzahl Clones sofort garantiert werden kann, muss ein Clone des Parent-Nodes mitgezogen werden.
4. Der neu erstelle Node muss von seinem Predecessor-Node die Key-Value-Pairs anfordern, für welche er nun zuständig ist. Es ist zu beachten, dass im gezeigten Beispiel die Ausnahme vom Kapitel [4.3 Zuständigkeitsbereich eines Nodes] betreffend dem Zuständigkeitsbereich des First- bzw. Last-Node zum Zuge kommt.

Mit den beschriebenen Verfahren wird garantiert, dass die minimale Anzahl Clones auf einem Node stets garantiert ist. Zusätzlich erreichen die im Tree höheren gelegenen Nodes eher die maximale Anzahl Nodes.

4.7 Netzeintritt mittels Boostrapping-Node

Damit ein neuer Peer dem Netz beitreten kann, muss die Adresse eines sich bereits im Netz befinden Peers bekannt sein. Über diesen Peer, welcher nun als Boostrapping-Node fungiert, kann der Prozess zur Aufnahme des Peers in das Netz gestartet werden. Wird keine Adresse angegeben, so wird der Peer als Root-Node eines neuen Netzes gesetzt.

5 Weitere konzeptionelle Ideen

Während dem Ausarbeiten der Konzepte stiessen wir auf weitere Aspekte, welche wir als interessant erachteten aber nicht implementiert haben. Im Folgenden werden die Ideen genauer erläutert und deren Sinn und Zweck aufgezeigt.

5.1 AVL-Tree

In einem BST ist die für eine Suchoperation aufzuwendende Zeit linear von der Höhe des Trees abhängig. Im Worst-Case ist die Höhe so gross wie die Anzahl der vorhandenen Elemente im Tree. Der Tree würde in diesem Fall einer Linked-List entsprechen. Ein AVL-Tree ist ein ausbalancierter BST. Durch den Einsatz eines AVL-Trees wird der beschriebene Worst-Case verhindert. Der Tree müsste allerdings von Zeit zu Zeit anhand der Höhenunterschiede der Subtrees ausbalanciert werden.

5.2 Splay-Tree

Der Einsatz eines Splay-Trees ist denkbar. Jedoch in einer abgeschwächten Form. Ein Peer, welcher als Ergebnis einer Suche gefunden wird, muss in Richtung des Root-Nodes nach oben verschoben werden. Die Verschiebungen finden schrittweise statt. D.h. der entsprechende Peer wird nicht sofort zum neuen Root-Node, sondern lediglich in einer höheren Ebene platziert. So wird die Suchzeit nach einem häufig angesprochenen Peers reduziert.

5.3 Bootstrapping-Node mittels DynDNS

Bis anhin musste dem Peer für den Eintritt in das Netz die IP-Adresse und der Port, eines sich bereits im Netz befindenden Peer, angegeben werden. Durch das Verwenden eines DynDNS-Services kann eine solche manuelle Eingabe vermieden werden. Beim Start eines Peers fragt dieser unter einer well-known URL die nötigen Eintrittsinformationen ab.

Die Verwendung des DynDNS-Dienst www.dyndns.org wäre beispielsweise denkbar. Dessen Interface-Beschreibung lässt sich unter folgender Adresse nachlesen:

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

5.4 Sicherheit

Verschlüsselung, Authentifikation und Authentizität sind wichtige Sicherheitsanforderungen in Bezug auf die Verwendung des Systems in realer Umgebung. Diese Anforderungen lassen sich durch den Einsatz von SSL-Zertifikate erfüllen. Visual Studio 2010 erlaubt ein einfaches Einbinden von SSL-Zertifikaten in WCF-Applikationen.

5.5 Keyword-Suche (Semantic Overlay Network)

In vielen Anwendungsfällen ist dem Benutzer der exakte Key eines Values unbekannt. Um trotzdem dem Benutzer das Auffinden eines Values zu ermöglichen, benötigt es einen weiteren Mechanismus. Dieser besteht darin, dass die jeweiligen Key-Value-Pairs mit einem zusätzlichen Attribut erweitert werden. Das Attribut enthält Tags, welche das Value beschreiben und klassifizieren. So könnte ein PDF-Paper über das Thema „DHT“ welches im Jahr 2010 an dem MIT geschrieben wurde, die Tags PDF, DHT, 2010 und MIT enthalten. Nun ist es dem Benutzer beispielsweise möglich mit den Stichworten DHT und MIT nach diesem Paper zu suchen. Dabei muss er den Key selbst nicht kennen.

5.6 Clustering

Damit nicht alle Peers bei einer Keyword-Suche involviert werden, müssen die Key-Value-Pairs geclustert werden. Im Folgenden werden zwei verschiedene Implementierungsvarianten aufgezeigt.

5.6.1 Mittels Hash-Anpassung

Prinzipiell wird von jedem Key-Value-Pair einen Hash gebildet. Von diesem Hash wird nun eine bestimmte Anzahl der höchstwertigen Bits abgeschnitten und durch eine Kategorie-Nummer ersetzt. Dies hat zur Folge, dass alle Values derselben Kategorie in einem Teilbereich des Trees abgespeichert werden. Folgendes Beispiel soll dies illustrieren.

Gegeben seien der Film „Avatar“, welcher den Hash „7631b26ea80b1b601c313b15cc4e2ab03faedf30“ besitzt und das Musikstück „Let It Be“ mit dem Hash „c2dce76333c5aa3aa6a4385b7c8226aa4d797714“. Für die Bildung der Kategorie-Nummer wird nur das erste Zeichen verwendet. Somit sind 16 Kategorien definierbar. In unserem Beispiel seien die Kategorie „Filme“ mit der Nummer 3 und die Kategorie „Musik“ mit der Nummer 8 definiert. So würden nun die beiden Hashs wie folgt aussehen:

- „Avatar“: 3631b26ea80b1b601c313b15cc4e2ab03faedf30
- „Let It Be“: 82dce76333c5aa3aa6a4385b7c8226aa4d797714

Die beiden Hashs werden dem folgenden Tree hinzugefügt.

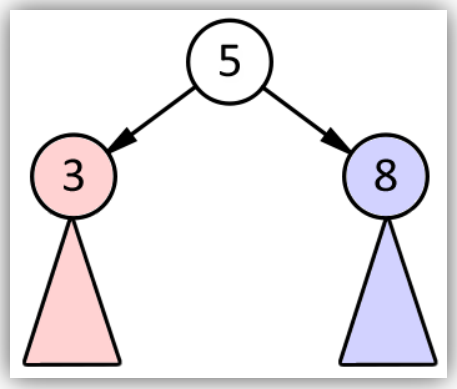


Abbildung 5-1: Clustering mittels Hash-Anpassung

Der Film „Avatar“ wird im roten und das Musikstück „Let It Be“ im blauen Subtree abgespeichert. Bei einer Suche nach dem Film „Avatar“ muss nun nicht mehr der gesamte Tree sondern nur noch die Peers des roten Subtrees durchsucht werden. So wird eine noch effizientere Suche ermöglicht.

5.6.2 Mittels separaten Trees

Die zweite Variante besteht darin, dass ein Tree pro Kategorien gebildet wird. Dabei wird ein zu speicherndes Key-Value-Pair kategorisiert und nur im entpsrechenden Tree gespeichert. Eine Suchanfrage wird an den jeweiligen Root-Node weitergeleitet. Somit muss nur der Tree der entsprechenden Kategorie durchsucht werden.

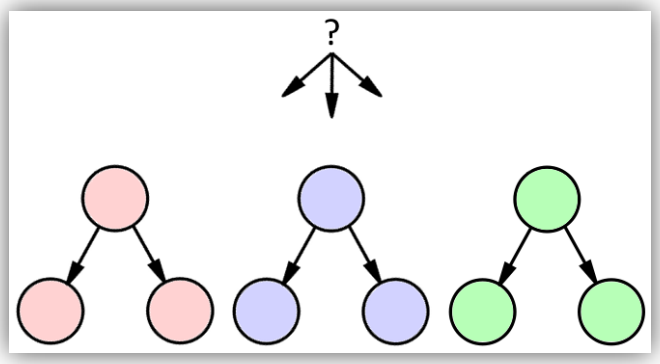


Abbildung 5-2: Clustering mittels separaten Trees

5.7 Logische Zeit

Die ganze Kommunikation im Netz ist asynchron. Das bedeutet, es gibt keine Garantie, dass die Nachrichten in der gleichen Reihenfolge ankommen, in der sie gesendet wurden. Sobald die zeitliche Abfolge der Ereignisse eine Rolle spielt, muss ein Mechanismus gefunden werden, welcher die Ereignisse zeitlich in Beziehung bringt. Um die relative Zeit zwischen zwei auftretenden Ereignissen in einem verteilten System ohne eine globale Uhr herauszufinden, benutzt man die Happened-Before-Relation. Diese Relation wird oft mit Hilfe der logischen Zeit sichergestellt. [3]

5.8 Tree-Höhe abhängige Clone-Anzahl

Das jetzige Konzept sieht vor, dass jeder Node im Tree durch die gleiche Anzahl Clones repräsentiert wird. Dadurch wird das Problem der hierarchischen Struktur abgeschwächt, jedoch nicht komplett gelöst. Damit jeder Peer gleichwertig ist, muss die Anzahl Clones von der Tree-Höhe abhängig sein. D.h. je näher ein Node dem Root-Node ist, desto grösser ist dessen Clone-Anzahl. Dadurch wird eine bessere Lastverteilung unter den Peers hergestellt. Die minimale Clone-Anzahl wird bei den Leaf-Nodes gesetzt. Dies hat zur Folge, dass die minimale Clone-Anzahl in Richtung Root-Node wächst. Anhand eines Beispiels soll das soeben beschriebene Konzept veranschaulicht werden.

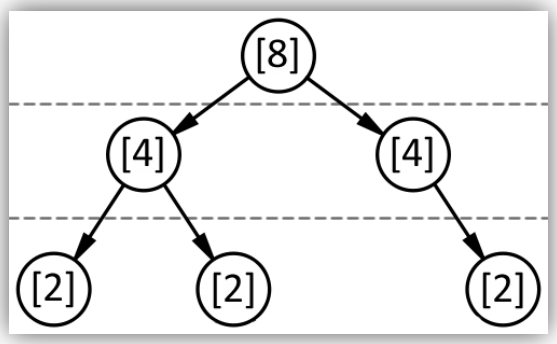


Abbildung 5-3: Tree-Höhe abhängige Clone-Anzahl

In der oben gezeigten Abbildung stellen die in eckigen Klammern gestellten Zahlen die Anzahl Clones des Nodes dar. Das Beispiel zeigt eine exponentiell zur Höhe wachsende Clone-Anzahl. Denkbar wäre auch eine Abhängigkeit, welche nicht exponentiell zur Höhe ist. Je nach Höhe des Trees wäre dies sogar sinnvoller, da ansonsten die Redundanz zu gross werden könnte. Die durch den Benutzer gewählte minimale Clone-Anzahl beträgt in diesem Fall zwei.

Da die Redundanz von der Clone-Anzahl eines Nodes abhängig ist, wächst diese jeweils mit der steigenden Anzahl mit. So ist die Redundanz beispielsweise in einem Leaf-Node deutlich geringer als im Root-Node. Dies ist ein Nebeneffekt, welcher zu beachten ist.

6 Architektur

Die vorhandenen Klassen werden in vier Packages eingeteilt. Jedes Package wird in einem einzelnen Kapitel erklärt. In einem Kapitel wird jeweils das Klassendiagramm gezeigt, die wichtigsten Klassen beschrieben und wenn nötig das Zusammenspiel der Klassen erläutert. Zum Schluss werden diverse Szenarios beschrieben, welche die Interaktion zwischen den einzelnen Klassen anhand Sequenzdiagramme aufzeigen und so zum Verständnis der DHT beitragen.

6.1 Packages

Die vorhandenen Klassen werden in die folgenden vier Packages eingeteilt.

- Core
- Context
- TestAndVisualization
- Utilities

Die Packages und die jeweils enthaltenen Klassen sind in der folgenden Abbildung dargestellt.

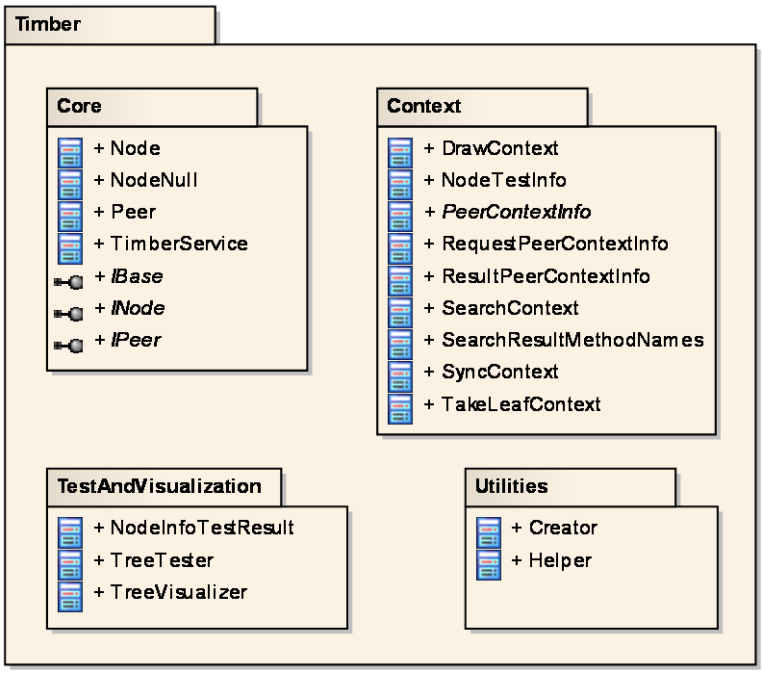


Abbildung 6-1: Package-Diagramm

In den folgenden Kapiteln wird auf die einzelnen Klassen der Packages eingegangen. Dabei wird die Beschreibung der Klassen in die Abschnitte Beschreibung, Verwendungszweck und Merkmale unterteilt. Der Abschnitt Merkmale zeigt die Besonderheiten der Klasse auf. So sollen die nicht intuitiv verständlichen Merkmale einer Klasse erklärt werden.

6.1.1 Core-Package

Das erste Package stellt den Grundstein des zur Verfügung gestellten Services dar. Es enthält die fundamentalen Klassen `Node` und `Peer`, welche das Netz als Tree darstellen.

6.1.1.1 Klassendiagramm

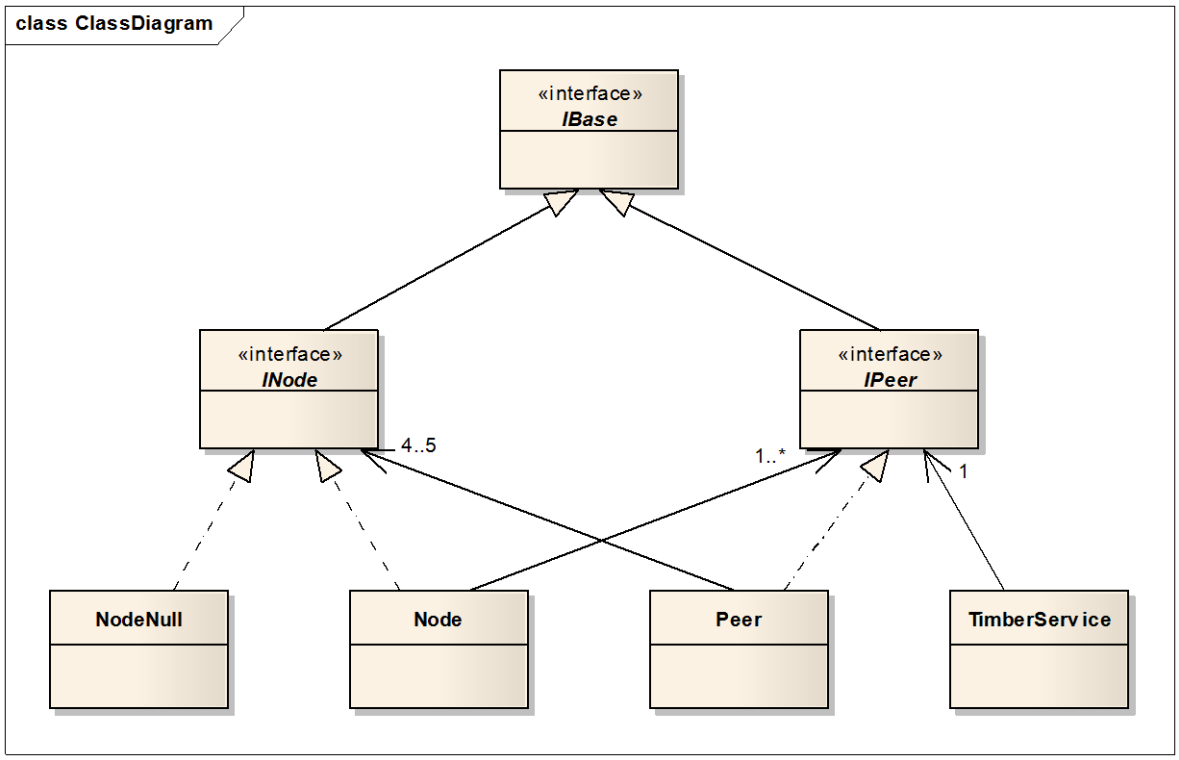


Abbildung 6-2: Klassendiagramm - Hauptklassen

Die Klasse `TimberService` besitzt genau eine Referenz auf einen `IPeer`. Diese Instanz wird auf dem lokalen Computer vom `TimberService` gehostet.

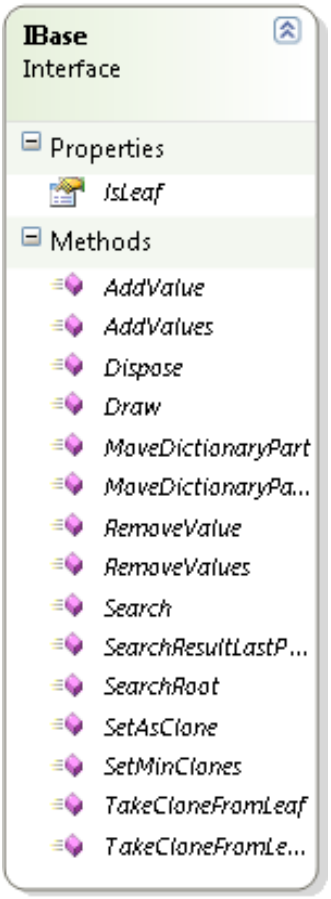
Die Klasse `Peer` kennt mindestens folgende vier `INode`s:

- Root-Node
- Left-Node
- Right-Node
- Eigener Node

Alle `Peer`s bis auf die Clones des Root-Node haben zusätzlich eine Beziehung zu ihrem Parent-Node. Die Beziehungen zwischen `INode` und `IPeer` bzw. deren Ableitungen wird im Kapitel [6.1.1.3 Zusammenspiel] eingehend beschrieben.

6.1.1.2 Interface- und Klassenbeschreibungen

6.1.1.2.1 IBase



Beschreibung

Das Interface `IBase` ist der generalisierte Typ sämtlicher Klassen des Core-Packages mit Ausnahme der Klasse `TimberService`. Alle Methoden, welche sowohl auf einem `Peer` wie auch auf einem `Node` aufgerufen werden können, werden in diesem Interface definiert.

Verwendungszweck

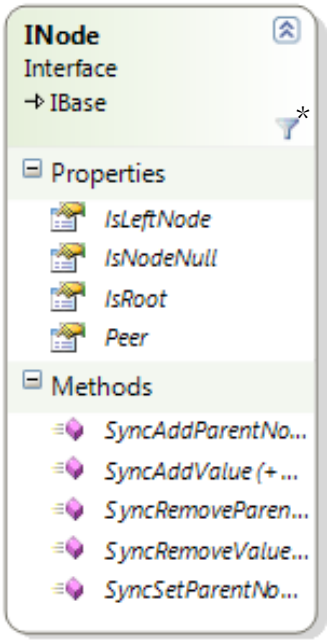
Dieses Interface ermöglicht eine zentrale Definition von gemeinsam verwendeten Methoden. D.h. es ist nur eine einmalige Definition dieser Methoden notwendig.

Merkmale

Alle Methoden, welche über das Netz asynchron auf einem `Peer` aufgerufen werden können, sind mit dem Attribut `OperationContract(IsOneWay=true)` versehen. Dabei bewirk das Property `IsOneWay` den asynchronen Aufruf. Das Interface selber muss daher als `ServiceContract` gekennzeichnet werden. Diese Attribute werden von .Net durch WCF bereitgestellt. Die Verwendung solcher Attribute ermöglicht das Erstellen von Peer-Stubs und das Aufrufen der entsprechenden Methoden über die Grenzen des lokalen Systems hinweg.

Tabelle 6-1: Interfacebeschreibung - IBase

6.1.1.2.2 INode



*Methodenliste reduziert

Beschreibung

Dieses Interface ist ein spezialisierter Typ des Interfaces `IBase` und beschreibt einen Knoten im Tree. Öffentliche Methoden eines Tree-Knotens werden in diesem Interface definiert.

Verwendungszweck

Die Verwendung dieses Interfaces erlaubt die Programmierung gegen eine generalisierte Node-Schnittstelle. Die Erweiterung des BSTs mit unterschiedlichen Knoten wird dadurch stark vereinfacht.

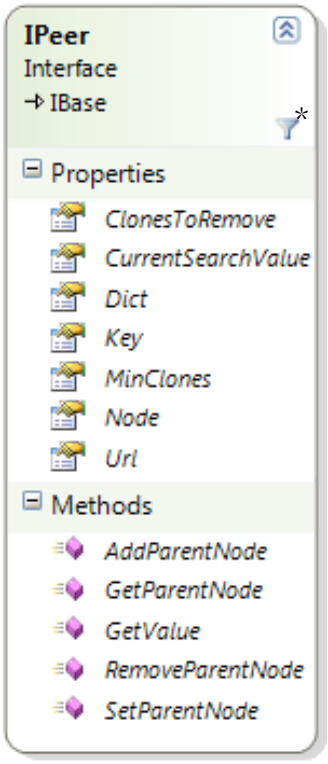
Merkmale

Ein `INode` wird nie über das Netz angesprochen. Die Methoden müssen also nicht mit einem `OperationContract` versehen werden.

Bei vielen der definierten Methoden handelt es sich lediglich um Synchronisationsmethoden.

Tabelle 6-2: Interfacebeschreibung - INode

6.1.1.2.3 IPeer



*Methodenliste reduziert

Beschreibung

`IPeer` ist das generalisierte Interface eines `Peer`s und ein spezialisierter Typ des Interfaces `IBase`.

Verwendungszweck

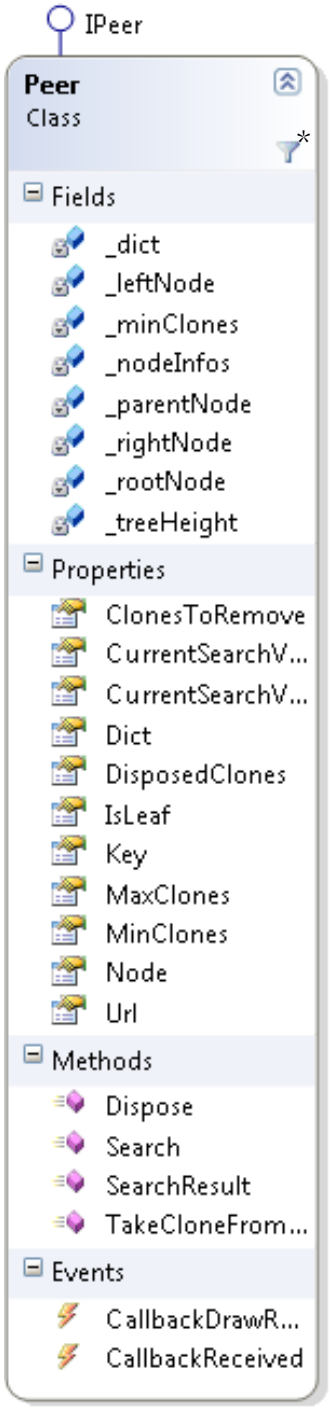
Wie alle Interfaces verhindert auch dieses eine Programmierung gegen konkrete Implementierungen. Die Wartung und Erweiterung des Systems wird dadurch vereinfacht. Entfernte Systeme werden nur über Stubs, welche des Interfaces `IPeer` implementieren, angesprochen. D.h. die Kommunikation über die Grenzen des lokalen Systems hinaus erfolgt nur über die Methoden dieses Interfaces.

Merkmale

Sämtliche Methoden, welche auf einem entfernten System asynchron aufgerufen werden können, müssen mit dem Attribut `OperationContract(IsOneWay=true)` versehen werden. Eine asynchrone Kommunikation über das Netz wird dadurch ermöglicht. Die Attribute `ServiceContract` und `OperationContract` ermöglichen in Zusammenhang mit der WCF-Klasse `ServiceHost` eine Kommunikation über SOAP. Die Instanz der Klasse `ServiceHost`, welche einen `IPeer` kapselt, befindet sich in der Klasse `TimberService`.

Tabelle 6-3: Interfacebeschreibung - IPeer

6.1.1.2.4 Peer



*Methodenliste reduziert

Beschreibung

Jede laufende Instanz von `TimberService` stellt einen `Peer` für das ganze System zur Verfügung. Ein `Peer` ist jeweils ein Teil eines `Node`s und implementiert das `IPeer` Interface. Mehrere `Peer`s zusammen bilden einen `Node` und werden als Clones dieses `Node`s bezeichnet.

Verwendungszweck

Ein `Peer` wird für die Kommunikation zwischen unterschiedlichen Computersystemen verwendet. Durch den Einsatz von `Peer`s bzw. Clones wird Lastverteilung, Redundanz und Ausfallsicherheit ermöglicht.

Merkmale

Die Methode `Draw()` ist eine Tree-Traversierung, welche für das Zeichnen und Testen des Trees verwendet wird. Sie liefert jeweils einen gesamten Ast des Trees zurück. Das Field `_treeHeight` kommt bei dieser Traversierung zum Einsatz. Bei einem Callback der Methode `Draw()` wird jeweils die Höhe des höchsten Astes des Trees in diesem Field gespeichert. Nachdem alle Äste eines Trees erhalten wurden, ist die Höhe des Trees darin gespeichert und kann für das Zeichnen und Testen des Trees verwendet werden.

Das Property `ClonesToRemove` ist eine einfache Zahl, welche bei der Prüfung, ob das Minimum der Anzahl Clones unterschritten wurde, mit einbezogen wird. Das Miteinbeziehen ist notwendig, da während dem Aufruf der Methode `TakeCloneFromLeaf()` weitere `Peer`s einem `Node` hinzugefügt resp. entfernt werden können. Um eine korrekte Prüfung auf das Minimum der Clone-Anzahl vorzunehmen, ist ein solches Vorgehen zwingend.

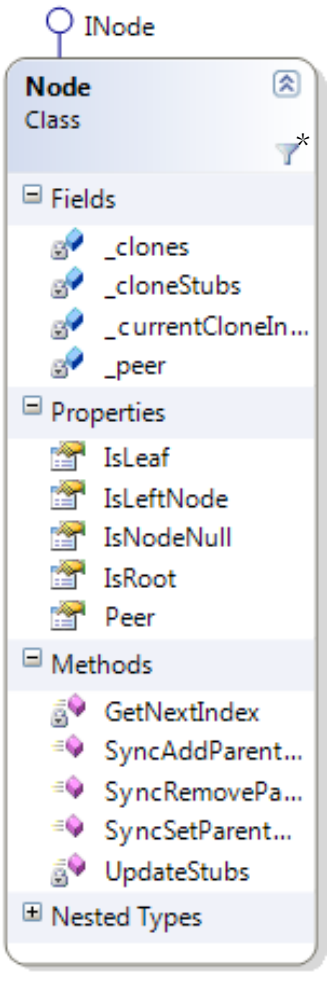
Um ein Value im Tree abzuspeichern, muss zuerst der für das Key-Value-Pair verantwortliche `Node` gefunden werden. Damit das Value während dem Suchvorgang nicht von `Node` zu `Node` weiter gegeben werden muss, wird das Property `CurrentSearchValue` bzw. `CurrentSearchValues` eingesetzt. Diese Properties sind für eine lokale und temporäre Zwischenspeicherung der Values zuständig.

Wenn ein `Peer` das System verlässt und dadurch die Auflösung eines `Node`s verursacht, müssen die Clones darüber Informiert werden. Die URLs dieser Clones werden temporär in die Liste `DisposedClones`

	geschrieben. Ist diese Liste nicht leer, bedeutet dies, dass ein `Node` aufgelöst wurde. Die Clones können also über die Auflösung des `Node`s informiert werden. Die Methode `Search()` wird in diversen Algorithmen verwendet. Um eine Unterscheidung der Callbacks zu ermöglichen, kommt ein ACT zum Einsatz. Die Methode `SearchResult()` dient als Callback-Methode für alle Algorithmen, welche die Methode `Search()` verwenden. Anhand des ACTs wird in dieser nun die effektive SearchResult-Methode aufgerufen.

Tabelle 6-4: Klassenbeschreibung - Peer

6.1.1.2.5 Node

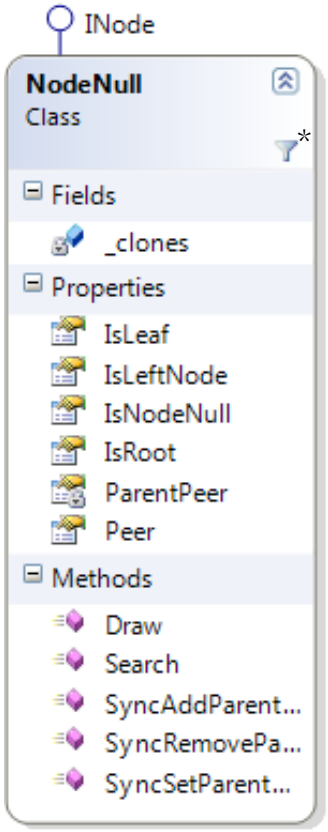


*Methodenliste reduziert

Beschreibung Die Klasse `Node` ist ein Grundelement des BSTs und implementiert das `INode` Interface. Jede Instanz stellt dabei einen Knoten des Trees dar. Ein `Node` wird durch mehrere `Peer`s resp. Clones repräsentiert.
Verwendungszweck Ein `Node` wird für die Kapslung von mehreren `Peer`s bzw. Clones verwendet. Der Hauptzweck dieser Klasse ist nun die Weiterleitung von Anfragen an seine Clones. Je nach Methodenaufruf erfolgt die Weiterleitung auf einen oder alle Clones des `Node`s.
Merkmale Die Methode `GetNextIndex()` kommt bei der Weiterleitung einer Anfrage an einen Clone zum Einsatz. Sie liefert bei jedem Aufruf einen Wert zwischen 0 und der Anzahl Clones - 1 zurück. Des Weiteren wird der zurückgelieferte Wert bei jedem Aufruf um 1 erhöht. Sobald der Wert die Anzahl Clones - 1 überschreitet, wird er wieder auf 0 zurückgesetzt. Jede Weiterleitung einer Anfrage kann nun anhand des zurückgelieferten Wertes an einen anderen Clone erfolgen. Durch diesen Mechanismus wird eine gleichmässige Beanspruchung der Clones und somit eine saubere Lastverteilung erzielt. In einem `Node` muss sichergestellt werden, dass die Stubs der Clones jeweils aktuell sind. Dies wird durch die Methode `UpdateStubs()` bewerkstelligt. Durch den Aufruf der Methode werden aus den mitgegebenen URLs neue Stubs generiert.

Tabelle 6-5: Klassenbeschreibung - Node

6.1.1.2.6 NodeNull



*Methodenliste reduziert

Beschreibung

Ein `NodeNull` ist eine weitere Klasse, welche das Interfaces `INode` implementiert. Er stellt einen Platzhalter für einen nicht existierenden Child-Node dar.

Verwendungszweck

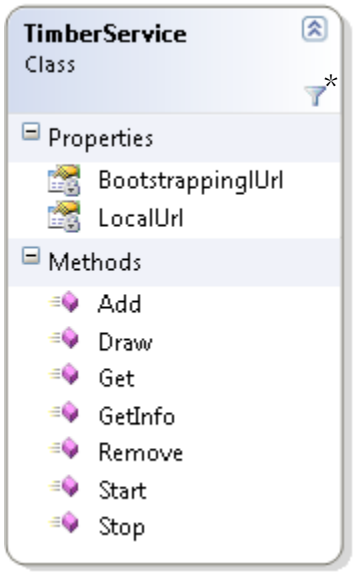
Wenn ein `Node` kein Left- oder Right-Node besitzt, wird stattdessen ein `NodeNull` eingesetzt. Wird eine Methode eines Algorithmus auf einem `NodeNull` aufgerufen, bedeutet dies, dass der Algorithmus terminiert. Dies kann durch die Implementation eines Default-Verhalten oder den Aufruf einer Callback-Methode erfolgen. Die Klasse `NodeNull` ermöglicht das Weiterleiten eines Methodenaufrufs, auch wenn der `Node` kein Child-Node besitzt. So wird die ständige Prüfung des Child-Nodes auf `Null` vermieden. Die Klasse `NodeNull` stellt eine Implementation des Patterns „Null-Object“ dar.

Merkmale

Viele Methoden der Klasse `NodeNull` sind sogenannte Dummy-Methoden. D.h. sie implementieren keine Logik. Aufgrund dessen haben sie auch keinen Einfluss auf das Verhalten des Trees.

Tabelle 6-6: Klassenbeschreibung - NodeNull

6.1.1.2.7 TimberService



*Methodenliste reduziert

Beschreibung

Die Klasse `TimberService` stellt das API der „Timber!"-DHT dem Benutzer zur Verfügung.

Verwendungszweck

Die Klasse `TimberService` wird benötigt, um die Komplexität des Systems zu verbergen. Sie bietet sämtliche Methoden an, welche zur Interaktion mit der DHT benötigt werden. Aufrufe auf dieser Klasse werden prinzipiell nur weitergeleitet. Somit erinnert die Klasse `TimberService` an eine Art „Facade"-Pattern.

Merkmale

Die Klasse `TimberService` bietet die typischen DHT-Methoden wie z.B. `Get()`, `Add()` und `Remove()` an. Die Klasse enthält eine Instanz der .Net-Klasse `ServiceHost`. Diese ist eine WCF-Klasse und ermöglicht eine Kommunikation über SOAP. Die Instanz kapselt den lokalen `IPeer`.

Tabelle 6-7: Klassenbeschreibung - TimberService

6.1.1.3 Zusammenspiel

Die Beziehungen zwischen `INode` und `IPeer` bzw. deren Ableitungen werden im Folgenden beschrieben. Zu Beginn wird der Aufbau eines `Peer`s detailliert aufgezeigt.

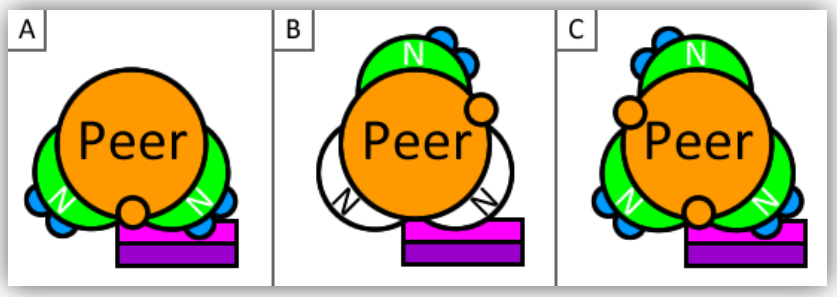


Abbildung 6-3: Aufbau eines Peers

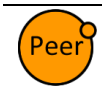
	**Peer**	Eine Instanz der Klasse `Peer` referenziert 4 bis 5 Objekte, welche das Interface `INode` implementieren. Dabei sind in der Abbildung weder der eigene Node noch der Root-Node dargestellt.
	**Node**	Die vom `Peer` referenzierten `INode`s können unter anderem Objekte der Klasse `Node` sein. Ein Node referenziert zwischen einem und mehreren `IPeer`s. Dabei stellen die `IPeer`s die jeweiligen Stubs dar.
	**NodeNull**	Ein vom `Peer` referenzierten `INode` kann unter Umständen ein Objekt der Klasse `NodeNull` sein. Ein `NodeNull` referenziert keine `IPeer`s bzw. Stubs.
	**IPeer / Stub**	Ein Stub ist vom Typ `IPeer`. Er stellt einen Stellvertreter für einen entfernten `Peer` dar und dient somit als Proxy. Die Generierung der Stubs bzw. Proxy wird vom WCF-Framework übernommen. Dies ermöglicht gleichzeitig eine Kommunikation über SOAP.
	**Key-Value-Pairs**	Jeder `Peer` speichert eine gewisse Anzahl Key-Value-Pairs.

Tabelle 6-8: Legende zum Aufbau eines Peers

Je nach dem wo sich ein `Peer` im Tree befindet, ist dessen Anzahl Referenzen auf `INode`s und deren konkreten Implementation unterschiedlich. Die Unterschiede sind in der Abbildung [Abbildung 6-3: Aufbau eines Peers] aufgezeigt und werden im Folgenden beschrieben.

A	**Root-Node**	Da der `Peer` keinen Parent-Node referenziert, handelt es sich um einen Clone eines Root-Nodes. Der `Peer` referenziert sowohl ein Left- wie auch ein Right-Node.
B	**Leaf- bzw. Left-Node**	Der `Peer` referenziert nun ein Parent-Node, er besitzt jedoch als Child-Nodes zwei `NodeNull`s. Die Stubs des Parent-Node zeigen nach rechts. Somit handelt es sich um einen Clone eines Leaf-Nodes, welcher zugleich auch ein Left-Node repräsentiert.
C	**Inner- bzw. Right-Node**	Es handelt sich hier um einen Clone eines Inner-Nodes da ein Parent-Node sowie mindestens ein Child-Node referenziert werden. Der entsprechende `Node` stellt wegen den nach rechts zeigenden Stubs des Parent-Nodes zusätzlich einen Right-Node dar.

Tabelle 6-9: Unterschiedliche Referenzen eines Peers

In diesem Abschnitt wird die effektive Kommunikation zwischen zwei `Node`s erläutert. Dabei wird die Abstraktion der kommunizierenden `Node`s aufgebrochen und die konkrete Implementierung aufgezeigt. Dabei ist zu beachten, dass in der folgenden Abbildung ein Objekt und dessen Stellvertreter dieselbe Farbe haben.

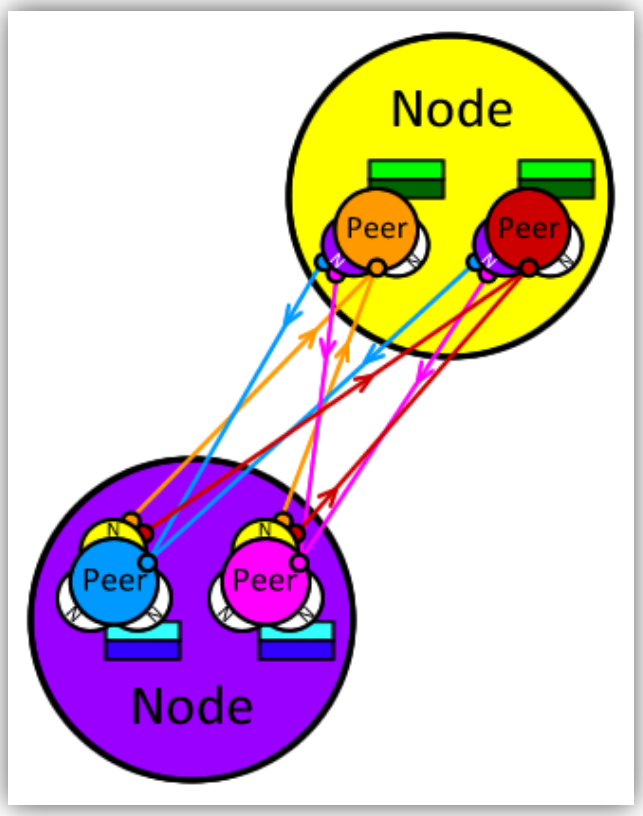


Abbildung 6-4: Beziehungen zwischen Nodes im Detail

Der gelbe Node wird durch zwei identische Peers bzw. Clones repräsentiert. Der orange und rote Clone referenzieren beide dieselben Child-Nodes. Dies ist anhand der übereinstimmenden Färbung der Child-Nodes ersichtlich. Die Left-Nodes der beiden Clones besitzen die gleichen IPeers bzw. Stubs. Ein Left-Node und dessen Stubs bilden zusammen einen Stellvertreter für den violetten Node. Dabei fungiert der Stellvertreter als Proxy-Objekt. Wie auch der gelbe Node wird der violette Node durch zwei Clones repräsentiert. Die vier Stubs der beiden Clones des gelben Nodes verweisen auf die beiden Clones des violetten Nodes.

Die Parent-Nodes der beiden Clones des violetten Nodes funktionieren nach demselben Prinzip wie die jeweiligen Left-Nodes der beiden Clones des gelben Nodes. Die erklärte Architektur ermöglicht eine in beide Richtungen arbeitende Kommunikation im Tree.

In der Abbildung [Abbildung 6-4: Beziehungen zwischen Nodes im Detail] ist ersichtlich, dass die Stubs und Key-Value-Pairs der Peers der jeweiligen Nodes identisch sind. Somit werden die Stubs und Key-Value-Pairs redundant gehalten. Durch die Redundanz besteht die Notwendigkeit der Synchronisation. Ohne Synchronisation der Clones entstehen inkonsistente Zustände, welche negative Einflüsse auf das Verhalten des Trees mit sich ziehen. Zusätzlich ist eine Synchronisation mit dem Parent- und den Child-Nodes notwendig. D.h. ein Node und dessen Stellvertreter müssen identisch gehalten werden. Somit muss auf jedem Clone des Parent- bzw. Child-Nodes ein neuer Stub erstellt werden, falls auf dem aktuellen Node ein neuer Clone hinzukommt. Die neuen Stubs referenzieren demzufolge den neuen Clone.

6.1.2 Context-Package

Das zweite Package fasst die Kontextklassen zusammen. Eine Kontextklasse beinhaltet alle nötigen Informationen, welche bei einer Kommunikation zwischen zwei Nodes ausgetauscht werden. Je nach Algorithmus werden andere Informationen ausgetauscht. Darum existiert oft pro Algorithmus eine eigene Kontextklasse.

6.1.2.1 Klassendiagramm

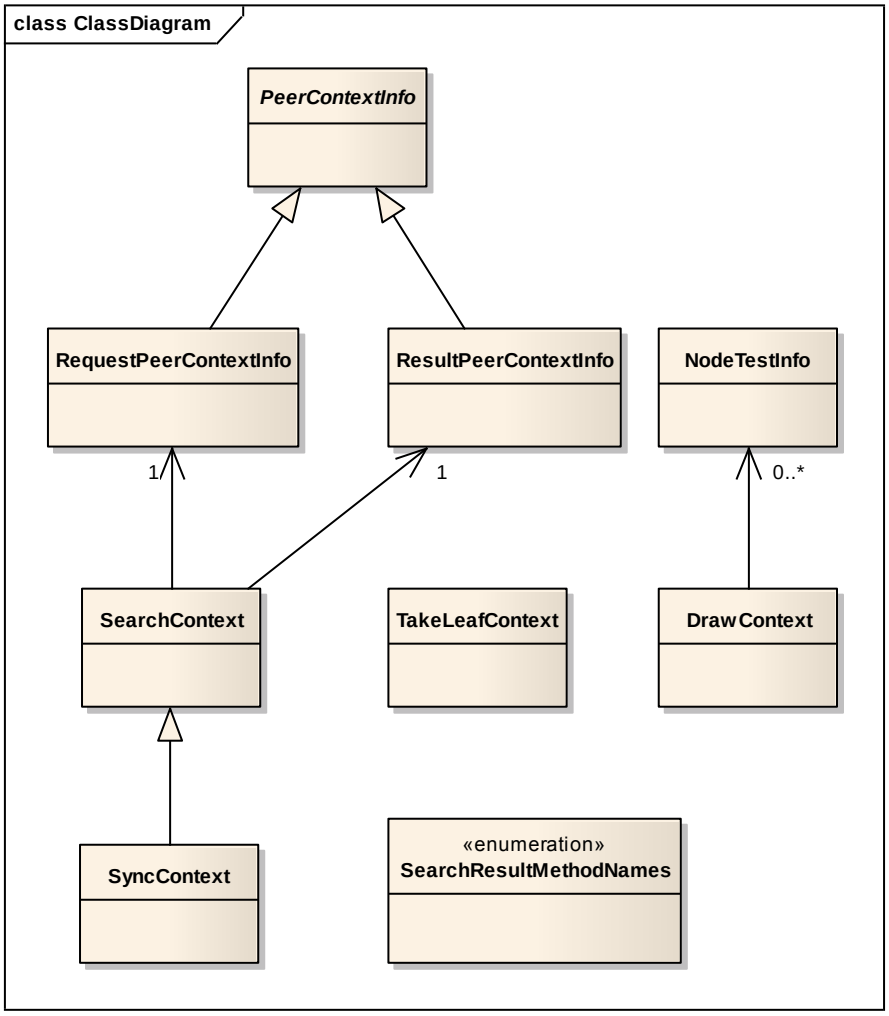
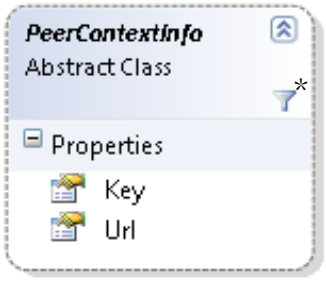


Abbildung 6-5: Klassendiagramm - Kontextklassen

6.1.2.2 Klassenbeschreibungen

Um Instanzen von Klassen mit Hilfe des WCF-Frameworks zu übermitteln, müssen diese Klassen mit dem Attribut `DataContract` versehen werden. Des Weiteren müssen deren Properties mit `DataMember` attributiert werden. Darum sind alle folgende beschriebenen Kontextklassen mit den entsprechenden Attributen gekennzeichnet. Eine Ausnahme stellt der Aufzählungstyp `SearchResultMethodNames` dar.

6.1.2.2.1 PeerContextInfo



*Methodenliste entfernt

Beschreibung

Die abstrakte Klasse `PeerContextInfo` legt die grundlegenden Properties `Key` und `Url` für ihre Unterklassen fest.

Verwendungszweck

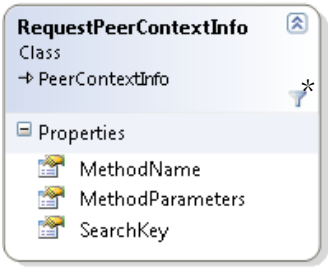
Die Klasse wurde erstellt um die Gemeinsamkeiten der Klassen `RequestPeerContextInfo` und `ResultPeerContextInfo` zusammenzufassen. Die Klasse stellt ein Teil des Patterns „Context-Object“ dar. Sie fasst alle Informationen zusammen, welche bei einer Kommunikation zwischen `Peer`s ausgetauscht werden. Sie verhindert eine grosse Anzahl Parameter bei Methoden Aufrufe und ermöglicht eine effiziente Erweiterung der auszutauschenden Informationen.

Merkmale

Der `Key` ist ein `string` in Form eines SHA1-Hashwertes. Die `Url` beinhaltet die komplette Adresse eines `Peer`s und ist vom Typ `string`. Ein Beispiel für einen Wert wäre „http://192.168.1.6:8000“.

Tabelle 6-10: Klassenbeschreibung - PeerContextInfo

6.1.2.2.2 RequestPeerContextInfo



*Methodenliste entfernt

Beschreibung

Die Klasse `RequestPeerContextInfo` leitet von der Klasse `PeerContextInfo` ab. Die Klasse beinhaltet alle benötigten Informationen über den `Peer`, welcher einen Algorithmus startet oder sonst eine Anfrage stellt.

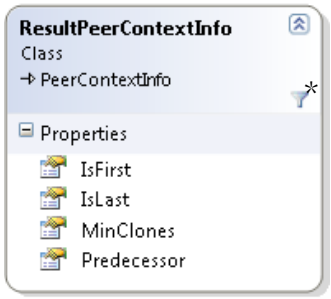
Verwendungszweck

Die Klasse wird benutzt um die Informationen über einen `Peer`, welcher eine Anfrage stellt, zusammenzufassen und zu speichern.

	Merkmale Anhand des Property `MethodName` wird das Pattern „ACT“ umgesetzt. D.h. das Property bestimmt welche Callback-Methode aufgerufen wird.

Tabelle 6-11: Klassenbeschreibung - RequestPeerContextInfo

6.1.2.2.3 ResultPeerContextInfo



*Methodenliste entfernt

Beschreibung

Wie auch `RequestPeerContextInfo` leitet die Klasse `ResultPeerContextInfo` von `PeerContextInfo` ab. Die Klasse `ResultPeerContextInfo` beinhaltet alle Informationen über einen `Peer`, welcher nach der Terminierung eines Algorithmus die Callback-Methode aufruft.

Verwendungszweck

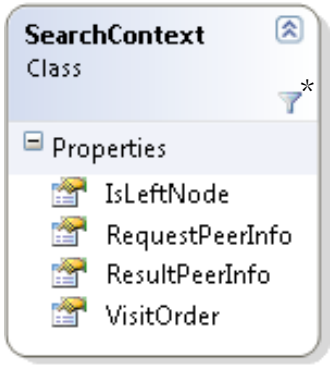
Die Klasse wird benötigt um beim Ablauf eines Algorithmus den Predecessor-Node zu speichern. Zusätzlich ist durch die Klasse zu erfahren, ob sich der entsprechende `Peer` in einem Last- resp. First-Node befindet.

Merkmale

Die beiden Properties `IsFirst` und `IsLast` werden für das Ausnahmeverhalten beim Suchalgorithmus verwendet. Das Property `Predecessor` wird gespeichert, da die einzelnen `Peer`s ihren Predecessor-Node nicht verwalten resp. speichern.

Tabelle 6-12: Klassenbeschreibung - ResultPeerContextInfo

6.1.2.2.4 SearchContext



*Methodenliste entfernt

Beschreibung

Die beiden Klassen `RequestPeerContextInfo` und `ResultPeerContextInfo` treten immer gemeinsam auf. Daher existiert die Klasse `SearchContext`, welche Referenzen auf Instanzen der beiden Klassen hält. Sie ermöglicht eine übersichtliche Struktur der Informationen.

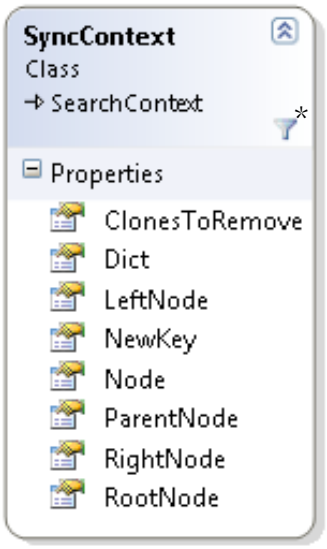
Verwendungszweck

Die Klasse entspricht dem Pattern „Context-Object“. Sie wird benötigt um sämtliche Informationen zusammenzufassen und zu strukturieren, welche bei einem Suchalgorithmus verwendet werden.

	Merkmale Das Property `VisitOrder` enthält die URLs der Clones alle `Node`s in der entsprechenden Aufrufreihenfolge, welche im Suchalgorithmus involviert waren.

Tabelle 6-13: Klassenbeschreibung - SearchContext

6.1.2.2.5 SyncContext



*Methodenliste entfernt

Beschreibung

Die Klasse `SyncContext` stellt eine Spezialisierung der Klasse `SearchContext` dar.

Verwendungszweck

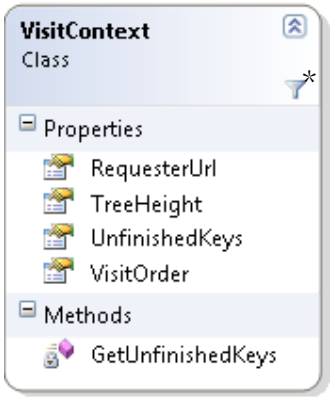
Die Klasse wird im Zusammenhang mit dem Synchronisations-algorithmus benötigt. Da beim Synchronisationsalgorithmus mehr Informationen benötigt werden, wurde eine Spezialisierung der Klasse `SearchContext` notwendig.

Merkmale

Bei einer Synchronisation sind jeweils der Parent- und die Child-Nodes betroffen. Darum besitzt die Klasse `SyncContext` die Attribute `ParentNode`, `LeftNode` und `RightNode`.

Tabelle 6-14: Klassenbeschreibung - SyncContext

6.1.2.2.6 DrawContext



*Methodenliste reduziert

Beschreibung

Es existiert einen Algorithmus, welcher durch den ganzen Tree abläuft. D.h. es sind alle `Node`s im Tree involviert. Die Klasse `DrawContext` stellt das Kontext-Objekt für diesen Algorithmus dar.

Verwendungszweck

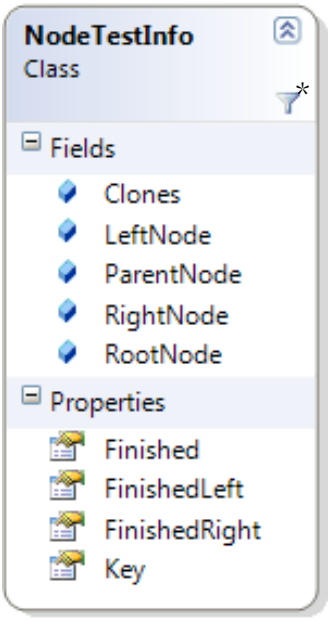
Eine Instanz der Klasse `DrawContext` speichert Informationen über den besuchten `Peer`.

Merkmale

Das Property `TreeHeight` wird für die grafische Darstellung des Trees verwendet. Anhand der Höhe lässt sich die Anzahl `Node`s auf der jeweiligen Stufe und im gesamten Tree berechnen.

Tabelle 6-15: Klassenbeschreibung - DrawContext

6.1.2.2.7 NodeTestInfo



*Methodenliste entfernt

Beschreibung

Während der Tree-Traversierung werden die Beziehungen jedes `Node`s in die Klasse `NodeTestInfo` gespeichert.

Verwendungszweck

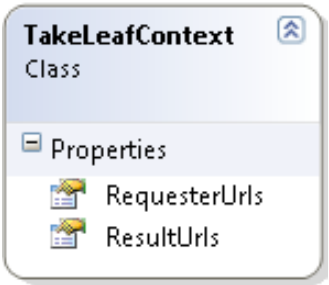
Die in der Klasse `NodeTestInfo` gespeicherten Daten werden zu einem späteren Zeitpunkt für das Testen und Zeichnen des Trees verwendet. Des Weiteren kann anhand der gespeicherten Informationen erkannt werden, wann die Tree-Traversierung abgeschlossen ist.

Merkmale

Wird ein Ast eines `Node`s traversiert, so wird das Property `FinishedLeft` bzw. `FinishedRight` auf `true` gesetzt. Wurden sowohl der linke, wie auch der rechte Ast eines `Node`s vollständig traversiert wird das Property `Finished` auf `true` gesetzt. Anhand dieses Properties kann nun erkannt werden, ob alle `Node`s des Trees besucht wurden. D.h. wenn bei jedem `Node` das Property auf `true` gesetzt ist, wurde der Tree vollständig traversiert.

Tabelle 6-16: Klassenbeschreibung - NodeTestInfo

6.1.2.2.8 TakeLeafContext



Beschreibung

Müssen `Peer`s eines Leaf-Nodes angefordert werden, so wird ein separater Algorithmus gestartet. Die während dem Algorithmus benötigten Informationen werden über die Klasse `TakeLeafContext` ausgetauscht.

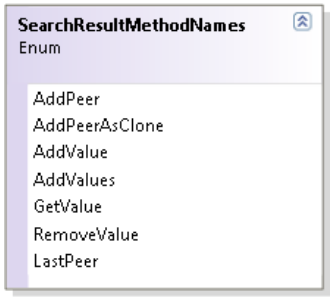
Verwendungszweck

Die in der Klasse gespeicherten Informationen werden vom aufrufenden `Peer` benötigt. Dieser kann aus den Informationen den `Node` ausfindig machen, von dessen er die benötigte Anzahl `Peer`s anfordern kann. Damit aber die Callback-Methode vom aufrufenden `Node` aufgerufen werden kann, wird dessen Adresse im Property `RequestUrls` des Kontext-Objekts mitgegeben.

	Merkmale Die Adresse eines `Node`s wird in Form einer Liste mit allen URLs der Clones eines `Node`s gespeichert.

Tabelle 6-17: Klassenbeschreibung - TakeLeafContext

6.1.2.2.9 SearchResultMethodNames



Beschreibung Der Aufzähltyp `SearchResultMethodNames` definiert alle Methodennamen der Callback-Methoden.
Verwendungszweck Jeder Methodenname stellt eine Callback-Methode dar. Die Methodennamen sind als `string`s abrufbar. Anhand der gelieferten `string`s und mit Hilfe von Reflection wird in der Methode `SearchResult()` der Klasse `Peer` die entsprechende Callback-Methode aufgerufen. Dieser Mechanismus wird für das ACT verwendet.
Merkmale Damit die Werte des Aufzählungstyps als `string` verfügbar sind, wurden diese mit dem eigenen Attribut `StringValueAttribute` versehen.

Tabelle 6-18: Klassenbeschreibung - SearchResultMethodNames

6.1.2.3 Zusammenspiel

Die nachfolgende Abbildung soll den Einsatz eines Kontext-Objektes erörtern. Dabei wird gezeigt wie die benötigten Informationen weitergereicht und ergänzt werden.

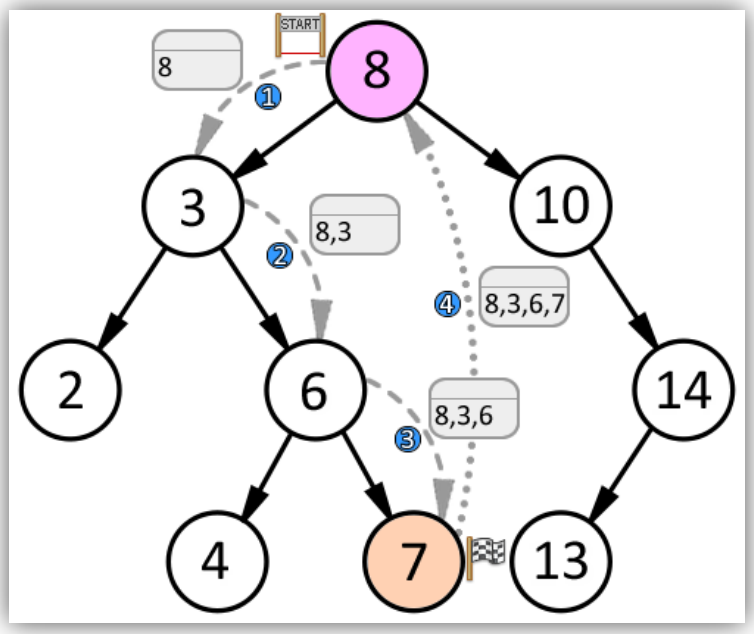


Abbildung 6-6: Weiterleitung des Kontext-Objekts

Das Beispiel in der oben gezeigten Abbildung stellt den Ablauf eines Algorithmus dar, der beim Root-Node startet und auf dem Leaf-Node mit dem Key 7 terminiert. Dabei wird das Kontext-Objekt jeweils von `Node` zu `Node` weitergereicht. Anschliessend ruft der Leaf-Node auf dem Root-Node eine Callback-Methode auf. Je nach Algorithmus können bestimmte Properties mit Informationen gefüllt werden.

Im Kontext-Objekt wird nur das Property `VisitOrder` als Beispiel dargestellt. Bei der Weiterleitung des Kontext-Objektes trägt jeder `Node` seinen Key resp. Adresse in das Kontext-Objekt ein. Ist der Algorithmus auf dem Leaf-Node mit dem Key 7 angelangt, ruft dieser eine Callback-Methode auf dem Root-Node auf und übergibt diesem das vollständig gefüllte Kontext-Objekt. Letzteres ist im vierten Schritt in der Abbildung dargestellt. Der Root-Node hat nun die Möglichkeit alle benötigten Informationen aus dem Kontext-Objekt auszulesen und zu verwerten. Im Folgenden soll der Mechanismus des ACTs genauer betrachten werden. Es wird darum auf den vierten Schritt in der Abbildung vertieft eingegangen.

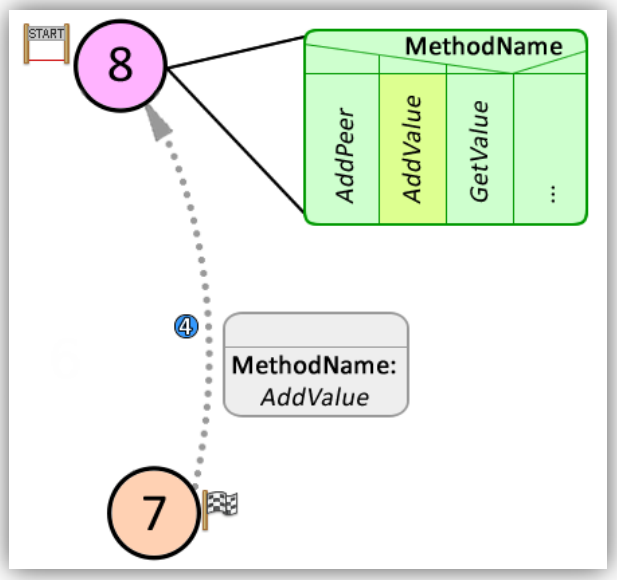


Abbildung 6-7: Callback-Methodenaufruf anhand ACT

Der vierte Schritt zeigt den Callback-Methodenaufruf auf dem Root-Node. Je nach Inhalt des ACTs wird eine andere Callback-Methode aufgerufen. Im Beispiel hat das Property `MethodeName` der Klasse `RequestPeerContextInfo` den Wert `AddValue` vom Aufzähltyp `SearchResultMethodNames`. Die Referenz auf das Objekt der Klasse `RequestPeerContextInfo` wird in einer Instanz der Klasse `SearchContext` oder `SyncContext` gehalten. Anhand dieses Werts entscheidet der Root-Node welche Callback-Methode schlussendlich aufgerufen wird.

6.1.3 TestAndVisualization-Package

Das dritte Package beinhaltet alle Klassen, welche für das Testen der Verbindungen zwischen `Node`s und für die Visualisierung des Trees verwendet werden.

6.1.3.1 Klassendiagramm

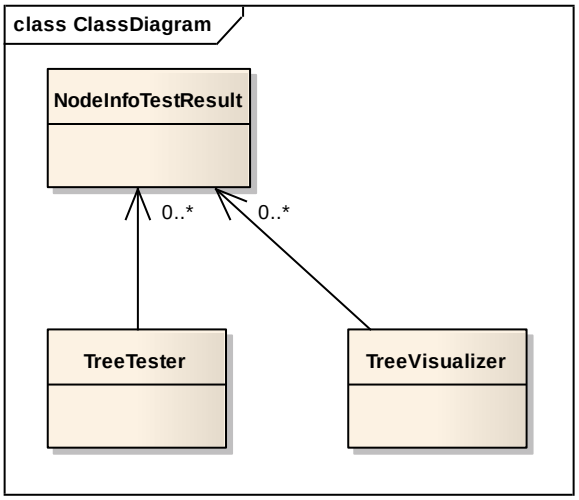
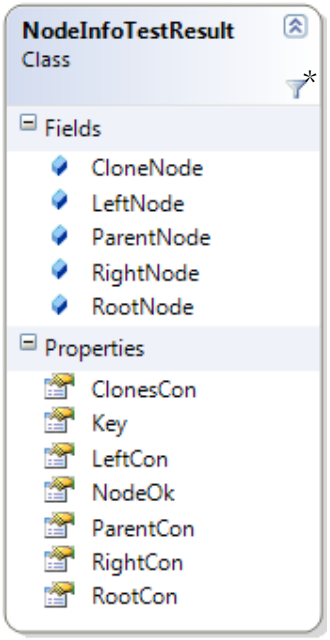


Abbildung 6-8: Klassendiagramm - Test- und Visualisierungsklassen

6.1.3.2 Klassenbeschreibung

6.1.3.2.1 NodeInfoTestResult



*Methodenliste entfernt

Beschreibung

Die Klasse `NodeInfoTestResult` ist eine Sammelklasse für verschiedene Informationen, die für das Testen und Zeichnen des Trees benötigt werden.

Verwendungszweck

Sämtliche Beziehungen von einem `Node` zu anderen `Node`s werden in dieser Klasse gesammelt. Ferner können diese Beziehungen als korrekt oder inkorrekt markiert werden. Anhand dieser Informationen kann festgestellt werden, ob der Tree fehlerfrei aufgebaut wurde. Diese Informationen werden auch bei der grafischen Darstellung des Trees miteinbezogen.

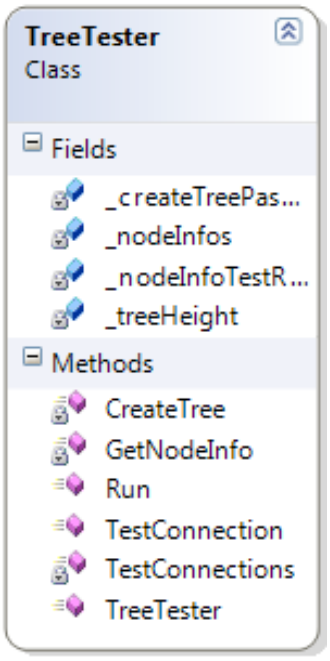
Merkmale

In den Fields werden die URLs der `Peer`s des entsprechenden `Node` in einer Liste gespeichert.

Bis auf das Property `Key` sind alle Properties vom Typ `bool` und geben über die Korrektheit der Beziehungen des entsprechenden `Node`s Bescheid.

Tabelle 6-19: Klassenbeschreibung - NodeInfoTestResult

6.1.3.2.2 TreeTester



Beschreibung

Die Klasse `TreeTester` speichert den Tree in einer Liste ab. D.h. die `NodeTestInfo`s jedes `Node`s werden an der korrekten Position im Array gespeichert. Sie testet des Weiteren die Beziehungen zwischen den `Node`s auf ihre Korrektheit.

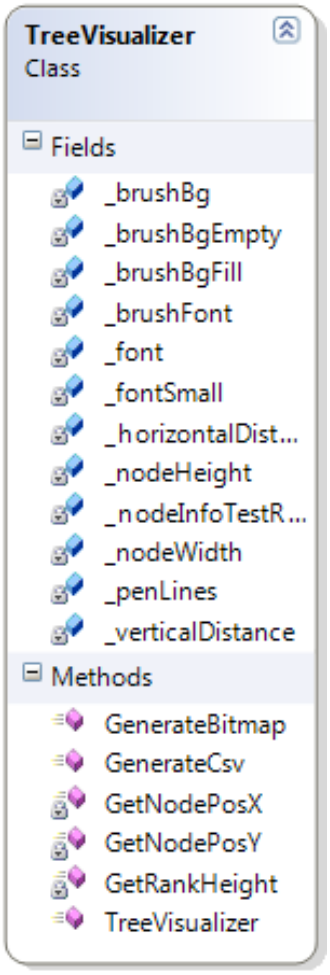
Verwendungszweck

Die Klasse wird benötigt, um die Beziehungen zwischen den `Node`s auf ihre Korrektheit zu prüfen. Diese Informationen werden in den Instanzen der Klasse `NodeInfoTestResult` gespeichert und später grafisch dargestellt.

	Merkmale Die Objekte der Klasse `NodeInfoTestResult` werden an klar vorgegebenen Positionen im Array abgespeichert. Die Berechnung dieser Positionen bzw. Indexe erfolgt anhand folgender Berechnungsformeln. Root-Node Index: 1 Parent-Node Index: x Left-Node Index: 2 • x Right-Node Index: 2 • x + 1 Vergleiche dazu [Abbildung 6-30: Abspeichern eines Trees in eine tree-komfortable Datenstruktur].

Tabelle 6-20: Klassenbeschreibung - TreeTester

6.1.3.2.3 TreeVisualizer



	Beschreibung Die Klasse `TreeVisualizer` stellt die gesammelten Daten über den Tree grafisch dar.
	Verwendungszweck Der Aufbau des Trees ist oft schwer zu erkennen. Eine Prüfung, ob der Tree korrekt aufgebaut wurde ist daher sehr aufwendig. Um dem Benutzer eine einfache Möglichkeit zu geben die Struktur des Trees zu prüfen, erstellt die Klasse `TreeVisualizer` eine Grafik. Diese Grafik zeigt wie der Tree effektiv aufgebaut wurde. So werden die Verbindungen der `Node`s, der Key der `Node`s und die Anzahl Clones pro `Node` dargestellt. Fehler, die bereits durch das System erkannt wurden, werden ebenfalls in der Grafik gezeigt. Siehe [Abbildung 6-32: Grafische Darstellung eines Trees]. Nebst der Grafik können alle relevanten Informationen des Trees in eine CSV-Datei exportiert werden.
	Merkmale Die grafische Abbildung wird in Form eines Objekts der .Net-Klasse `Bitmap` zurückgeliefert. Die CSV-Datei wird immer unter dem Namen „Tree.csv“ im Ordner der laufenden Applikation gespeichert.

Tabelle 6-21: Klassenbeschreibung - TreeVisualizer

6.1.3.3 Zusammenspiel

Folgende Abbildung demonstriert das Zusammenspiel der Test- und Visualisierungsklassen. Nachfolgend wird der Ablauf vom Sammeln der Informationen über die `Node`s bis hin zur grafischen Darstellung dieser Daten beschrieben.

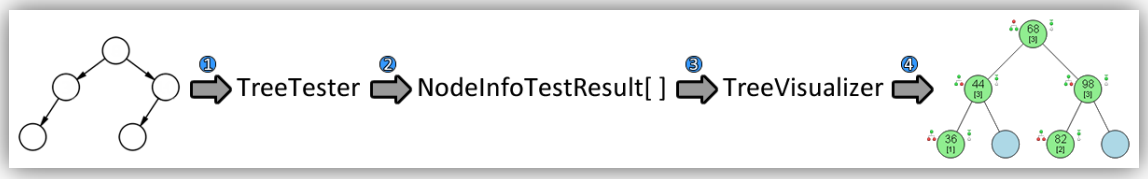


Abbildung 6-9: Zusammenspiel der Test- und Visualisierungsklassen

Der Ablauf wird anhand folgender vier Schritte erklärt.

1. Zu Beginn müssen die Informationen jedes einzelnen `Node`s gesammelt und der Klasse `TreeTester` übergeben werden.
2. Die Klasse `TreeTester` wertet die erhaltenen Informationen aus und speichert diese in ein Array von Objekte der Klasse `NodeInfoTestResult` ab.
3. Das komplette Array wird der Klasse `TreeVisualizer` für das Zeichnen übergeben.
4. Die Klasse `TreeVisualizer` liefert als Ergebnis die grafische Repräsentation des Trees.

6.1.4 Utilities-Package

Das vierte Package besteht aus Klassen, welche unterstützende Methoden beinhalten. Da keine Zusammenhänge zwischen den Klassen existieren, wird auf ein Kapitel „Zusammenspiel“ verzichtet.

6.1.4.1 Klassendiagramm

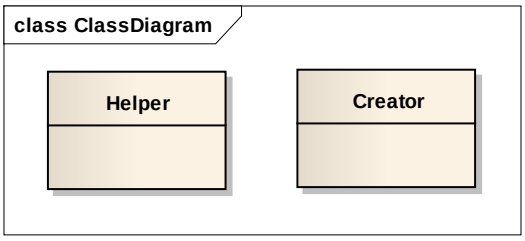
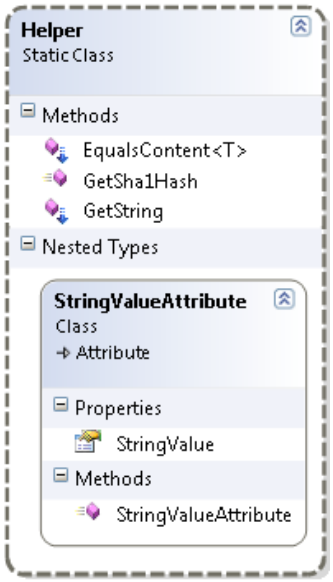


Abbildung 6-10: Klassendiagramm - Unterstützungsklassen

6.1.4.2 Klassenbeschreibung

6.1.4.2.1 Helper



Beschreibung

Die Klasse `Helper` definiert eine kleine Anzahl von unterstützenden Methoden und die verschachtelte Klasse `StringValueAttribute`.

Verwendungszweck

Die Klasse wird unter anderem benutzt um Erweiterungsmethoden zu definieren. Die Methode `EqualsContent()` ist eine Erweiterungsmethode der .Net-Klasse `List`. Sie erlaubt das Prüfen der Inhalte einer Liste auf deren Gleichheit. Die Erweiterungsmethode `GetString()` erweitert die .Net-Klasse `Enum`. Sie liefert einen `string` eines Wertes in einem Aufzählтyp. Ebenfalls enthält die Klasse die Methode `GetSha1Hash()` welche den SHA1-Hashwert eines `string`s berechnet.

Merkmale

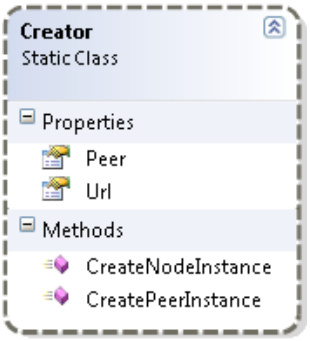
Die verschachtelte Klasse `StringValueAttribute` erbt von der .Net Klasse `Attribute`. D.h. die Klasse kann als Attribut in einer Klasse verwendet werden. Der folgende Code illustriert die Verwendung der Attribut-Klasse.

```
public enum SearchResultMethodNames
{
    [StringValueAttribute("SearchResultAddPeer")]
    AddPeer,
    [StringValueAttribute("SearchResultAddPeerAsClone")]
    AddPeerAsClone,
    …
}
```

Das Attribut `StringValueAttribute` wird in Zusammenhang mit dem Aufzählтyp `SearchResultMethodNames` und der Erweiterungsmethode `GetString()` verwendet.

Tabelle 6-22: Klassenbeschreibung - Helper

6.1.4.2.2 Creator



Beschreibung

Die Klasse `Creator` ist eine statische Klasse und stellt eine Factory dar.

Verwendungszweck

Die Klasse wird für das Erstellen von Instanzen vom Type `IPeer` und `INode` verwendet.

Merkmale

Für das Erstellen eines `IPeer`s auf eine bestimmte Endpoint-Adresse wird die .Net WCF-Klasse `ChannelFactory` benutzt. Die Methode `CreateNodeInstance()` liefert je nach Kontext eine Instanz von der Klasse `Node` oder `NodeNull` als Rückgabewert.

Tabelle 6-23: Klassenbeschreibung - Creator

6.2 Szenarios

Im Folgenden Kapitel werden die Funktionalitäten der DHT in Form von Szenarios beschrieben. Die Interaktionen zwischen den einzelnen Klassen werden mit Sequenzdiagrammen dargestellt. Das Kapitel soll dem besseren Verständnis für die wichtigsten Abläufe innerhalb der DHT dienen.

6.2.1 Value hinzufügen

Soll ein neues Value im Tree abgespeichert werden, so muss zuerst der für diesen Key zuständige `Node` mittels einer Suchanfrage ermittelt werden. Danach kann das Value auf allen Clones des zuständigen `Node`s abgespeichert werden.

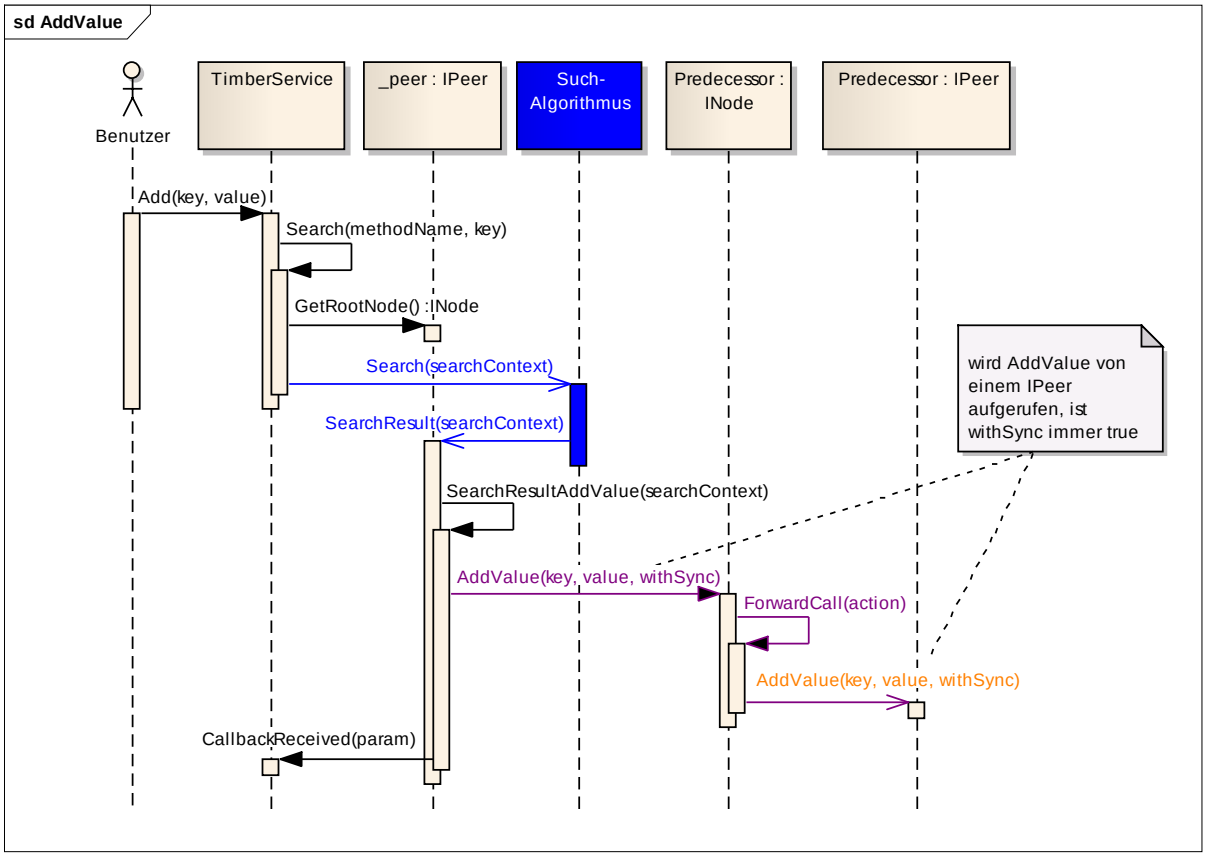


Abbildung 6-11: Sequenzdiagramm - Value hinzufügen

Der Benutzer kann über die `TimberService`-Schnittstelle die Methode `Add()` aufrufen. Der `TimberService` startet daraufhin eine Suchanfrage mittels der Methode `Search()` auf dem über die Methode `GetRootNode()` ermittelten Root-Node. Die dadurch ausgelösten Methoden werden im Suchalgorithmus zusammengefasst welcher im Kapitel [6.2.1.1 Suchalgorithmus] beschrieben ist. So wird die Übersicht beibehalten. Nach der Terminierung des Suchalgorithmus ruft die Methode `SearchResult()` die spezifische Callback-Methode `SearchResultAddValue()` auf. Dies wird durch ein ACT, welches sich im `SearchContext` befindet, ermöglicht. Der Methodenaufruf `AddValue()` auf dem `Node` muss an einen `Peer` weitergeleitet werden. Eine solche Weiterleitung mit Hilfe der Methode `ForwardCall()` ist in allen Sequenzdiagrammen violett hervorgehoben und im Kapitel

[6.2.1.2 Weiterleitung] beschrieben. Der Aufruf der Methode `AddValue()` auf dem `Peer` läuft nach dem im Kapitel [6.2.1.3 Synchronisationsalgorithmus] beschriebenen Synchronisationsalgorithmus ab. Zum Schluss wird ein Event vom Delegate-Typ `CallbackHandle` mit der Methode `CallbackReceived()` geworfen.

6.2.1.1 Suchalgorithmus

Jeder Key wird an einem klar definierten `Node` zugewiesen. Der `Node`, welcher den gesuchten Key verwaltet, wird mit Hilfe des Suchalgorithmus gefunden. Der verantwortliche `Node` entspricht dem Predecessor-Node des `NodeNull`s, auf welchem der Suchalgorithmus terminiert. Da der Predecessor-Node eines `Peer`s nicht gespeichert wird, muss dieser während der Suche nach dem Key ermittelt werden. Ist der Suchvorgang bei einem `NodeNull` angelangt, kann nun aus dem Kontext-Objekt der Predecessor-Node ausgelesen und als Resultat dem Aufrufer mitgeteilt werden.

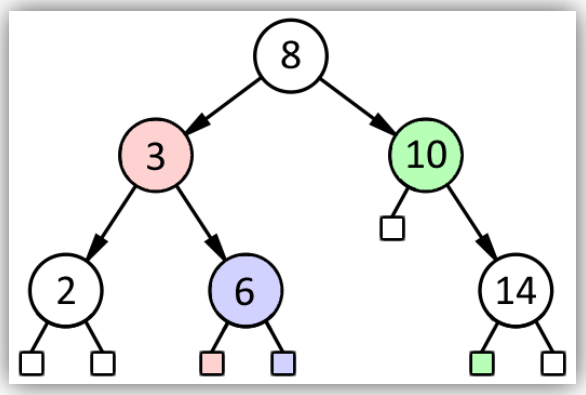


Abbildung 6-12: Predecessor-Node

Im folgenden Beispiel wird nach dem `Node`, welcher für die Verwaltung des Keys 5 zuständig ist, gesucht. Der Suchalgorithmus terminiert auf dem linken `NodeNull` des `Node`s mit dem Key 6. Der Predecessor-Node dieses `NodeNull`s ist der `Node` mit dem Key 3. Dieser `Node` entspricht dem gesuchten `Node`. Die betroffenen Elemente sind in der Abbildung rot hervorgehoben.

Ein weiteres Beispiel ist die Suche mit dem Key 7. Bei dieser Suche sind die blauen Elemente betroffen. Die grün markierten Elemente zeigen ein Beispiel auf, in welchem mit dem Key 11 gesucht wird.

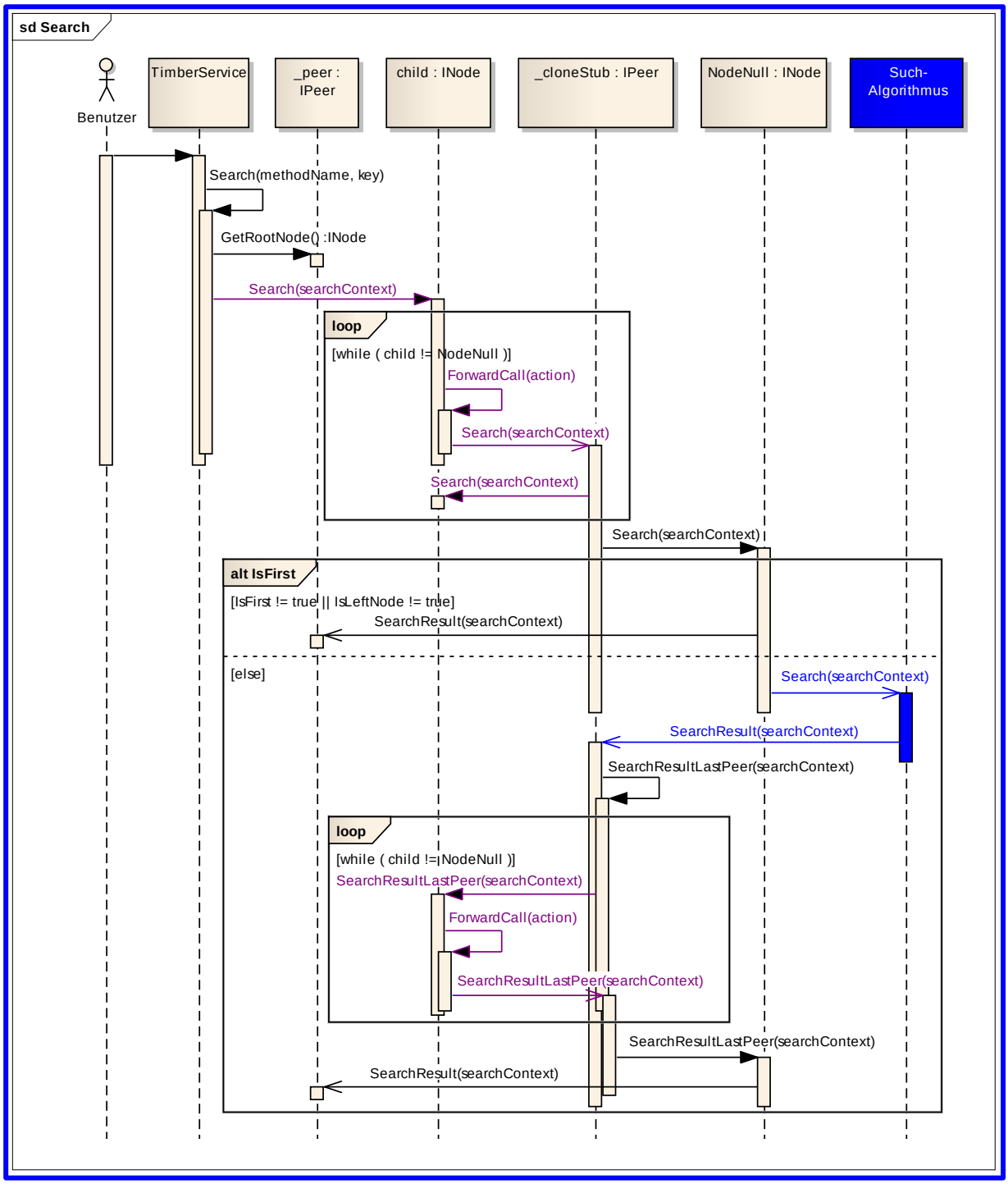


Abbildung 6-13: Sequenzdiagramm - Key suchen

Der Benutzer löst durch den Aufruf gewisser Methoden über die `TimberService`-Schnittstelle den Suchalgorithmus auf. Worauf der lokale `Peer` auf seinem Root-Node, welcher mit der Methode `GetRootNode()` ermittelt wurde, die Methode `Search()` auslöst. Der Suchaufruf wird nun an einen `Peer` des Root-Nodes weitergeleitet. Dieser wird anhand der `ForwardCall()`-Methode gewählt. Dadurch wird der Suchalgorithmus auf dem Tree gestartet. Der gewählte `Peer` entscheidet an welchen Child-Node er den Aufruf weiterleiten muss. Da sich der Suchalgorithmus auf einem `Node` befinden, muss nun wieder einen `Peer` gewählt werden, an den der Aufruf weitergeleitet werden kann. Dieser Vorgang wird solange wiederholt, bis ein Child-Node vom Typ `NodeNull` ist. Wird die

Methode `Search()` auf einem `NodeNull` ausgeführt, ruft dieser die allgemeine Callback-Methode `SearchResult()` auf dem anfragenden `Node` auf. Über den `SearchContext` kann nun der `Node`, welcher für die Verwaltung des gesuchten Keys zuständig ist, ermittelt werden. Zusätzlich kann über den Kontext die spezifische Callback-Methode identifiziert und anschliessen aufgerufen werden.

Der oben beschrieben Ablauf stellt den Normalfall dar. Ist nun aber der gesuchte Key kleiner als der erste `Node` der Inorder-Traversierung, so muss dieser auf dem letzten `Node` der Inorder-Traversierung gesucht werden. Der gefundene `Node` kann dadurch als Predecessor-Node des ersten `Node`s angegeben werden. Um diesen letzten `Node` zu ermitteln, wird der Suchalgorithmus vom `NodeNull` aus nach dem maximalen Key-Wert über die Methode `Search()` gestartet. Der `SearchContext` der Suche wird mit dem entsprechenden ACT versehen, damit bei der Beendigung des Suchalgorithmus die passende Methode aufgerufen werden kann. So wird nun die Anfrage über die Methode `SearchResultLastPeer()` den Child-Nodes bis zu einem `NodeNull` weitergeleitet. Im `NodeNull` wird der `SearchContext` mit dem neu ermittelten Predecessor-Node angepasst. Somit kann auf dem `Peer`, welcher die Suche startete, die normale Callback-Methode `SearchResult()` aufgerufen werden. Dieses Spezialverhalten entspricht der im Kapitel [4.3 Zuständigkeitsbereich eines Nodes] beschriebene Ausnahme betreffendem Zuständigkeitsbereich.

6.2.1.2 Weiterleitung

Beim Aufruf von bestimmten Methoden auf einem `Node`, findet eine Weiterleitung des Aufrufs auf einen Clone des `Node`s statt. Die Weiterleitung wird anhand der Methode `AddValue()` erläutert.

```
public void AddValue(string key, string value, bool withSync)
{
      ForwardCall(peer => peer.AddValue(key, value, withSync));
}
```

Tabelle 6-24: Aufruf der Methode ForwardCall

Eine Weiterleitung erfolgt über den Aufruf der Methode `ForwardCall()`. Diese bestimmt einen `Peer`, an den der Aufruf weitergeleitet wird.

```
private void ForwardCall(Action<IPeer> action)
{
	foreach (IPeer peer in _cloneStubs)
	{
		try
		{
			//Call the method to forward
			action(_cloneStubs[GetNextIndex()]);
			return;
		}
		catch (Exception)
		{
			//Code which define the behavior when peer wasn't reachable.
		}
	}
}
```

Tabelle 6-25: Weiterleitung eines Aufrufs

Der Aufruf wird an einen durch die Methode `GetNextIndex()` bestimmten `Peer` weitergeleitet. Ist dieser `Peer` jedoch z.B. wegen eines unkontrollierten Ausfalls nicht erreichbar, wird versucht die Anfrage an einen anderen `Peer` weiterzuleiten. Der unkontrollierte Ausfall einzelner `Peer`s hat somit keinen negativen Einfluss auf das korrekte Verhalten des Trees.

6.2.1.3 Synchronisationsalgorithmus

Dadurch, dass ein `Node` durch mehrere Clones repräsentiert wird, müssen die Clones identisch gehalten werden. Bestimmte Methoden ändern gewisse Eigenschaften eines Clones. Sobald eine solche Methode aufgerufen wird, muss diese Änderung mit den anderen Clones synchronisiert werden. Die Methoden, welche eine Synchronisation auslösen, werden im Folgenden Synchronisationsmethoden genannt. Der Synchronisationsalgorithmus wird anhand der Synchronisationsmethode `AddValue()` aufgezeigt.

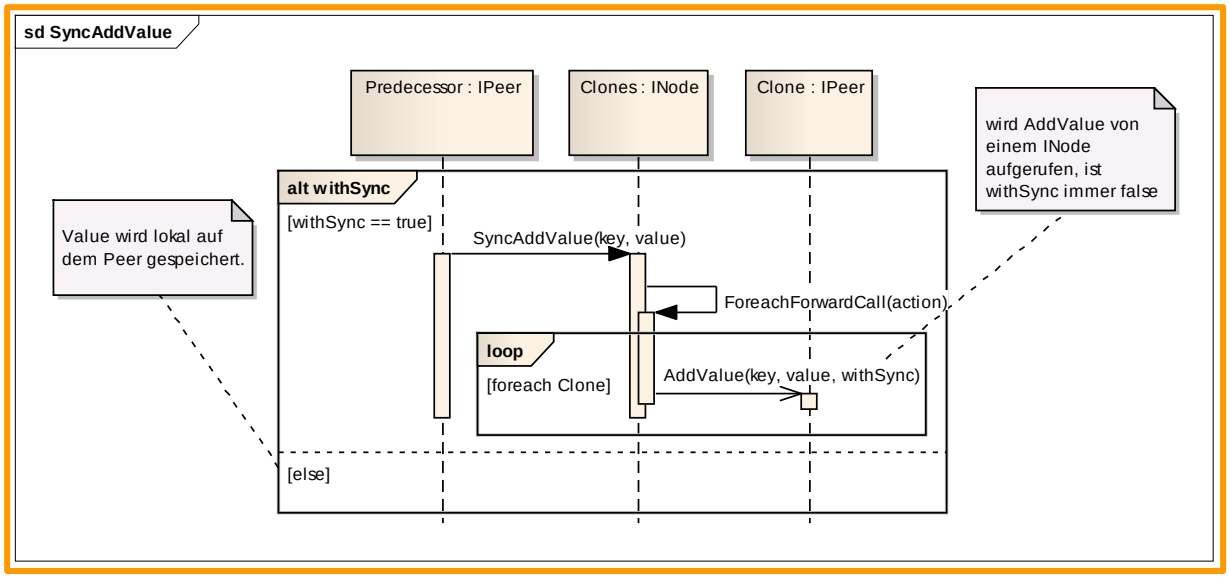


Abbildung 6-14: Sequenzdiagramm - Synchronisationsalgorithmus

Durch den Methodenaufruf `AddValue()` auf einem `Peer` und einem auf `true` gesetzten `withSync`-Parameter, wird die Methode `SyncAddValue()` aufgerufen. Die Methode `ForeachForwardCall()` ruft auf jedem erreichbaren Clone die Methode `AddValue()` erneut auf. Jedoch hat der `withSync`-Parameter nun den Wert `false`. Dadurch wird das Value auf jedem Clone lokal abgespeichert.

Der Parameter `withSync` hat immer den Wert `true`, sobald die Methode `AddValue()` von einem `Peer` aufgerufen wird. Dies ist zwingend um eine Synchronisation unter den Clones auszulösen. Wird die Methode von einem `Node` aufgerufen, ist der Parameter immer auf `false`. So wird sichergestellt, dass keine Endlosschleife bei der Synchronisation entsteht. Die beschrieben Abhängigkeiten für den Parameter `withSync` gelten auch in den folgenden Sequenzdiagrammen.

Ein solcher Synchronisationsalgorithmus wird bei sämtlichen Änderungen einer `Peer`-Eigenschaft angewendet. In den folgenden Sequenzdiagrammen wird der Synchronisationsalgorithmus wegen Übersichtsgründen nicht mehr in diesem Detailgrad aufgezeigt aber orange markiert.

6.2.2 Value entfernen

Erteilt der Benutzer den Befehl ein Key-Value-Pair zu löschen, wird eine Suche nach dem entsprechenden Key durchgeführt. Falls der Key während dem Suchalgorithmus gefunden wird, kann das Key-Value-Pair unmittelbar gelöscht werden. Der Löschvorgang muss mit den Clones des `Peer`s synchronisiert werden.

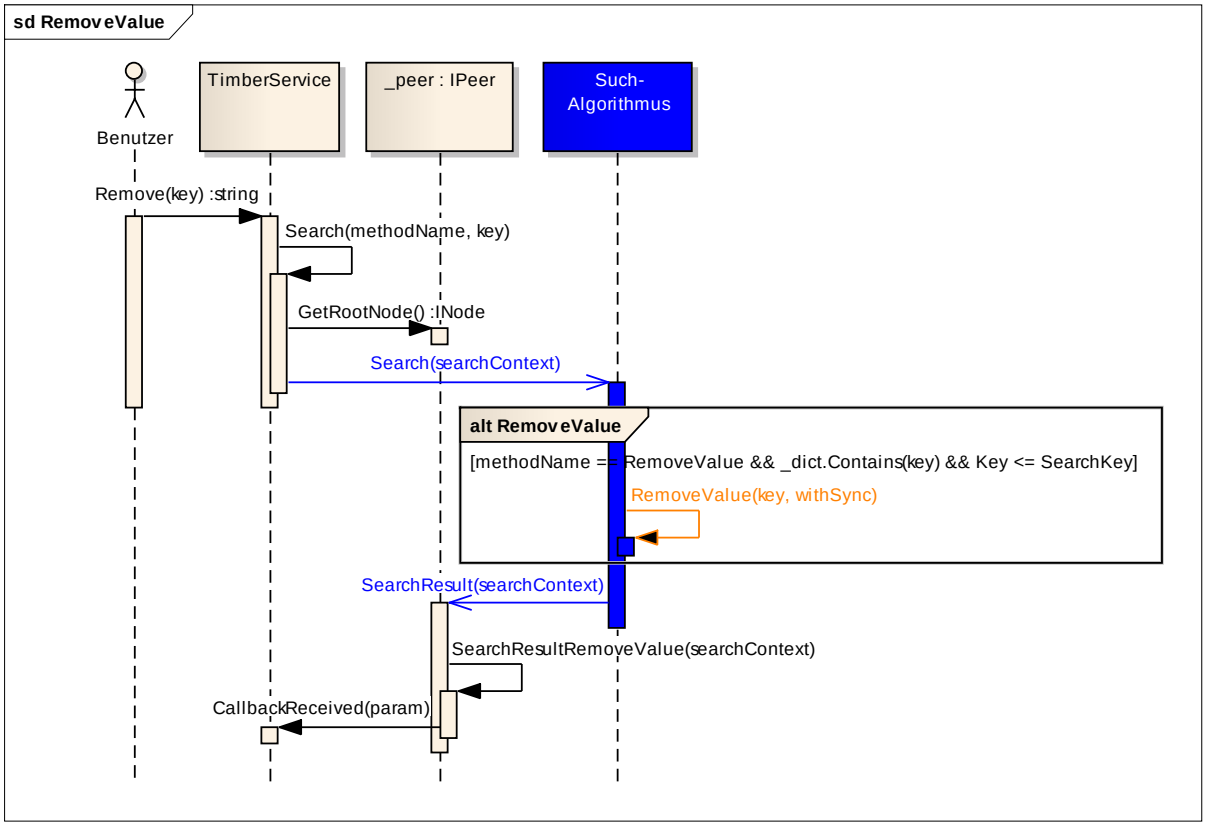


Abbildung 6-15: Sequenzdiagramm - Value entfernen

Der Benutzer kann mit der Methode `Remove()` ein Key-Value-Pair entfernen. Dieses Ereignis löst auf dem Root-Node die Methode `Search()` und somit den Suchalgorithmus auf. Während dem Ablauf des Suchalgorithmus werden bereits alle möglichen Kandidaten nach dem zu löschenden Key durchsucht. Sobald ein `Peer` den gesuchten Key besitzt d.h. die Alternative „RemoveValue“ trifft zu, löst dieser direkt den Synchronisationsalgorithmus über die Methode `RemoveValue()` aus. Dadurch wird das entsprechende Key-Value-Pair auf allen Clones gelöscht. Wie bereits erwähnt, wird diese Synchronisation nicht weiter erläutert.

Wie gewohnt, wird am Schluss des Suchalgorithmus die Callback-Methode `SearchResult()` aufgerufen, welche wiederum die spezifische Methode `SearchResultRemoveValue()` anhand des ACTs aufruft. Das Beenden des Suchalgorithmus wird mit dem Event `CallbackReceived()` am `TimberService` mitgeteilt.

6.2.3 Value abfragen

Sobald ein Key eines Values im Tree abgefragt wird, prüft jeder möglicher Kandidat, welcher für den gesuchten Key zuständig sein könnte, ob er den gesuchten Key verwaltet. Wurde der Key nicht gefunden, wird die Suche fortgesetzt. Die möglichen Kandidaten werden geprüft, damit bei einem frühzeitigen Treffer der Suchalgorithmus bereits terminiert werden kann. Dadurch endet nicht jede Suchanfrage erst in einem Leaf-Node und spart somit teure Netzwerkkommunikation.

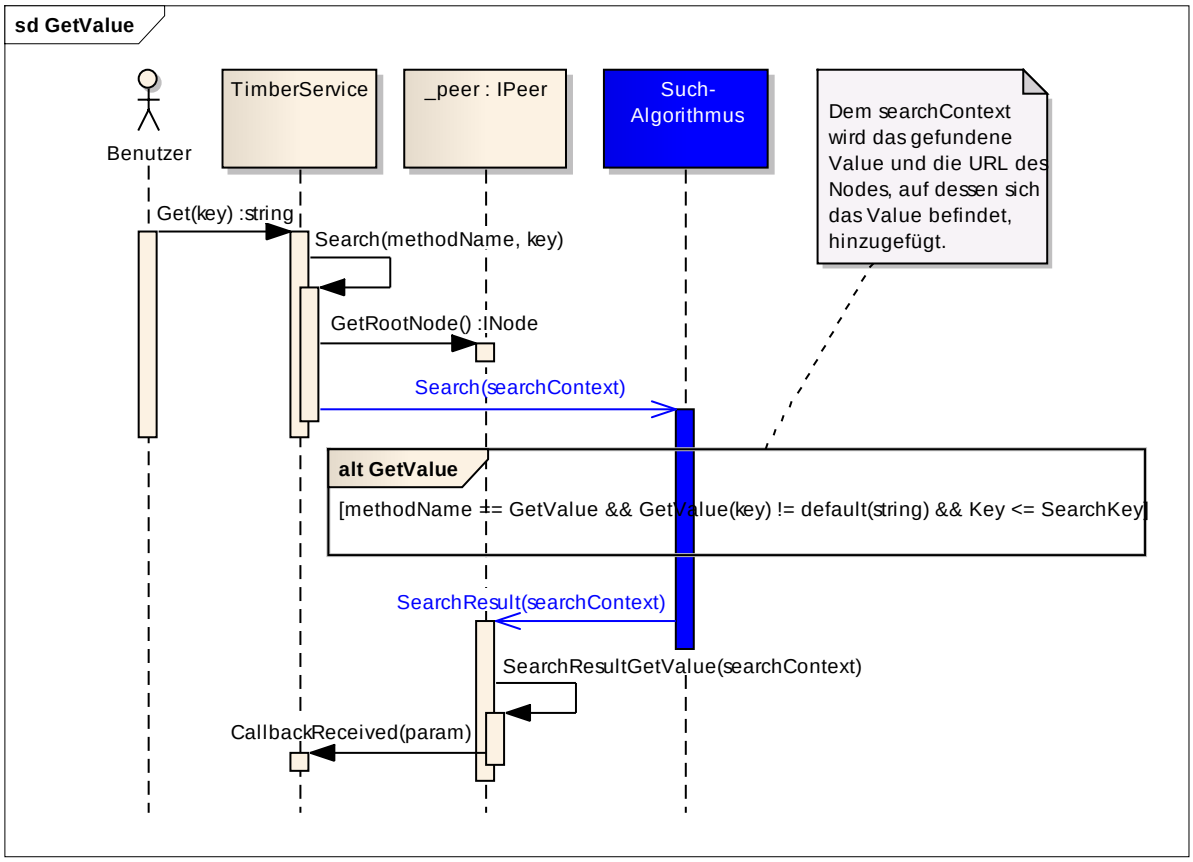


Abbildung 6-16: Sequenzdiagramm - Value abfragen

Der Benutzer startet durch den Aufruf der Methode `Get()` den Suchalgorithmus mit dem entsprechenden ACT. Während des Suchalgorithmus werden die möglichen Kandidaten bereits durchsucht. Wird das Value auf einem Kandidaten gefunden d.h. die Alternative „GetValue“ trifft zu, wird es dem `SearchContext` hinzugefügt. Dieser wird bei dem Callback-Aufruf `SearchResult()` als Parameter mitgegeben. Sofern das gesuchte Value nicht gefunden wird, terminiert der Suchalgorithmus auf einem Leaf-Node. Dies hat zur Folge, dass der `SearchContext` mit einem leeren Value zurückgegeben wird. Das Value wird mittels Event `CallbackReceived()` zurückgegeben.

6.2.4 Peer starten

Der `TimberService` kann auf zwei unterschiedliche Arten gestartet werden. Gibt der Benutzer beim Start des `TimberService`s die URL eines Bootstrapping-Nodes an, so wird der `Peer` einem bereits existierenden Netz hinzugefügt. Ohne Angabe einer URL wird der `Peer` zu einem Root-Node eines neuen Trees.

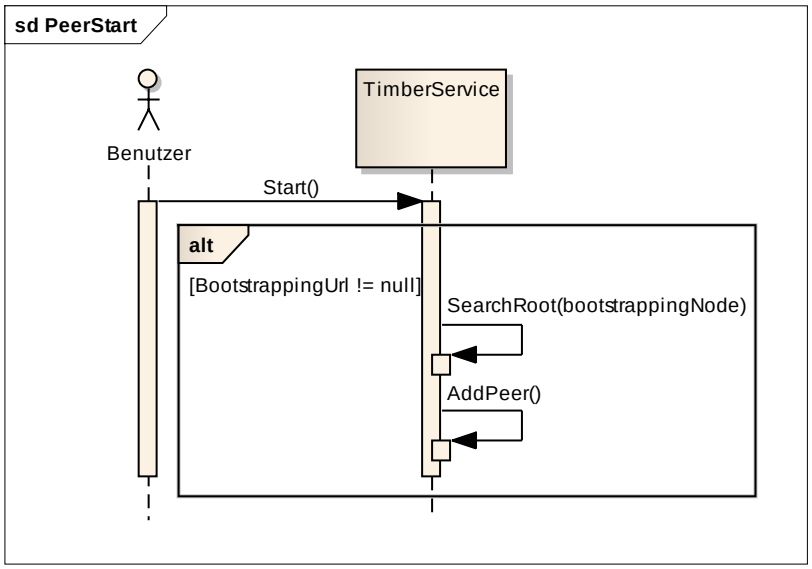


Abbildung 6-17: Sequenzdiagramm - Peer-Start

Wird der neue `Peer` einem existierendem Netz hinzugefügt, läuft das Szenario [6.2.5 Peer hinzufügen] ab. Dieses beschreibt die beiden Methoden `SearchRoot()` und `AddPeer()`.

6.2.5 Peer hinzufügen

Damit sich ein neuer `Peer` in den Tree einfügen kann, muss ein `Node`, welcher sich bereits im Netz befindet, bekannt sein. Dieser fungiert als Bootstrapping-Node. Über den Bootstrapping-Node kann nun der Root-Node ermittelt werden. Damit sich der neue `Peer` im Tree einordnen kann, muss dessen zukünftigen Parent-Node gefunden werden. Dieser wird mit Hilfe des Suchalgorithmus ermittelt, welcher auf dem Root-Node gestartet wird. Um die Diagramme übersichtlich zu halten, wird das Suchen und Hinzufügen getrennt beschrieben.

6.2.5.1 SearchRoot

Im Wesentlichen besteht die Methode `SearchRoot()` darin, dessen Aufruf so lange dem Parent-Node weiterzuleiten bis der Root-Node erreicht wurde. Der gefundene Root-Node wird auf dem neu hinzukommenden `Peer` gesetzt.

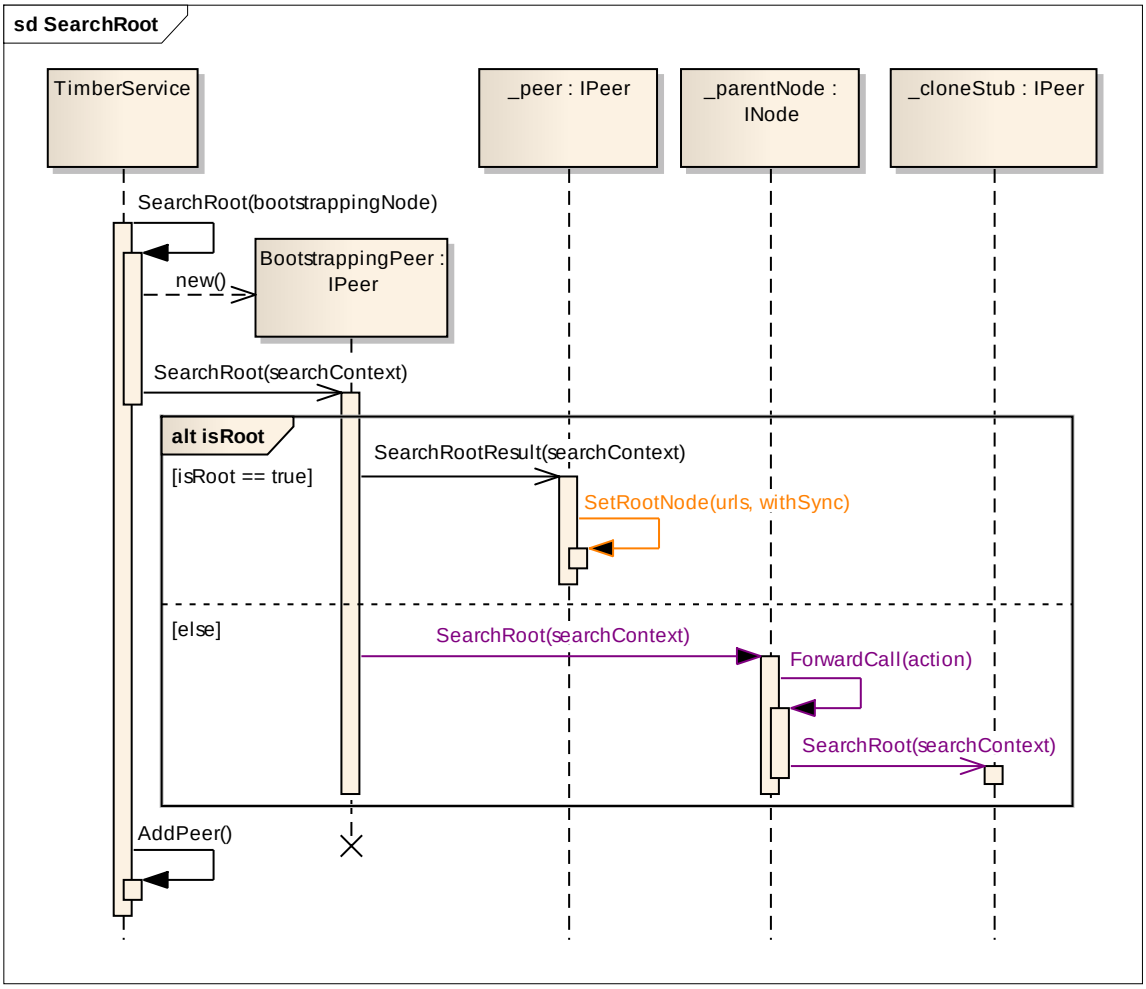


Abbildung 6-18: Sequenzdiagramm - SearchRoot

Aus der vom Benutzer angegebenen URL wird ein Stub von einem `Peer` im Tree erstellt. Der über den Stub angesprochene `Peer` dient als Bootstrapping-Node. Auf diesem wird die Methode `SearchRoot()` aufgerufen. Trifft die Alternative „isRoot“ nicht zu d.h. der aktuelle `Peer` ist kein Clone des Root-Nodes, muss der Aufruf der Methode `SearchRoot()` bis zum Root-Node

weitergeleitet werden. Hierzu wird die Methode `SearchRoot()` auf dem Parent-Node aufgerufen. Anhand der Methode `ForwardCall()` wird ein `Peer` gewählt. Auf diesem wird die Methode `SearchRoot()` ausgeführt. Ab hier wiederholt sich der Ablauf bis ein `Peer` gefunden wurde, welcher ein Clone des Root-Nodes ist. D.h. die Alternative „isRoot“ wird mehrfach aber auf unterschiedlichen `Peer`s ausgewertet. Wurde ein Clone des Root-Nodes erreicht bzw. die Alternative trifft zu, wird die Callback-Methode `SearchRootResult()` auf dem neu hinzukommenden `Peer` aufgerufen. Dieser setzt sich mit Hilfe der Synchronisationsmethode `SetRoot()` den gefunden `Node` als Root-Node. Zum Schluss wird die Methode `AddPeer()` aufgerufen. Diese wird detailliert im nächsten Abschnitt beschrieben.

6.2.5.2 AddPeer

Prinzipiell kann der neue `Peer` auf zwei unterschiedliche Arten dem Tree hinzugefügt werden. Bei der ersten Art wird ein neuer `Node` erzeugt. Dies beinhaltet eine Aufteilung des Zuständigkeitsbereichs der Hash-Werte des Predecessor-Nodes und die Übernahme von bereits vorhandenen Values. Bei der Übernahme werden die in den neuen Zuständigkeitsbereich fallenden Values vom Predecessor-Node auf den soeben erstellten `Node` verschoben. Zusätzlich muss geprüft werden, ob die minimale Anzahl Clones auf diesem `Node` eingehalten wird. Ist dies nicht der Fall, zieht der neue `Peer` die fehlenden `Peer`s zu sich hinunter. Die zweite Art sieht vor, dass der `Peer` zu einem Clone eines bereits existierenden `Node`s wird. In diesem Fall müssen sämtliche Attribute des `Node`s kopiert werden.

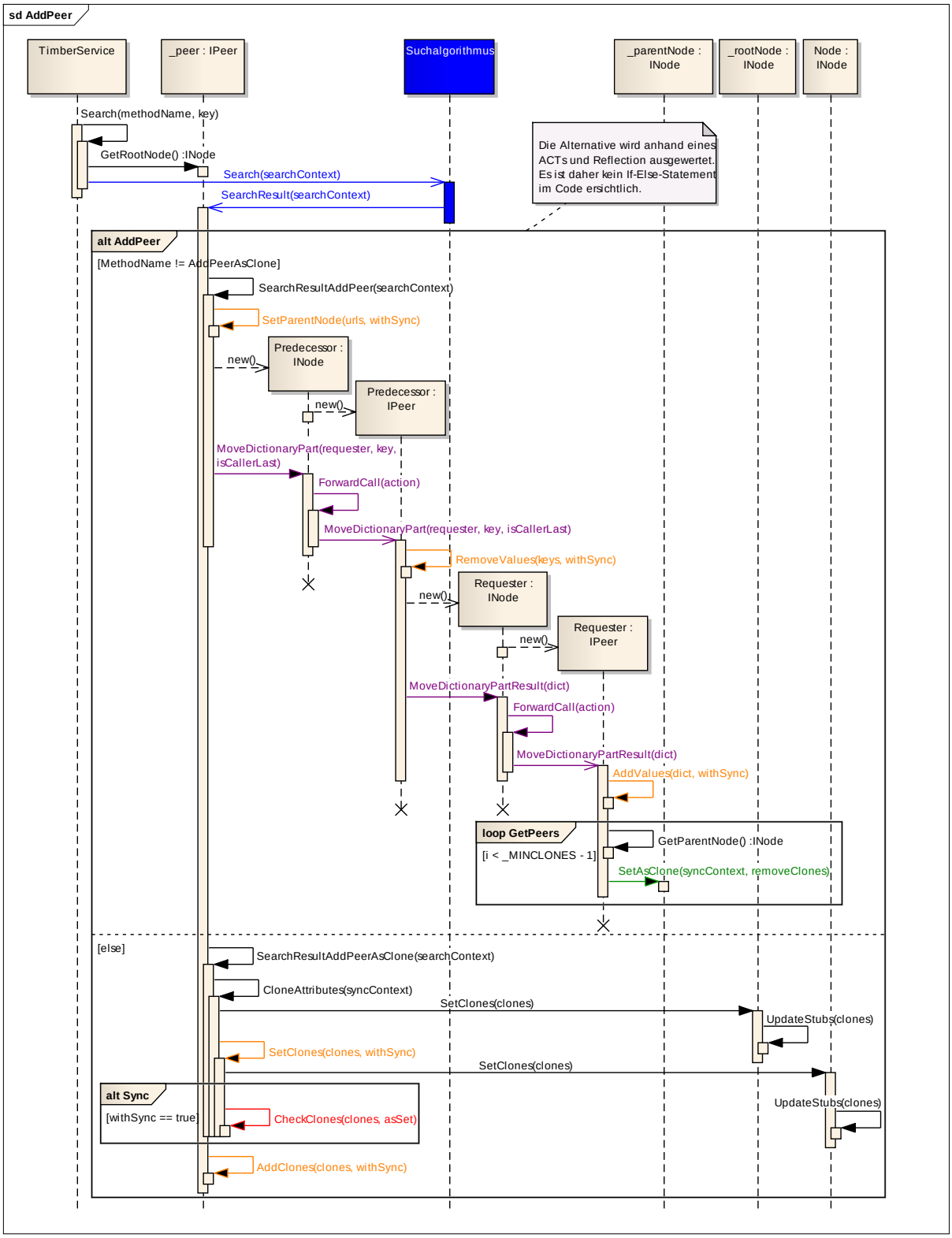


Abbildung 6-19: Sequenzdiagramm - Peer hinzufügen

Beim Start des `TimberService` wird die Methode `Search()` auf dem, über die Methode `GetRootNode()` ermittelten Root-Node aufgerufen. Um den Parent-Node des neuen `Peer`s `_peer` zu finden, wird nach dessen Key gesucht. Wird die maximal erlaubte Anzahl Clones überschritten, muss der neue `Peer` als `Node` und nicht als neuer Clone hinzugefügt werden. Diese Unterscheidung wird

durch das Anpassen des ACTs festgehalten. Durch das veränderte ACT ist die Methode `SearchResult()` in der Lage, die spezifische Callback-Methode aufzurufen.

Trifft die Bedingung der Alternative „AddPeer“ zu, so muss der neue `Peer` als `Node` hinzugefügt werden. Für das Hinzufügen werden die beiden Methoden `SearchResultAddPeer()` und `SetParentNode()` der Reihe nach aufgerufen. Da der `Peer` neu ist und daher noch keine Clones besitzt, wird die Änderung nicht synchronisiert. D.h. der Parameter `withSync` hat den Wert `false`. Der neue `Node` muss die Values deren Key grösser oder gleich gross sind wie der Eigene, von seinem Predecessor-Node übernehmen und verwalten. Dazu wird die Methode `MoveDictionaryPart()` auf dem Predecessor-Node aufgerufen. Mit Hilfe der Methode `ForwardCall()` wird ein `Peer` des `Node`s ausgewählt, auf welchem die Methode `RemoveValues()` aufgerufen wird. Diese löscht die Key-Value-Pairs, welche dem neuen `Node` zur Verwaltung übergeben werden. Die Übergabe erfolgt durch den Aufruf der Methode `MoveDictionaryPartResult()` auf dem `Node`. Dieser wird unmittelbar an einen `Peer` weitergeleitet, auf welchem die übergebenen Key-Value-Pairs gespeichert werden. Dies findet nach dem im Kapitel [6.2.1.2 Weiterleitung] beschriebene Ablauf statt. Das Speichern erfolgt über die Synchronisationsmethode `AddValues()`. In der Schleife „GetPeers“ wird die benötigte Anzahl `Peer`s vom Parent-Node angefordert, um die gesetzte minimale Redundanz sicherstellen zu können. Dazu wird auf dem über die Methode `GetParentNode()` ermittelten Parent-Node die Methode `SetAsClone()` entsprechend viele Male aufgerufen. Die Methode `SetAsClone()` wird im Kapitel [6.2.5.2.1 SetAsClone] separat beschrieben.

Wird der neu hinzukommende `Peer` als neuer Clone hinzugefügt, d.h. die Alternative „AddPeer“ trifft nicht zu, wird die Methode `SearchResultAddPeerAsClone()` aufgerufen. Durch die Methode `CloneAttributes()` werden alle Attribute des zu klonenden `Node`s übernommen. Dies beinhaltet unteranderem das Setzen der Referenzen auf den Root-, Parent- und Child-Nodes des lokalen `Peer`s. Zusätzlich müssen die URLs der Clones auf dem eigenen `Node` aktualisiert werden. Hierzu wird die Methode `SetClones()` auf den entsprechenden `Node`s des lokalen `Peers` aufgerufen. Das Setzen der Parent- und Child-Nodes unterscheiden sich nur minimal und sind daher wegen der Übersicht nicht im Diagramm dargestellt. Sobald sich die URLs der Clones ändern, werden die dazugehörigen Stubs durch die Methode `UpdateStubs()` aktualisiert. Falls die Bedingung der Alternative „Sync“ zutrifft und somit eine Synchronisation für das Setzen der Clones gefordert wird, prüft die Methode `CheckClones()` ob die minimale Anzahl `Peer`s garantiert ist. Die Methode `CheckClones()` ist im Kapitel [6.2.5.2.2 CheckClones] beschrieben. Die URL des neuen `Peer`s wird den anderen Clones durch die Methode `AddClones()` mitgeteilt. Der Clone-Vorgang ist somit abgeschlossen.

6.2.5.2.1 SetAsClone

Die Aufgabe der Methode `SetAsClone()` ist einen Clone von einem `Peer` eines bereits existierenden `Node`s zu erstellen. Dazu werden alle relevanten Attribute des `Peer`s kopiert. Weil nun ein `Peer` eines Leaf-Nodes abgezogen wird, muss geprüft werden ob sich der betroffene `Node` auflösen muss oder ob die minimale Anzahl Clones noch vorhanden ist. Findet eine Auflösung des `Node`s statt, werden dessen `Peer`s ebenfalls zu Clones des `Node`s, welcher die Methode `SetAsClone()` aufgerufen hat. Anhand der folgenden Abbildung wird das beschriebene Vorgehen schrittweise erläutert.

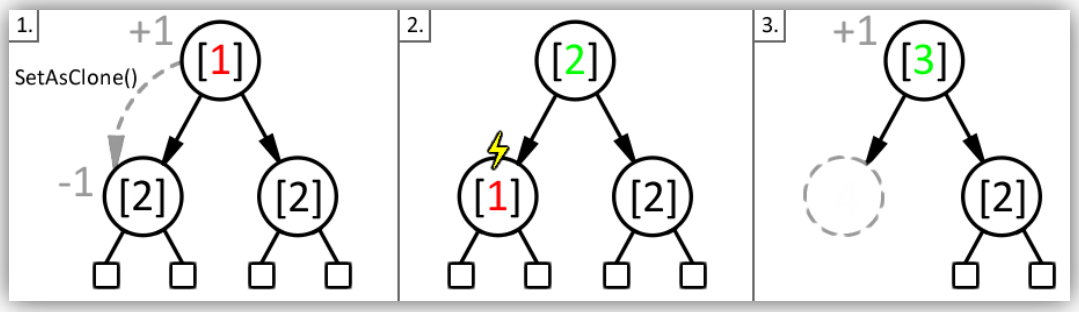


Abbildung 6-20: Methode - SetAsClone

1. Unter Umständen besitzt ein `Node` zu wenig Clones. In diesem Fall fordert er einen `Peer` über die Methode `SetAsClone()` eines anderen `Node`s an.
2. Da ein `Peer` einen `Node` verlässt, besteht die Möglichkeit, dass der betroffene `Node` zu wenig Clones besitzt.
3. Tritt die im Punkt 2 beschriebene Ausnahme ein, so wird der betroffene `Node` aufgelöst. Dazu wird der noch vorhandene `Peer` ebenfalls geklont. Die vom sich auflösende `Node` verwalteten Key-Value-Pairs werden über den Root-Node dem Tree neu hinzugefügt.

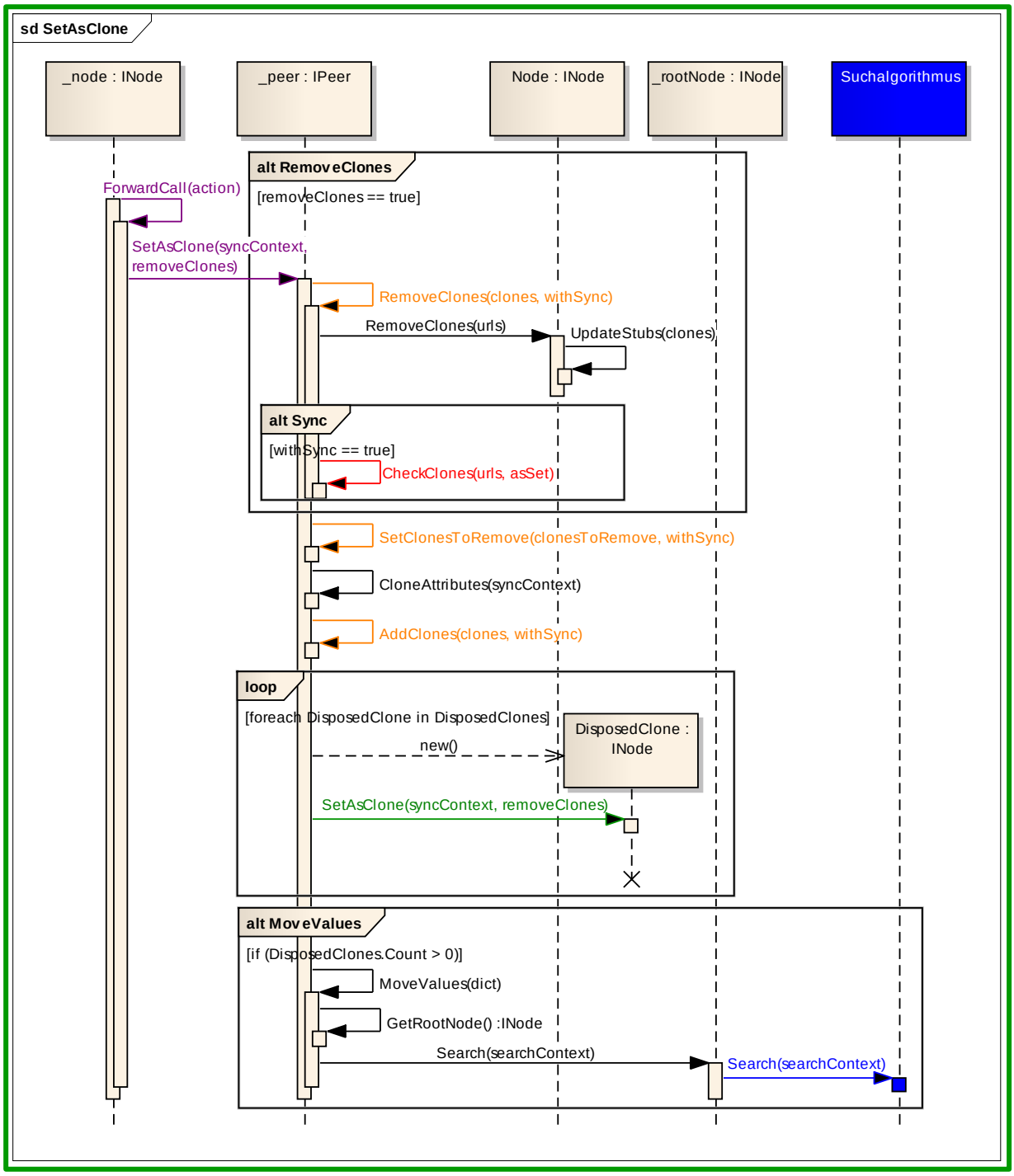


Abbildung 6-21: Sequenzdiagramm - SetAsClone

Zu Beginn ruft die Weiterleitungsmethode `ForwardCall()` die Methode `SetAsClone()` auf. Die Methode `SetAsClone()` teilt den zurückgelassen Clones das Wegfallen des `Peer`s über die Synchronisationsmethode `RemoveClones()` mit. Diese Synchronisation wird nur durchgeführt, falls der Parameter der Methode `SetAsClone()` auf `true` gesetzt ist. Werden die Clones neu gesetzt, müssen die Stubs mit der Methode `UpdateStubs()` aktualisiert werden. Besitzt auch der Parameter `withSync` den Wert `true`, so wird durch die Methode `CheckClones()` geprüft, ob die geforderte minimale Anzahl Clones noch eingehalten wird. Innerhalb der Methode `CheckClones()` wird die

Liste `DisposedClones` nachgeführt. Falls der `Node` aufgelöst wird, enthält die List die URLs der restlichen `Peer`s des sich aufzulösenden `Node`s.

Das Property `ClonesToRemove` wird durch die Methode `SetClonesToRemove()` um Eins reduziert. Dies ist nötig, da durch den Aufruf der Methode `SetAsClone()` ein Clone entfernt wird. Alle relevanten Eigenschaften werden in der Methode `CloneAttributes()` vom Kontext-Objekt übernommen. Der neu hinzukommende `Peer` trägt sich als weiteren Clone über die Synchronisationsmethode `AddClones()` beim `Node` ein.

Falls der `Node` aufgelöst wird, enthält die Liste `DisposedClones` die entsprechenden Peer-URLs. Jeder in der Liste `DisposedClones` referenzierte `Peer` wird ebenfalls geklont. Hierzu wird innerhalb einer Schleife durch jede URL der Liste `DisposedClones` iteriert. Anhand der URLs werden die entsprechenden `Node`s erstellt und auf jedem die Methode `SetAsClone()` aufgerufen.

Um die Persistenz sicherzustellen, müssen die auf den `Node` gespeicherten Key-Value-Pairs während dessen Auflösung auf den Predecessor-Node verschoben werden. Dies wird durch das Aufrufen der Methode `MoveValues()` bewerkstelligt. Ein solches Verschieben ist allerdings nur nötig, falls die Liste `DisposedClones` mindestens ein Element enthält und die Alternative „MoveValues“ somit zutrifft. Innerhalb der Methode `MoveValues()` werden die Key-Value-Pairs über den Root-Node mittels normalen Suchalgorithmus dem Tree neu hinzugefügt. Vergleiche hierzu Kapitel [6.2.1 Value hinzufügen].

6.2.5.2.2 CheckClones

Wird die Clone-Anzahl eines `Node`s reduziert, entspricht dies einem Ereignis, dass an bestimmte Bedingungen geknüpft ist. So muss bei jeder Reduzierung der Anzahl Clones eines `Node`s geprüft werden, ob das Minimum nicht unterschritten wurde. Solch eine Prüfung ist nötig um die gewünschte Redundanz garantieren zu können.

Es werden zwei Fälle unterschieden. Im ersten Fall wird die minimale Anzahl Clones nicht unterschritten und es müssen keine weiteren Massnahmen ergriffen werden. D.h. der Tree bleibt in seiner bisherigen Struktur bestehen. Im zweiten Fall aber wird das Minimum unterschritten und die gewünschte Redundanz muss wieder hergestellt werden. Je nach dem wo sich der betroffene Clone im Tree befindet, wird dies unterschiedlich bewerkstelligt. Hat der `Node` mindestens ein Child-Node, so wird die benötigte Anzahl `Peer`s von einem Leaf-Node zu ihm geholt. D.h. die beim Leaf-Node bestimmten `Peer`s werden Clones vom unterbesetzten `Node`. Ist der betroffene `Node` ein Leaf-Node, so können keine zusätzlichen `Peer`s von einem Leaf-Node angefordert werden. Stattdessen löst sich der `Node` auf. Die so zurückgebliebenen `Peer`s werden normal über den Root-Node dem Tree wieder hinzugefügt. Vergleiche dazu die im Kapitel [6.2.5 Peer hinzufügen] beschriebene Vorgehensweise.

Das oben beschriebene Verhalten hat den Vorteil, dass der Tree von unten d.h. von den Leaf-Nodes her abgebaut wird. Es werden nie Inner-Nodes aus dem Tree entfernt sondern ausschliesslich Leaf-Nodes. Das Entfernen eines Leaf-Nodes hat eine geringere Komplexität als das Entfernen eines Inner-Nodes.

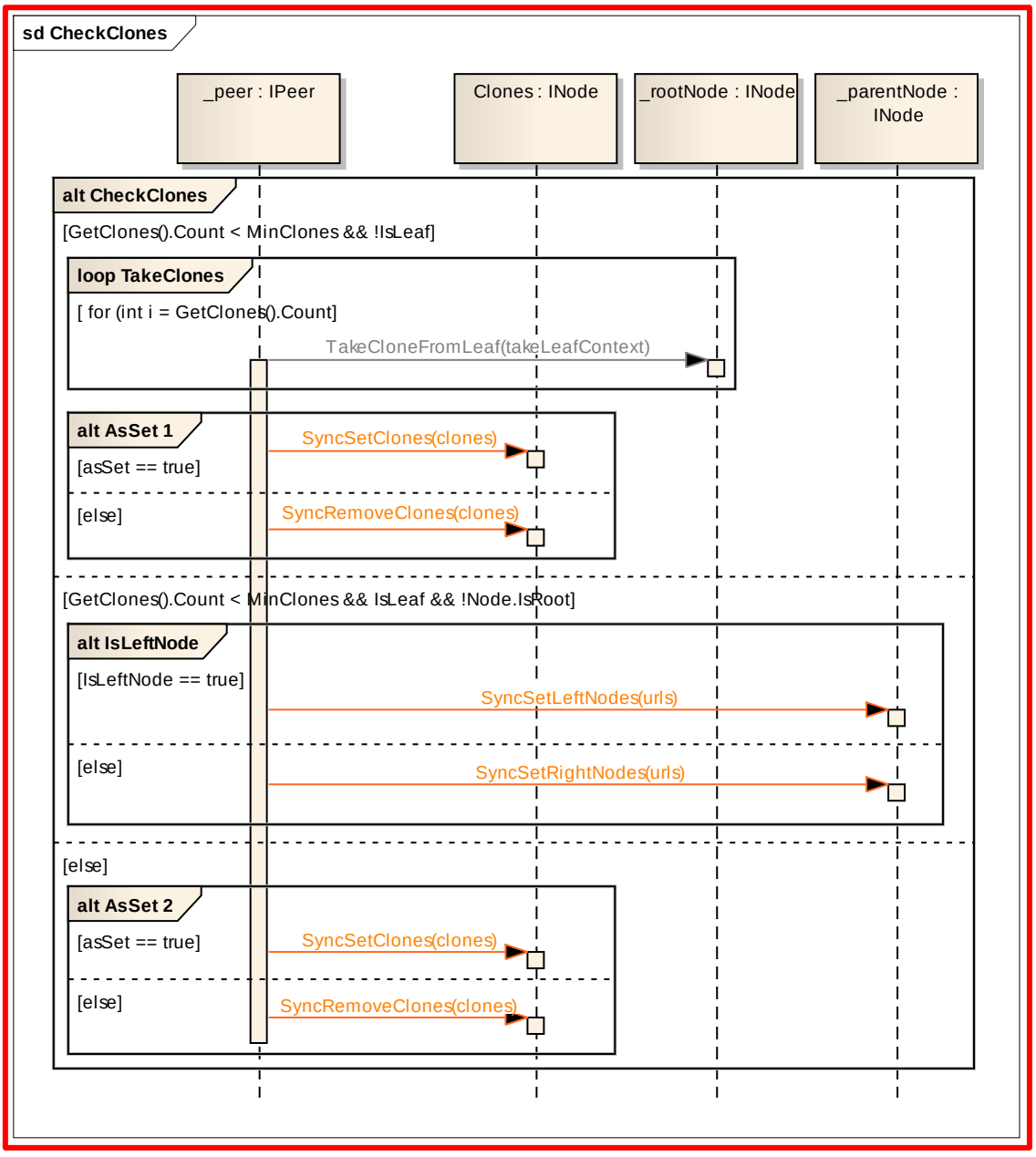


Abbildung 6-22: Sequenzdiagramm - CheckClones

Die Methode `CheckClones()` enthält die Alternative „CheckClones“, welche drei Fälle unterscheidet. Im ersten Fall wird die Anzahl Clones geprüft. Wurde die minimale Anzahl Clones unterschritten und handelt es sich beim betroffenen `Node` um einen Inner-Node, so wird mit Hilfe der Methode `TakeCloneFromLeaf()` die benötigte Anzahl `Peer`s innerhalb der Schleife „TakeClones“ von einem Leaf-Node angefordert. Der Aufruf von `TakeCloneFromLeaf()` startet eine Abfolge von Methodenaufrufe, welche aus Übersichtsgründen in einem separatem Diagramm im

Kapitel [6.2.5.2.3 TakeCloneFromLeaf] beschrieben sind. Je nachdem wie der Parameter `asSet` der Methode `CheckClones()` gesetzt ist, werden in der Alternative „AsSet 1“ andere Synchronisationsmethoden ausgeführt. So wird beim Entfernen eines Clones die Methode `SyncRemoveClones()` aufgerufen und beim Setzen der Clones die Methode `SyncSetClones()`.

Im zweiten Fall der Alternative „CheckClones“ wird ein `Node` aufgelöst. Der zweite Fall tritt ein, wenn die minimale Anzahl Clones unterschritten wurde und es sich um ein Leaf-Node aber nicht um den Root-Node handelt. Wird ein Leaf-Node aufgelöst, muss dieses Ereignis dem Parent-Node mitgeteilt werden. Ist der `Node` ein Left-Node so trifft die Alternative „IsLeftNode“ zu und die Synchronisationsmethode `SyncSetLeftNodes()` wird aufgerufen. Trifft die Alternative nicht zu, wird die Methode `SyncSetRightNodes()` aufgerufen. Die beiden Synchronisationsmethoden setzen im Wesentlichen den entsprechenden Child-Node des Parent-Nodes auf einen `NodeNull`.

Treffen beide beschriebenen Bedingungen nicht zu, findet die Synchronisation nur unter den Clones desselben `Node`s statt. Wieder wird der Parameter `asSet` in der Alternativen „AsSet 2“ für die Wahl der Synchronisationsmethode verwendet.

6.2.5.2.3 TakeCloneFromLeaf

Wird auf einem Inner-Node die minimale Clone-Anzahl unterschritten, wird mit Hilfe der Methode `TakeCloneFromLeaf()` ein Leaf-Node bestimmt von dessen einen `Peer` angefordert wird. Hierzu wird von dem Root-Node der Tree bis zu einem Leaf-Node hinunter traversiert. Beim Abstieg muss bei jedem Inner-Node entschieden werden, auf welchem Child-Node der Abstieg fortgesetzt wird. Die folgende Abbildung zeigt wann welche Entscheidung anhand welcher Kriterien gefällt wird.

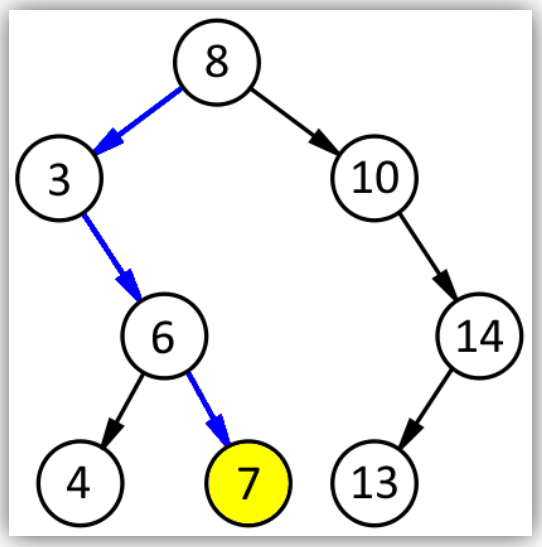


Abbildung 6-23: TakeCloneFromLeaf-Algorithmus

Die erste Entscheidung muss auf dem Root-Node gefällt werden. Da er sowohl ein Left-Node wie auch ein Right-Node besitzt, wird mit Hilfe eines zufällig generierten Index ein Child-Node bestimmt. Im Beispiel fällt die Wahl auf den `Node` mit dem Key 3. Hier ist die Entscheidung klar, da dieser `Node` lediglich ein Right-Node besitzt. Beim `Node` mit dem Key 6 wird wieder zufällig ein Child-Node bestimmt. Mit der vorherigen Entscheidung ist der Algorithmus bei einem Leaf-Node angelangt, welcher nun dem aufrufenden `Node` als Ergebnis zurückgegeben werden kann.

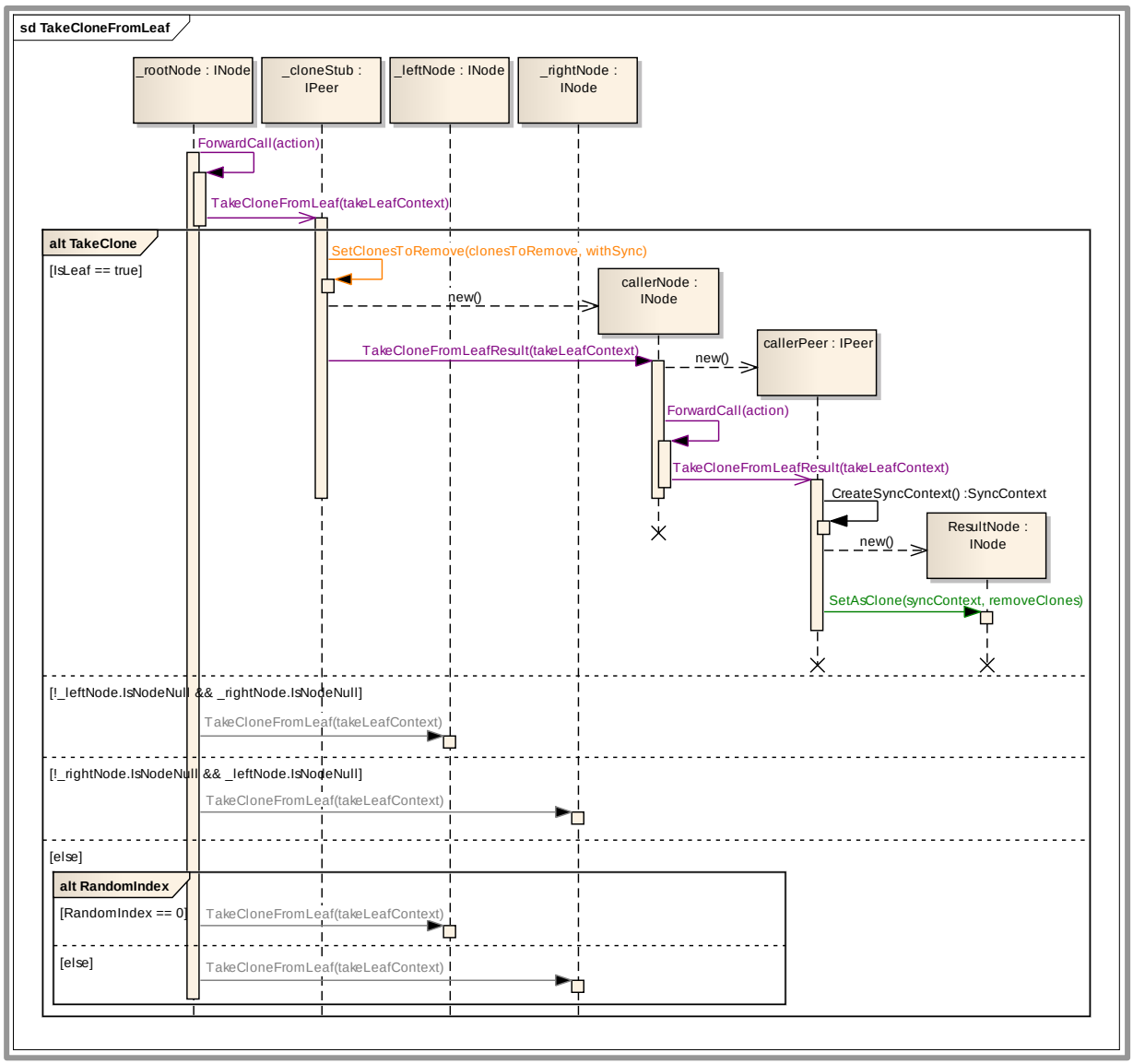


Abbildung 6-24: Sequenzdiagramm - TakeCloneFromLeaf

Jeder Peer im Netz kennt den Root-Node. Somit ist jeder `Peer` in der Lage die Methode `TakeCloneFromLeaf()` auf dem Root-Node aufzurufen. Dieser leitet den Aufruf an einen Clone weiter, welcher mit der Methode `ForwardCall()` bestimmt wird. Nun werden die Entscheidungen, welche anhand der Abbildung [Abbildung 6-23: TakeCloneFromLeaf-Algorithmus] aufgezeigt wurden, gefällt.

Die Alternative „TakeClone“ unterscheidet vier Fälle. Im ersten Fall handelt es sich beim aktuellen Node um ein Leaf-Node und die Weiterleitung des Algorithmus terminiert. D.h. als Erstes wird die Anzahl der zu entfernenden Nodes mit der Methode `SetClonesToRemove()` erhöht und synchronisiert. Danach kann auf dem Node, welcher den Algorithmus gestartet hat, die Callback Methode `TakeCloneFromLeafResult()` aufgerufen und wie gewohnt mit Hilfe der Methode `ForwardCall()` an einen Peer weitergeleitet werden. In der aufgerufenen Callback-Methode erstellt die Methode `CreateSyncContext()` eine Kopie aller Attribute des Nodes und speichert diese in ein Kontext-Objekt ab. Innerhalb der Methode `SetAsClone()` wird das Kontext-Objekt verwendet um alle Werte des aufrufenden Nodes zu übernehmen. Dadurch wird eine bestimmte Anzahl Peers des gefundenen Leaf-Nodes zu Clones des Nodes, welcher den Algorithmus aufgerufen resp. gestartet hat. Die Methode `SetAsClone()` wird aus Übersichtsgründen nicht dargestellt. Sie ist jedoch in einem separatem Diagramm im Kapitel [6.2.5.2.1 SetAsClone] beschrieben.

In den restlichen Fällen befindet sich der Algorithmus auf einem Inner-Node. Es muss daher entschieden werden, wohin der Abstieg fortgesetzt werden kann. Der Aufruf wird dem Left-Node weitergeleitet, falls dies der einzige Child-Node des aktuellen Nodes ist. Umgekehrt wird der Algorithmus auf dem Right-Node fortgesetzt, falls dieser der alleinige Child-Node ist. Die Weiterleitung findet statt indem auf dem entsprechenden Node die Methode `TakeCloneFromLeaf()` aufgerufen wird. Besitzt der Node ein Left-Node sowie ein Right-Node muss die Wahl zufällig bestimmt werden.

6.2.6 Peer entfernen

Wird ein `Peer` kontrolliert heruntergefahren, meldet sich dieser bei seinen Clones ab. Nach der Abmeldung wird eine allfällige Auflösung des `Node`s eingeleitet.

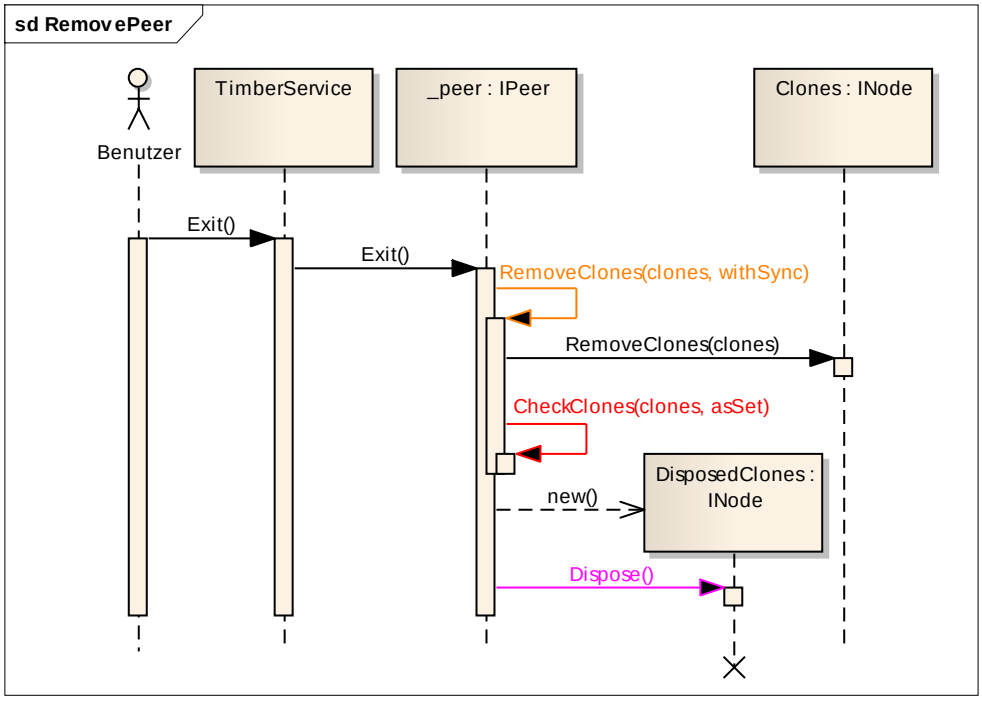


Abbildung 6-25: Sequenzdiagramm - Peer entfernen

Der Benutzer kann über die `TimberService`-Schnittstelle den Service beenden indem er die Methode `Exit()` aufruft. Deren Aufruf wird unmittelbar an den gehosteten `Peer` weitergeleitet. Die URL des sich beendenden `Peer`s muss aus der Clone-Liste der anderen Clones vom betroffenen `Node` entfernt werden. Hierzu wird die Methode `RemoveClones()` auf dem `Peer` aufgerufen. Da es sich bei der Methode `RemoveClones()` um eine Synchronisationsmethode handelt, wird der Aufruf an den `Node` weitergeleitet. Da das Entfernen eines Clones an bestimmte Bedingungen geknüpft ist, wird nicht wie bisher direkt die entsprechende Synchronisationsmethode aufgerufen. Diese wird stattdessen innerhalb der Methode `CheckClones()` aufgerufen. Die Bedingungen und die Methode `CheckClones()` sind im Kapitel [6.2.5.2.2 CheckClones] separat beschrieben.

Während der Methode `CheckClones()` werden Clones, welche sich auflösen und sich dem Tree neu hinzufügen müssen, in eine Liste gespeichert. Anhand dieser Liste wird ein neuer `Node` erzeugt, auf welchem die Methode `Dispose()` aufgerufen wird. Der Ablauf der Methode `Dispose()` ist im Kapitel [6.2.6.1 Dispose] beschrieben.

6.2.6.1 Dispose

Der Benutzer kann auf Wunsch einen `Peer` herunterfahren. Tritt dieses Ereignis ein, wird geprüft ob die minimale Anzahl Clones noch eingehalten wird. Wird das Minimum unterschritten, kann die geforderte Redundanz nicht mehr gehalten werden. Dies hat zur Folge, dass sich der betroffene Leaf-Node auflöst und sich dessen `Peer`s dem Tree neu hinzufügen. Bevor sich ein `Peer` neu hinzufügen kann, müssen dessen Key-Value-Pairs dem Predecessor-Node übergeben werden.

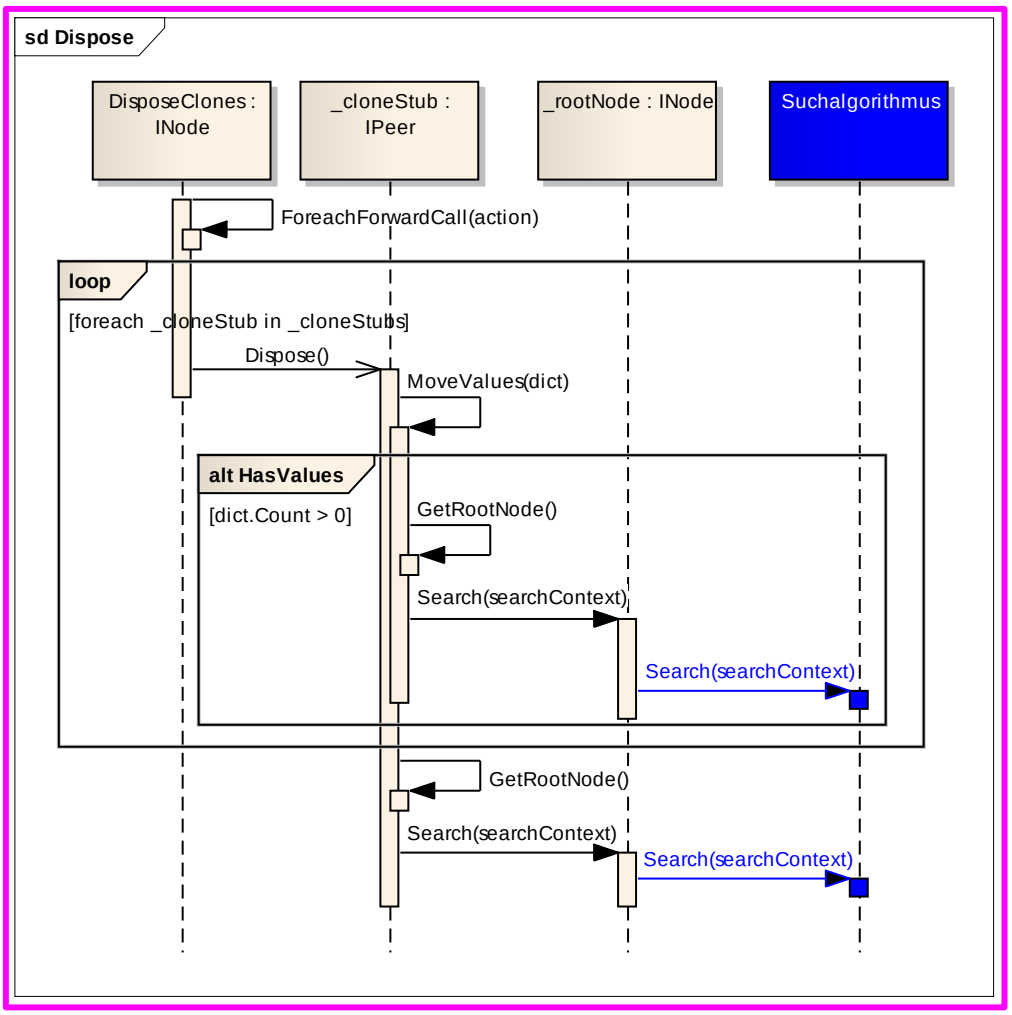


Abbildung 6-26: Sequenzdiagramm - Dispose

Wird die Methode `Dispose()` auf einem `Node` aufgerufen, wird der Aufruf unmittelbar durch die Methode `ForeachForwardCall()` an alle `Peer`s des `Node`s weitergeleitet. Das Beenden eines `Peer`s hat zur Folge, dass alle auf dem `Peer` gespeicherten Key-Value-Pairs auf dessen Predecessor-Node verschoben werden. Dies geschieht über den Aufruf der Methode `MoveValues()`. Die Alternative „HasValues“ prüft ob der `Peer` überhaupt irgendwelche Key-Value-Pairs verwaltet. Ist dies der Fall werden die Values verschoben resp. über den Suchalgorithmus und dem passendem ACT dem Tree neu hinzugefügt. Ein weiteres Mal wird der Suchalgorithmus verwendet um die Clones des sich auflösenden `Node`s dem Tree hinzuzufügen. Der Suchalgorithmus wird jeweils auf dem Root-Node gestartet. Dieser wird durch die Methode `GetRootNode()` geliefert.

6.2.7 Clone fällt aus

Damit ein `Peer` feststellen kann, ob einer seiner Clones ausgefallen und somit nicht mehr erreichbar ist, muss ein ständiges Polling stattfinden. Der durch den `Peer` festgestellte Ausfall wird dem Parent- und den Child-Nodes sowie seinen Clones mitgeteilt.

Dieses Szenario wurde aus Zeitgründen nicht implementiert.

6.2.8 Tree zeichnen

Der Benutzer hat die Möglichkeit die Netzstruktur des BSTs grafisch darzustellen. Um dies zu ermöglichen, wird ein Clone jedes `Node`s traversiert. Dabei werden sämtliche Informationen der einzelnen `Node`s gesammelt. Sobald alle Informationen erhalten wurden, kann der Test- und Visualisierungsvorgang gestartet werden.

Das Zurückgeben der gesammelten Informationen und das Bestimmen wann alle `Node`s traversiert wurden, soll im Folgenden genauer betrachtet werden.

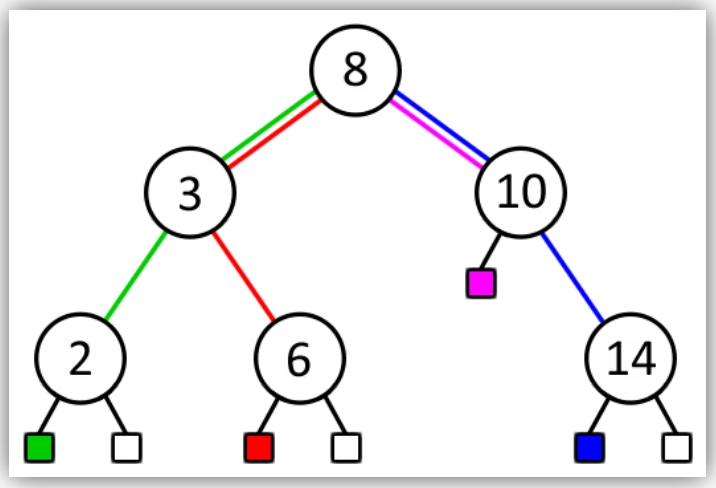


Abbildung 6-27: Zurückgegebene Äste der Methode Draw()

Die gesammelten Informationen werden nur von linken `NodeNull`s zurückgegeben. Die Informationen enthalten jeweils die Daten der `Node`s eines Astes. Die sogenannten Äste sind farblich in der Abbildung hervorgehoben. Jeder Ast wird asynchron dem `Peer`, welcher das Zeichnen initiierte, zurückgegeben.

Das Erkennen der vollständigen Traversierung des Trees stellt auf Grund der Asynchronität ein etwas komplexerer Sachverhalt dar. Sobald ein `Node` traversiert wird, speichert dieser die Richtung der Weiterleitung des Aufrufs in das Kontext-Objekt ab. Dieser Sachverhalt soll nachfolgend erläutert werden.

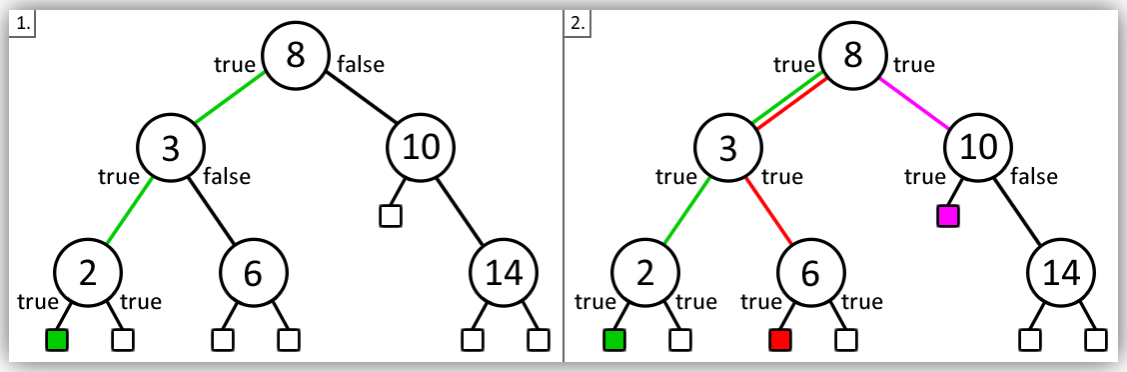


Abbildung 6-28: Erkennung einer vollständigen Traversierung

Sobald der initiierende `Peer` einen Ast erhält, prüft dieser, ob alle bereits erhaltene `Node`s sowohl links wie auch rechts vollständig traversiert wurden. Ein `Node` gilt als vollständig traversiert, wenn aus mindestens zwei unterschiedlichen Ästen festgestellt werden kann, dass der `Node` das Kontext-Objekt sowohl dem Left- wie auch Right-Node weitergeleitet hat. In der Abbildung ist dies jeweils mit `true` resp. `false` neben einem `Node` dargestellt. Im ersten Schritt der oben gezeigten Abbildung ist ersichtlich, dass nur der `Node` mit dem Key 2 vollständig traversiert wurde. Durch die `Node`s mit den Keys 3 und 8 kann das Fehlen mindestens zweier Äste festgestellt werden. Durch das Hinzukommen des roten und rosa Astes wurden die `Node`s mit den Keys 3, 6 und 8 vollständig traversiert. Da der rosa Ast jedoch einen noch unvollständig traversierten Ast enthält, muss der initiierende `Peer` weiterhin auf die Komplettierung der Traversierung warten. Der gesamte Sachverhalt ist im unten stehenden Sequenzdiagramm dargestellt.

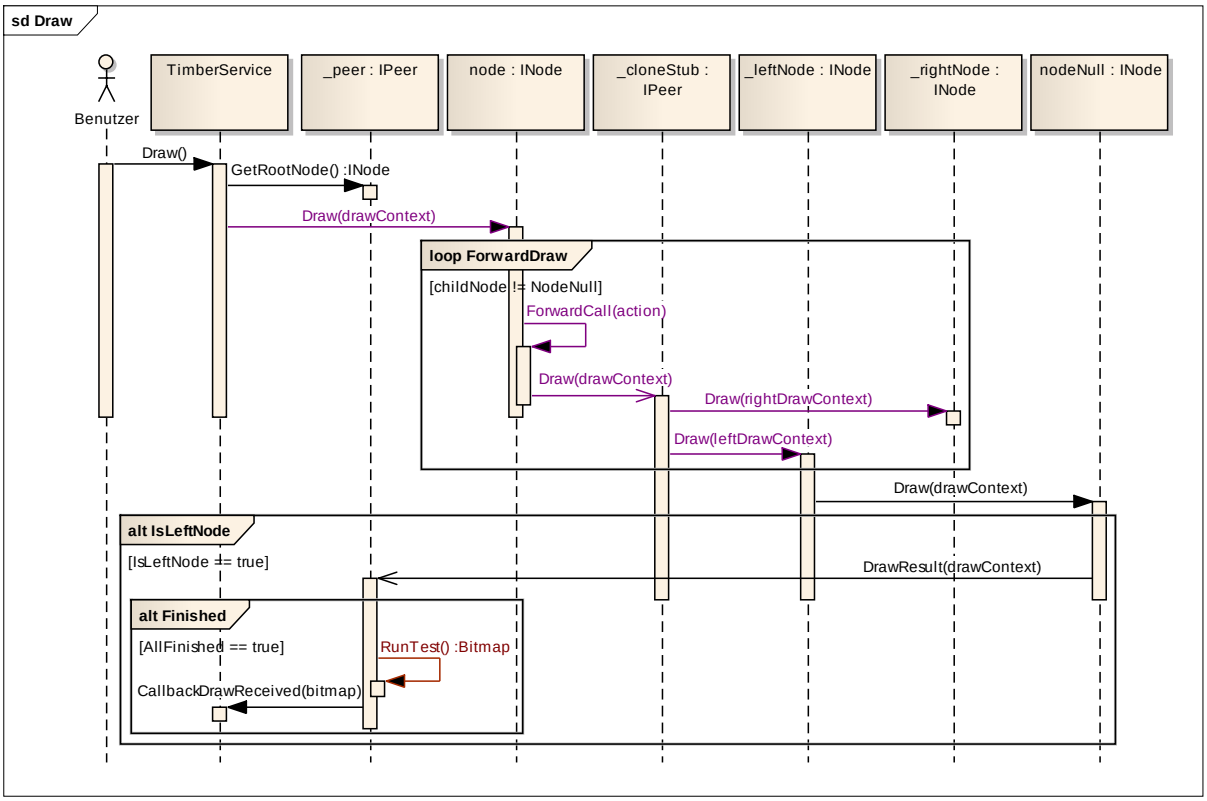


Abbildung 6-29: Sequenzdiagramm - Draw

Die `TimberService`-Schnittstelle stellt dem Benutzer die Methode `Draw()` zur Verfügung. Diese erlaubt es ihm den Zeichnungsvorgang zu initiieren. Der Vorgang startet auf dem durch die Methode `GetRootNode()` ermittelten Root-Node. Der Aufruf der Methode `Draw()` wird sowohl dem Left- wie auch Right-Node weitergeleitet. Bei jeder Weiterleitung speichert der Clone die Informationen über seinen `Node` in eine Liste mit `NodeTestInfo` Instanzen. Zu beachten ist, dass für den Left- und Right-Node jeweils eine unterschiedliche Instanz der Klasse `DrawContext` mitgegeben wird. Das Kontext-Objekt bzw. der Methodenaufruf wird solange an die Child-Nodes weitergeleitet, bis der Aufruf ein `NodeNull` erreicht. Dieser sich wiederholende Vorgang ist mit der Schleife „ForwardDraw“ dargestellt. Wird die Methode `Draw()` auf einem `NodeNull` aufgerufen und handelt es sich hierbei um ein Left-Node, wird die Callback-Methode `DrawResult()` auf dem `Peer` aufgerufen. Dabei handelt es sich um den `Peer`, auf welchem der Benutzer die Methode `Draw()` aufgerufen hat. Der Aufruf der Callback-Methode `DrawResult()` stellt das Zurückgeben eines Astes dar und ist in der Alternative „IsLeftNode“ dargestellt. In der Methode `DrawResult()` wird die Methode `RunTest()` gestartet, falls alle `Node`s traversiert wurden bzw. die Alternative „IsFinished“ zutrifft. Das durch die Methode `RunTest()` erstellte Bitmap wird mit der Methode `CallbackReceived()` dem `TimberService` zurückgegeben.

Aus Übersichts- und Komplexitätsgründen wird der Ablauf der Methode `RunTest()` und der weitere Aufruf der Methode `Run()` separat beschrieben. Als Vorbereitung für das Zeichnen des Trees werden alle `Node`s in einer tree-komfortablen Datenstruktur gespeichert. Während dem Abspeichern werden

die gesammelten Daten einfachen Tests unterzogen. Das Abspeichern soll anhand folgender Abbildung erläutert werden.

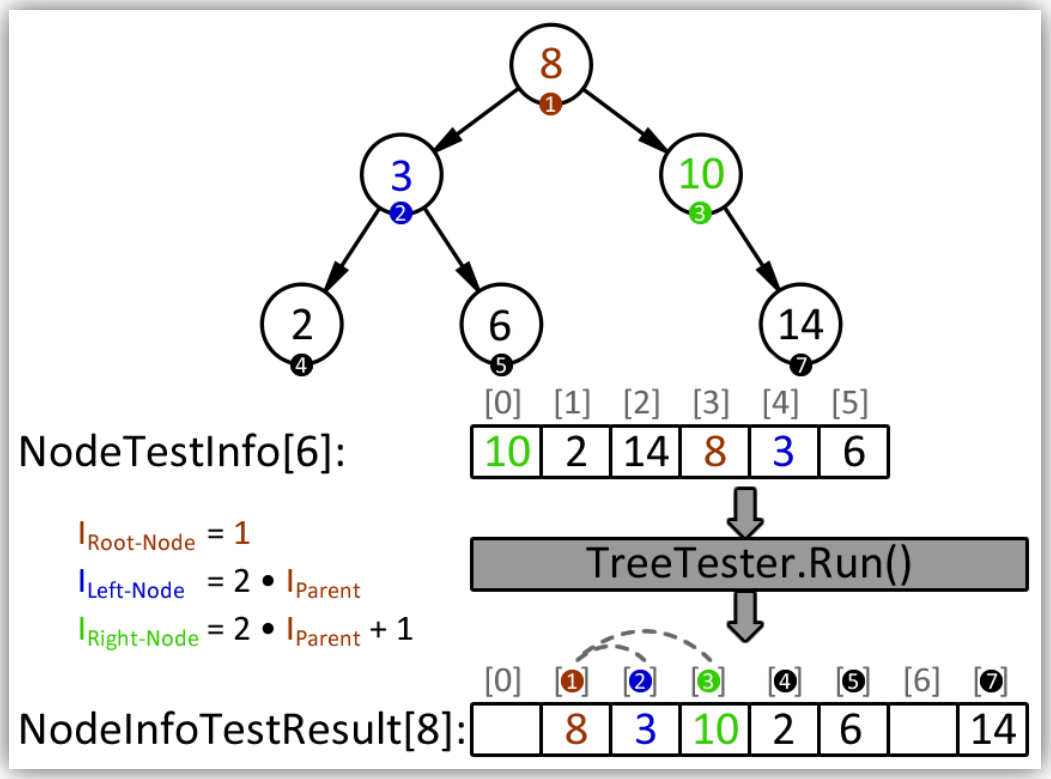


Abbildung 6-30: Abspeichern eines Trees in eine tree-komfortable Datenstruktur

Die gesammelten `NodeTestInfo`s befinden sich unsortiert in einer Liste. Um den kompletten Tree in einer Datenstruktur abzuspeichern, prüft die Methode `Run()` der Klasse `TreeTester` die Beziehungen zwischen den `NodeTestInfo`s. Sind die Beziehungen eines `NodeTestInfo`s soweit in Ordnung, werden die Resultate in Form einer Instanz der Klasse `NodeInfoTestResult` im dafür vorgesehenen Array abgespeichert. Jede Instanz der Klasse `NodeInfoTestResult` hat einen klar definierten Index im Array. Das Berechnen der Indexe erfolgt nach den in der Abbildung gezeigten Formeln. Nach dieser Transformation ist der komplette Tree in einem Array abgespeichert und somit für die Visualisierung vorbereitet.

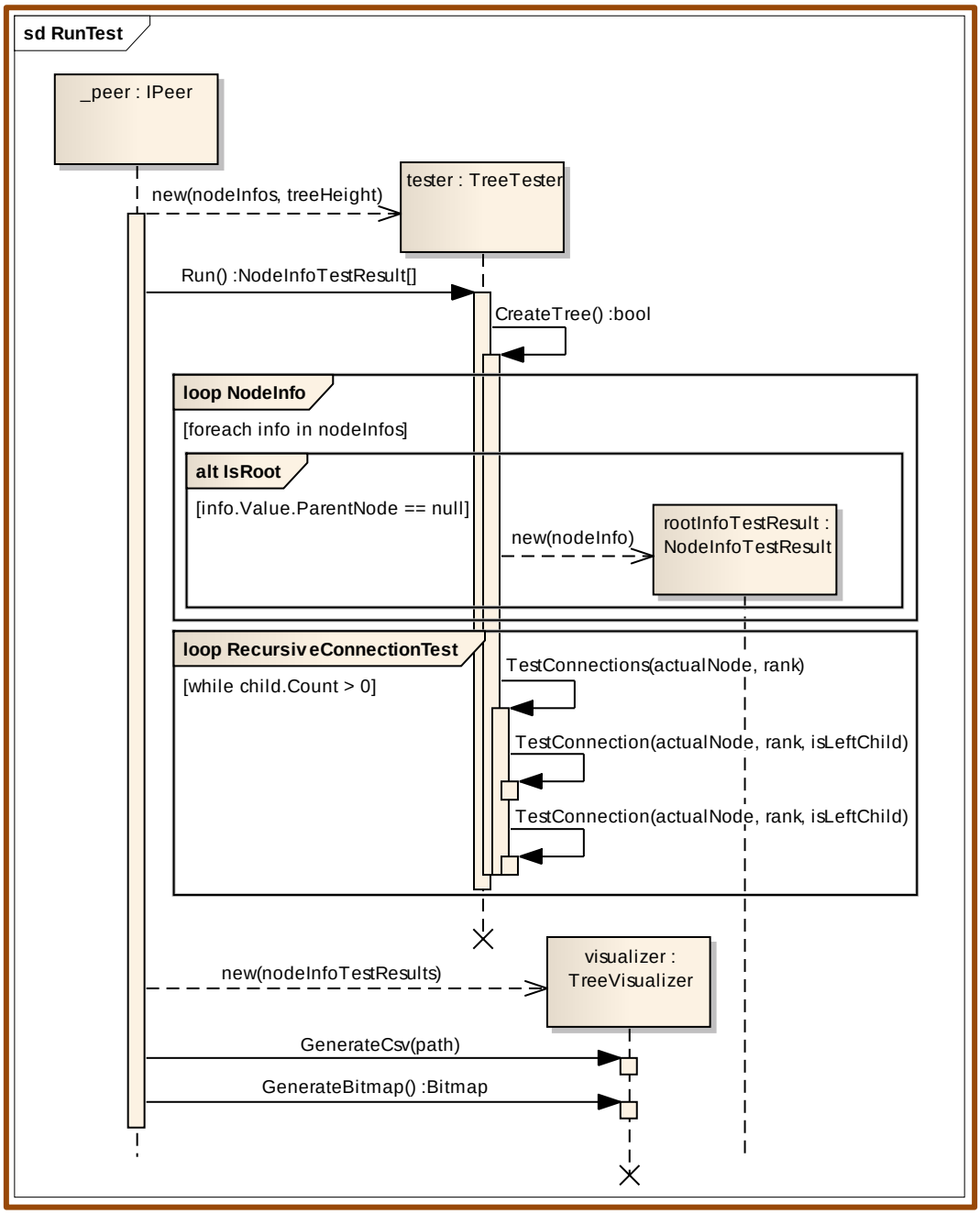


Abbildung 6-31: Sequenzdiagramm - RunTest

Die vorhin gesammelten Informationen über die `Node`s werden in Form einer Liste im Konstruktor der Klasse `TreeTester` mitgegeben. Nach der Erstellung eines `TreeTester`s wird die Methode `Run()` durch den `Peer` aufgerufen. Die Methode `CreateTree()` sucht als Erstes die Informationen über den Root-Node. Um den Root-Node zu identifizieren, wird in der Schleife „NodeInfo“ durch die vorhin übergebene Liste iteriert. Sobald die Alternative „IsRoot“ zutrifft, wird das Iterieren unterbrochen und der gefundene Root-Node als `actualNode` gesetzt. Dieser wird der Methode `TestConnections()` übergeben. Die Methode `TestConnections()` startet einen rekursiven Aufruf, welcher in der Schleife „RecursiveConnectionTest“ im Sequenzdiagramm dargestellt ist. Die beiden Aufrufe der Methode `TestConnection()` beinhalten die Prüfung der Beziehungen zu den jeweiligen Child-Nodes und das korrekte Speichern der `NodeInfoTestResult`s. Nach jeder Speicherung wird

die Methode `TestConnections()` erneut, jedoch mit einem neu gesetzten Parameter `actualNode`, aufgerufen. Somit bestehen Rekursionen, welche über zwei Methodenaufrufe hinweg verlaufen und jeweils bei einem Leaf-Node terminieren.

Nach dem Aufbereiten der tree-komfortablen Datenstruktur wird diese einer neuen Instanz der Klasse `TreeVisualizer` übergeben. Um die Daten des Trees in einem exportierbaren Format anzubieten, werden diese in einer CSV-Datei abgespeichert. Die Aufgabe wird von der Methode `GenerateCSV()` erledigt. Der Aufruf der Methode `GenerateBitmap()` berechnet bzw. erstellt die grafische Darstellung des Trees. Diese wird als Rückgabewert der Methode `RunTest()` gesetzt.

Im folgenden Abschnitt wird das Ergebnis einer grafischen Repräsentation des Trees aufgezeigt. Dabei wird auf die dargestellten Informationen und das Erstellen der Grafik eingegangen.

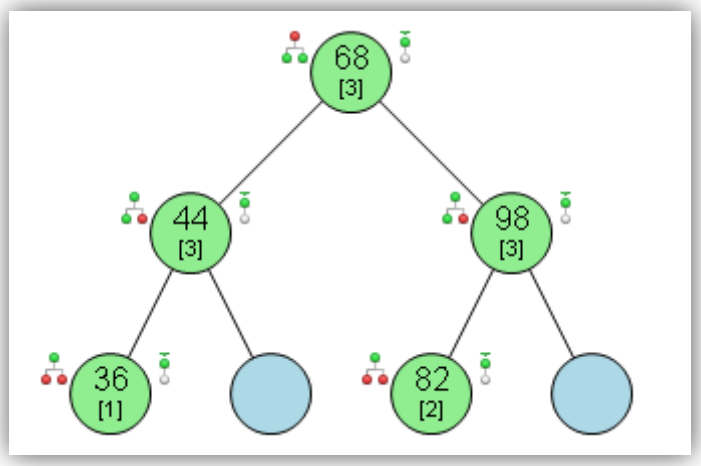
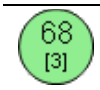


Abbildung 6-32: Grafische Darstellung eines Trees

 **Node:**	Einen grün eingefärbten Kreis stellt einen `Node` im Tree dar. Die obere Zahl im Kreis repräsentiert den Key des `Node`s. Es sind jedoch nur die ersten zwei Zeichen des SHA1-Hashwertes ersichtlich. Die etwas kleinere und in eckigen Klammern gesetzte Zahl zeigt die Anzahl `Peer`s resp. Clones innerhalb des `Node`s.
 **Platzhalter:**	Einen blau eingefärbten Kreis dient als Platzhalter für nicht existierende `Node`s. Sie stellen keine weiteren Informationen dar und können daher ignoriert werden.

	Parent- / Child-Node Status	Die jeweiligen Verbindungsstatus zu dem Parent- bzw. Child-Nodes werden links von einem `Node` dargestellt. Ein roter Punkt bedeutet, dass entweder keine entsprechende Verbindung existiert oder diese fehlerhaft ist. Ein grüner Punkt zeigt eine korrekte Verbindung.
	Root-Node Status	Rechts von einem `Node` ist der Verbindungsstatus zum Root-Node ersichtlich. Wobei ein grüner Punkt wieder eine korrekte und ein roter Punkt eine fehlerhafte Verbindung darstellt.

Tabelle 6-26: Legende zur grafischen Darstellung eines Trees

Für die Erstellung der Grafik wurden eigenständig hergeleitete Formeln verwendet. Das Zeichnen des Trees und die Anwendung der Formeln soll im Folgenden aufgezeigt werden.

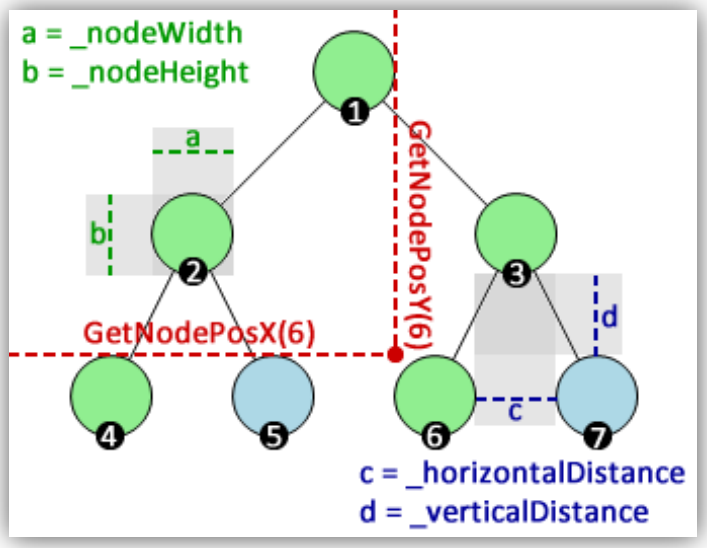


Abbildung 6-33: Zeichnen des Trees

Anhand des Ranks eines `Node`s können die genauen X/Y-Koordinaten des entsprechenden `Node`s bestimmt werden. Der Rank eines `Node`s wiederspiegelt den Index des aufbereiteten Arrays, welches die `NodeInfoTestResult` Instanzen enthält. Vergleiche dazu Abbildung [Abbildung 6-30: Abspeichern eines Trees in eine tree-komfortable Datenstruktur].

Der Code, der in der oben gezeigten Abbildung dargestellten Methoden `GetNodePosX()` und `GetNodePosY()` ist in folgender Tabelle ersichtlich. Die beiden Methoden enthalten die selber hergeleiteten Formeln, welche für das Zeichnen des Trees eingesetzt werden.

```
private int GetNodePosX(int rank)
{
	int h = GetRankHeight(rank);
	int s = (int)((_nodeInfoTestResults.Length / Math.Pow(2, h)) - 1)
		* _horizontalDistance;
	return (int)((rank % Math.Pow(2, h)) * s + (rank % Math.Pow(2, h))
		* _nodeWidth + (int)(s / 2));
}

private int GetNodePosY(int rank)
{
	int h = GetRankHeight(rank);
	return h * _nodeHeight + h * _verticalDistance;
}

private int GetRankHeight(int rank)
{
	return (int)Math.Log(rank, 2);
}
```

Tabelle 6-27: Code - Berechnungsformeln zum Zeichnen des Trees

7 Probleme

7.1 Hierarchische Struktur

Der Einsatz eines BSTs bringt den Nachteil einer hierarchischen Struktur mit sich. D.h. je weiter oben sich ein Node im Tree befindet, desto wichtiger ist dieser für das Netz. So ist der Ausfall des Root-Nodes fatal. Fällt jedoch ein Leaf-Node aus, beinträchtig dies den BST kaum. Hinzu kommt, dass die Belastung der einzelnen Node unterschiedlich ist. So wird zum Beispiel der Root-Node bei jeder Suchanfrage involviert. Die Wahrscheinlichkeit, dass aber ein Leaf-Node eine Suchanfrage empfängt, ist einiges geringer.

Mit dem Clone-Konzept wird diesem Problem entgegengewirkt. Die unterschiedliche Wichtigkeit eines Nodes besteht weiterhin, jedoch in einer abgeschwächten Form. Gleiches gilt für die Auslastung der Nodes. So werden die Clones des Root-Nodes immer noch stärker belastet als die Clones eines Leaf-Nodes. Durch das geeignete Setzen einer minimalen Anzahl Clones auf einem Node durch den Benutzer kann das Problem weitestgehend umgangen werden.

Um dieses Problem vollständig zu lösen, könnte der im Kapitel [5.8 Tree-Höhe abhängige Clone-Anzahl] beschriebenen Ansatz verwendet werden.

7.2 Synchronisationsproblem

Clones können unter bestimmten Umständen ihre Node-Zugehörigkeit ändern. Treten während diesem Vorgang gewisse Ereignisse ein, können inkonsistente Zustände entstehen. Wie dies genau zustande kommt, ist in der folgenden Abbildung festgehalten.

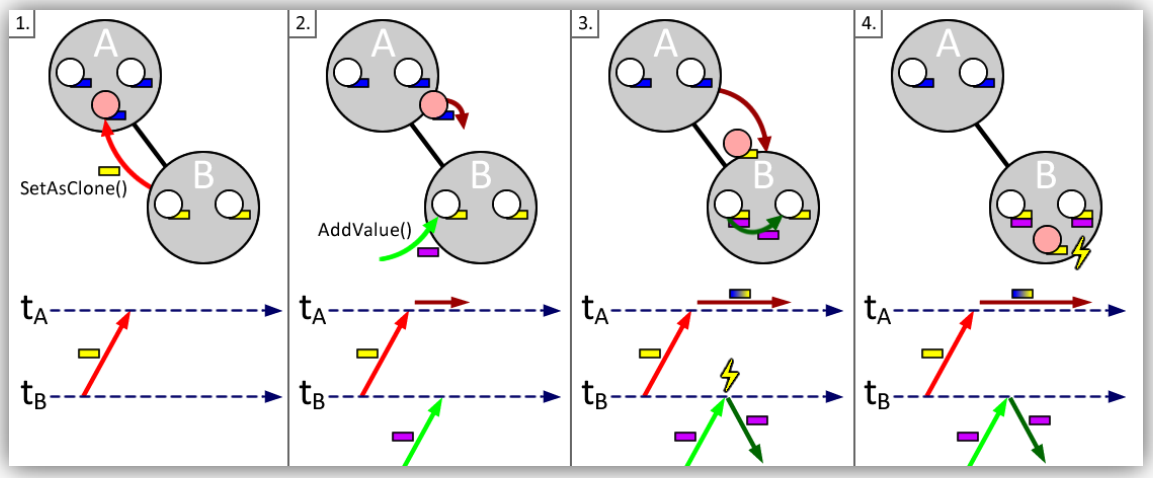


Abbildung 7-1: Synchronisationsproblem

Das Synchronisationsproblem wird anhand eines Beispiels erläutert, in dem ein `Node` ein neues Value erhält. Dieser Vorgang wird mit Hilfe der in der Abbildung gezeigten vier Schritte erklärt.

1. Der `Node` B benötigt aus hier irrelevanten Gründen einen zusätzlichen `Peer`. Dieser fordert den benötigten `Peer` mit Hilfe der Methode `SetAsClone()` vom `Node` A an. Beim Aufruf übergibt `Node` B einem `Peer` von `Node` A alle für das Klonen benötigten Informationen. Diese beinhalten unteranderem das gelbe Key-Value-Pair.
2. Der bestimmte `Peer` des `Nodes` A beginnt sich zu klonen. Währenddessen erhält `Node` B das neu zu verwaltende violette Key-Value-Pair.
3. Der `Peer`, welcher das neue Key-Value-Pair erhält, synchronisiert diese Änderung mit seinen Clones. Da sich der rot eingefärbte `Peer` noch im Klonvorgang befindet und somit noch kein Clone des `Node`s B ist, erfährt er nichts von der Synchronisation.
4. Der Klonvorgang des `Peer`s ist nun vollständig abgeschlossen. Da der `Peer` von der Synchronisation ausgeschlossen blieb, fehlt im nun das violette Key-Value-Pair. Es existiert also einen inkonsistenten Zustand innerhalb des `Node`s B.

Das geschilderte Problem bringt bestimmte Auswirkungen mit sich. Existiert ein inkonsistenter Zustand auf einem `Node` kann ein gesuchtes Value teilweise nicht gefunden werden obwohl es vorhanden sein sollte. Eine weitere Auswirkung besteht darin, dass ein `Peer` möglicherweise von einem fehlerhaften `Peer` geklont wird.

Ähnliche Synchronisationsprobleme könnten auch in anderen Situationen als nur beim Hinzufügen eines neuen Key-Value-Pairs auftreten. Das geschilderte Problem lässt sich auf die Asynchronität im Tree zurückführen. Allgemein existiert bei asynchronen Aufrufen die Gefahr auf die erhaltene Nachricht bzw. Event falsch zu reagieren. Es stellt sich immer die Frage ob nun auf die Nachricht reagiert werden soll oder nicht. Oder wie in unserem Fall, dass sogar eine Nachricht gar nicht erst empfangen wird.

Ein möglicher Lösungsansatz bestände darin, periodisch Synchronisationsnachrichten unter den Clones eines `Node`s auszutauschen. Diese Nachrichten würden die inkonsistenten Zustände ausgleichen. Um das richtige Interpretieren der empfangenen Nachrichten zu ermöglichen, wäre das Einführen einer logischen Zeit denkbar.

8 Tutorial - How to Timber!

Dieses Kapitel beschreibt die API und die Verwendung von „Timber!“. Jegliche Benutzeraktionen werden durch die Klasse `TimberService` entgegengenommen, verarbeitet und weiterleitet. D.h. der Benutzer ruft lediglich Methoden dieser Klasse auf. Darum werden zu Beginn alle Methoden der Klasse `TimberService` beschrieben. Später wird anhand einer Beispielanwendung die Verwendung von „Timber!“ demonstriert.

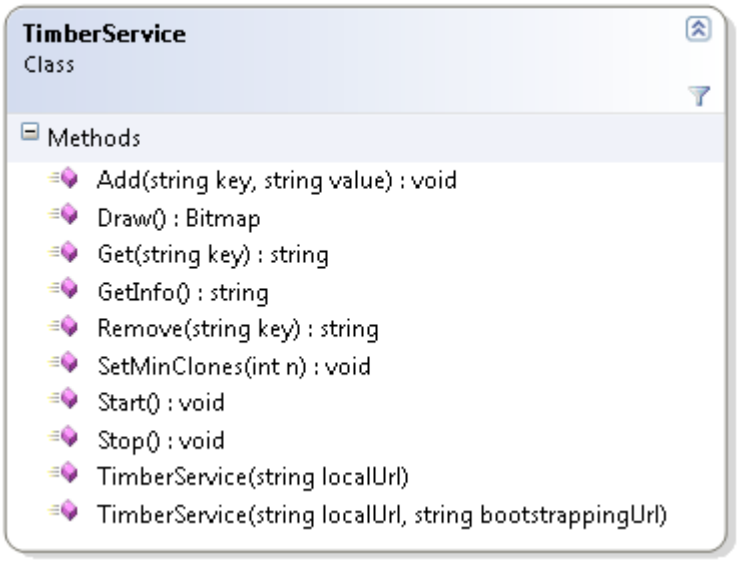


Abbildung 8-1: „Timber!“ - API

Add	Die Methode `Add()` erlaubt das Abspeichern von neuen Values in der DHT. Jedes zu speichernde Value muss mit einem eindeutigen Key identifizierbar sein. Darum muss sowohl ein Key wie auch ein Value angegeben werden. Wird ein bereits verwendeter Key angegeben, so wird das bisherige Value mit dem Neuen überschrieben. Der Methodenaufruf läuft synchron ab und blockiert daher den Programmablauf.
Draw	Um eine grafische Repräsentation des Netzes zu erhalten, kann der Benutzer die Methode `Draw()` aufrufen. Die Methode liefert ein Objekt der .Net-Klasse `Bitmap` zurück. Dieses enthält die grafische Repräsentation und ermöglicht ein komfortables Anzeigen resp. Abspeichern.

Get	Das Abrufen eines Values wird durch die Methode `Get()` ermöglicht. Ein Value wird durch den in der Methode `Add()` angegebenen Key identifiziert. Dieser wird nun für das Abrufen eines Values benötigt. Die Methode blockiert bis ein Value zurückgeliefert wurde. Falls kein Value mit dem entsprechenden Key gefunden wurde, wird ein leerer `string` zurückgegeben.
GetInfo	Informationen über den lokal gehosteten `Peer` können über die Methode `GetInfo()` abgefragt werden. Die Informationen enthalten die Verbindungsdaten zum Root-, Parent- und Child-Nodes so wie die auf dem `Peer` gespeicherten Key-Value-Pairs.
Remove	In der DHT gespeicherte Values können mit der Methode `Remove()` gelöscht werden. Wie bei der Methode `Get()`wird der entsprechende Key für das zu löschende Value benötigt. Das gelöscht Value wird als `string` zurückgeben. Falls kein Value gefunden wurde, wird ein leerer `string` zurückgeliefert.
SetMinClones	Die minimale Anzahl Clones eines `Node`s ist durch die Methode `SetMinClones()` konfigurierbar. D.h. die Redundanz und Leistungskapazität eines `Node`s kann vom Benutzer frei gesetzt werden. Die neu gesetzte minimale Anzahl Clones wird im ganzen Netz bzw. auf jedem `Peer` aktualisiert.
Start	Die Methode `Start()` startet den Service. Ohne Aufruf dieser Methode sind die Funktionalitäten der DHT nicht nutzbar. Daher muss die Methode als Erste nach der Initialisierung des Services aufgerufen werden.
Stop	Wird der Service nicht mehr benötigt, kann mit der Methode `Stop()` den Service beendet werden und der lokale `Peer` kontrolliert vom Netz abgemeldet werden.
TimberService	Die Klasse `TimberService` bietet zwei Konstruktoren mit einer unterschiedlichen Anzahl Parametern. Der Konstruktor mit nur einem Parameter dient für die Erstellung eines neuen Root-Nodes bzw. Netzes. Der Konstruktor mit zwei Parametern hingegen wird für das Beitreten in ein bestehendes Netz verwendet. Dabei ist zu beachten, dass mindestens ein `Peer`, welcher sich bereits im Netz befindet, bekannt sein muss. Die URL des bekannten `Peer`s wird als Wert des Parameters `boostrappingUrl` angegeben.

Tabelle 8-1: API-Beschreibung

Die Verwendung der DHT wird in vier Schritte beschrieben. Jeder Schritt wird anhand einer Abbildung bzw. Beispiel-Code erläutert.

Der zu verwendende Service liegt in Form einer Bibliothek vor. Daher muss im ersten Schritt die Bibliothek zu Beginn dem eigenen Projekt als Referenz hinzugefügt werden. Über einen Rechtsklick auf den Ordner „Reference“ im eigenen Projekt lässt sich eine neue Referenz hinzufügen.

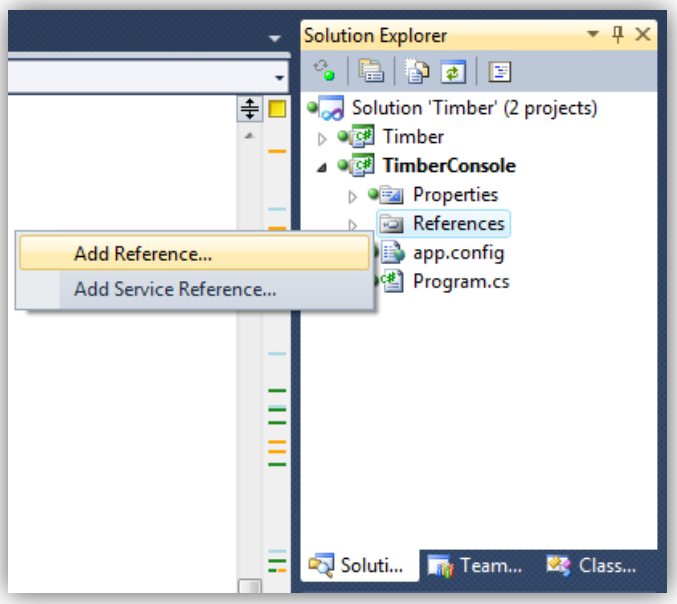


Abbildung 8-2: Bibliothek als Referenz hinzufügen

Über den erschienenen Dialog kann die Bibliothek „Timber.dll“ ausgewählt werden. Mit einem Klick auf „OK“ wird die Referenz dem Projekt hinzugefügt.

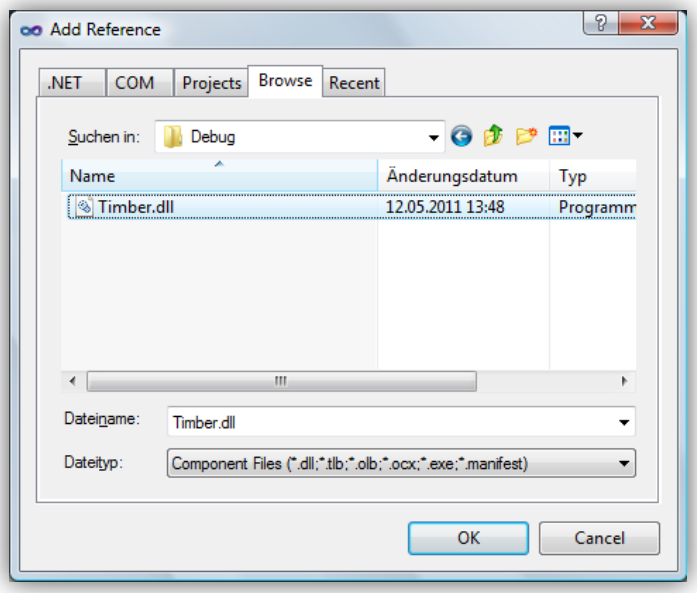


Abbildung 8-3: Bibliothek auswählen

Der nachstehende Code zeigt in einem zweiten Schritt wie der Service initialisiert, gestartet und konfiguriert wird. Zu Beginn wird unterschieden, ob ein neuer Tree erstellt wird oder der lokale `Peer` einem bestehenden Netz beitritt. Anschliessend wird der Service mit der Methode `Start()` gestartet. Bevor diese Methode aufgerufen wird, ist das Hinzufügen, Löschen und Updaten von Key-Value-Pairs nicht möglich. Nachdem der Service erfolgreich gestartet wurde, kann mit der Methode `SetMinClones()` die minimale Anzahl Clones eines `Node`s konfiguriert werden.

```
TimberService timber;

if (startAsRoot)
{
	//Start as root
	timber = new TimberService("http://127.0.0.1:8000");
}
else
{
	//Join the net as normal peer
	timber = new TimberService("http://127.0.0.1:8000",
				"http://192.168.131.94:8006");
}

//Start the service
timber.Start();
//Configurate the service
timber.SetMinClones(2);
```

Tabelle 8-2: Service initialisieren und starten

Im dritten Schritt wird der Service verwendet. Es werden Values hinzugefügt, abgefragt und gelöscht. Am Ende wird ein Bild der Netzstruktur abgespeichert und Informationen über den `Peer` ausgegeben.

```
//Save some values
timber.Add("Song1", "Let It Be");
timber.Add("Song2", "Smoke On the Water");
timber.Add("Movie1", "Avatar");
timber.Add("Movie2", "Titanic");
timber.Add("Foo", "Bar");

//Get the values with those keys
string movies = timber.Get("Movie1");
string songs = timber.Get("Song1");

//Remove the value with the Key "Foo"
timber.Remove("Foo");

//Get the visual representation of the DHT and save it as a PNG-file
timber.Draw().Save("Tree.png", ImageFormat.Png);

//Get the peer informations and print it on the console
Console.WriteLine(timber.GetInfo());
```

Tabelle 8-3: Verwendung des Services

Die folgende Abbildung zeigt die durch die Methode `Draw()` erstelle grafische Repräsentation des Netzes.

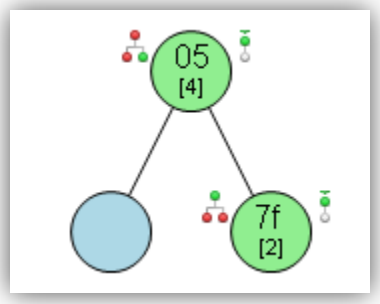


Abbildung 8-4: Gespeichertes Bild

Wird die Methode Draw() aufgerufen wird nebst der grafischen Repräsentation eine CSV-Datei erstellt. Folgende Abbildung zeigt ein Ausschnitt einer CSV-Datei.

	A	B	C	D	E	F	G
1	Key	Clones	Root-Node	Parent-Node	Left-Node	Right-Node	Values
2	05b954008a0aef8c29f14b84c6e511c96146d910	http://192.168.1.5:8000, http://192.168.1.5:8003, http://192.168.1.5:8004, http://192.168.1.5:8006,	http://192.168.1.5:8000, http://192.168.1.5:8003, http://192.168.1.5:8004, http://192.168.1.5:8006,			http://192.168.1.5:8002, http://192.168.1.5:8005,	Let It Be, Avatar, Titanic,
3							
4	7ff76ac87370d37e88b0ec74322a4078c4081644	http://192.168.1.5:8002, http://192.168.1.5:8005,	http://192.168.1.5:8000, http://192.168.1.5:8003, http://192.168.1.5:8004, http://192.168.1.5:8006,	http://192.168.1.5:8000, http://192.168.1.5:8003, http://192.168.1.5:8004, http://192.168.1.5:8006,			Smoke On The Water,

Abbildung 8-5: Erstelle CSV-Datei

Der folgende Printscreen zeigt die Ausgabe der Informationen über den `Peer` auf einer Konsole, welche mit der Methode `GetInfo()` abgefragt wurden.

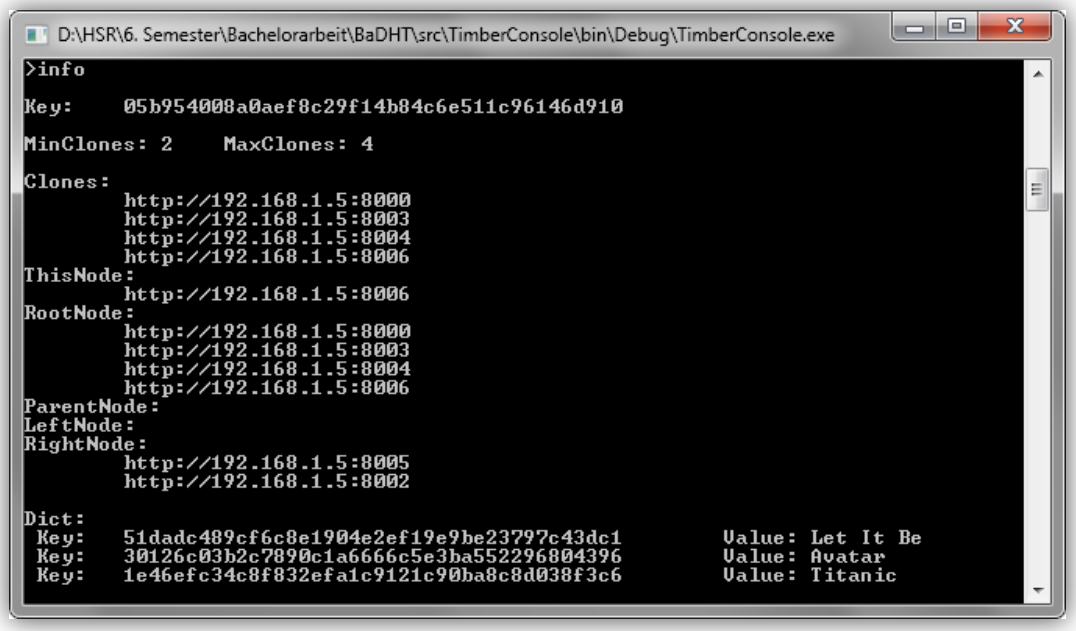


Abbildung 8-6: Informationsausgabe auf einer Konsole

Im letzten Schritt wird der `Peer` vom Netz abgemeldet und der Service somit beendet.

```
//Logout peer and stop the service
timber.Stop()
```

Tabelle 8-4: Beendigung des Services

9 Reflektionen

9.1 Pair-Programming

Während der gesamten Zeit wurde hauptsächlich zusammen an einer Aufgabe gearbeitet. Wir sind der Meinung, dass diese Art der Zusammenarbeit sich positiv auf das Resultat und dessen Qualität ausgewirkt hat. Bestimmte komplexe Probleme konnten so effizient und vermutlich besser als in Einzelarbeit bewältigt werden.

In Zukunft wird Pair-Programming von uns weiterhin bevorzugt.

9.2 Asynchronität

Asynchrone Methodenaufrufe klingen in der Theorie praktisch und relativ simple. Während der vorliegenden Bachelorarbeitet wurde jedoch festgestellt, dass Asynchronität nicht nur Vorteile besitzt und Probleme löst, sondern auch die Komplexität erhöht und andere Probleme verursacht. So konnten wir uns beispielsweise nicht mehr darauf verlassen, dass die Reihenfolge der Methodenaufrufe zwingend der Ausführungsreihenfolge entspricht. Dies führte beim Sammeln der Informationen für die grafische Darstellung des Netzes zu einem komplexeren Algorithmus als dies bei synchronen Aufrufen der Fall gewesen wäre.

Bei weiteren Projekten sollten die Pros und Contras der Asynchronität stets berücksichtigt werden.

9.3 Windows Communication Foundation

WCF erleichterte uns erheblich die Netzwerkkommunikation zu realisieren. So konnte Remote Procedure Call (RPC) schnell und komfortabel implementiert werden und die Verwendung von Stubs verursachte keine Schwierigkeiten.

Soll in Zukunft ein weiteres verteiltes System entwickelt werden, ist der Einsatz von WCF in Betracht zu ziehen.

9.4 Einsatz von Gelerntem

Während dem Projekt stiessen wir unerwartet auf Themen, welche wir in bestimmten Vorlesungen an der Schule behandelten. So kam es vor, dass wir vor einem Problem standen und bei der Suche nach einer Lösung auf ganz bestimmte Patterns oder Konzepte stiessen. Unter anderem wendeten wir die Patterns ACT, Template Method und Null-Object an. Des Weiteren machten wir uns Gedanken über das Konzept der logischen Zeit. Es war spannend etwas in der Praxis anwenden zu können von dem wir bis anhin nur immer die Theorie gehört hatten.

9.5 Komplexitätszunahme

Zu Beginn der Arbeit erschien der Einsatz eines BSTs für die Implementierung einer DHT als extrem einfach. Im Verlauf des Projekts wurde jedoch bemerkt, dass mit jeder zusätzlich implementierten Funktionalität die Komplexität unerwartet anstieg. Bei der Implementierung für das Entfernen eines Peers wurde dies speziell stark festgestellt.

Wir sind der Meinung, dass eine DHT grundsätzlich eine bestimmte Komplexität aufweist. So wurde auch unsere DHT komplexer als ursprünglich geplant war. Damit unsere DHT wirtschaftlich einsatzbar wäre, müssten noch weitere Funktionalitäten implementiert werden. D.h. die Komplexität und der Zeitaufwand würden weiter ansteigen.

9.6 Testschwierigkeiten

Bekannterweise ist das Testen verteilter Software System nicht trivial. Ein weiteres Mal wurde dies durch das Projekt am eigenen Leib erfahren. Es war teilweise schwierig ein Testszenario so zu erstellen, dass der Debugger am richtigen Ort auf die gewünschten Break-Points ansprang. Des Weiteren ist das Durchführen von Unit-Tests in der bekannter Form kaum möglich. Darum wurden andere Testkonzepte wie zum Beispiel die grafische Repräsentation des Trees verwendet. Siehe [Abbildung 6-32: Grafische Darstellung eines Trees].

10 Erfahrungsberichte

10.1 Martin Weber

Zu Beginn des 6. und voraussichtlich letzten Semesters des Bachelorstudiums starteten wir auch mit der Bachelorarbeit. Die Grundidee der zu erstellenden Arbeit war bereits bekannt. Diese Ideen wurden in den ersten Wochen zu geeigneten Konzepten ausgearbeitet. Während dieser Zeit fanden lange und ausgiebige Diskussionen im Team statt. So konnten Philippe und ich oft Stundenlang unsere Meinungen über neue oder verbesserte Konzepte austauschen. Ich bin überzeugt, dass gerade diese Diskussionen wichtig waren und für gute Ergebnisse im Projekt führten.

Nachdem keine Schwachstellen mehr in den Konzepten gefunden wurden, begannen wir mit deren Implementation. Hierbei wurde oft zusammen an einem Bildschirm gearbeitet und gemeinsam eine Lösung gesucht und programmiert. Die Komplexität der Implementation nahm bei jeder hinzukommenden Funktionalität unerwartet zu, weswegen sich Pair-Programming als besonders nützlich erwies. Ich denke, dass diese spezielle Form der Zusammenarbeit zu einer guten und qualitativ besseren Arbeit führte. Die Dokumentation wurde während dieser Zeit etwas vernachlässig. Es war nun also an der Zeit dies zu ändern. Dabei wurden die Konzepte detailliert beschrieben und anhand erstellter Bilder erläutert. Mit Hilfe von Sequenzdiagrammen wurden die implementierten Funktionalitäten erklärt. Dies nahm besonders viel Zeit in Anspruch, da bei kleinsten Änderungen im Code die Sequenzdiagramme wieder neu erstellt werden mussten. Es war also ein guter Entscheid mit der Beschreibung der Implementation erst etwas später zu beginnen. Dadurch konnte viel mühsame Arbeit erspart werden.

Die Dokumentation nahm viel Zeit in Anspruch, weswegen wir gegen Ende nur noch wenige Erweiterungen implementieren. Im Grossen und Ganzen kann ich auf eine, für mich erfolgreiche, Arbeit zurück blicken. Die Zusammenarbeit mit Philippe war äusserst angenehm und lehrreich. Zusätzlich fanden wir mit Herrn Joller einen hervorragenden Betreuer, der uns sowohl grosse Freiheit bei der Entwicklung, wie auch Unterstützung in schwierigen Situationen bot.

10.2 Philippe Morier

Zu Beginn der Bachelorarbeit starteten wir mit der Ausarbeitung der grundlegenden Konzepte. Anschliessend begannen wir mit der Implementierung. Dieses Vorgehen empfand ich als sehr motivierend, da so die eigenen Ideen und Konzepte sofort umgesetzt werden konnten ohne vorhin alles bis ins letzte Detail dokumentieren zu müssen. Im Gegenzug gab es dafür eine umso intensivere Dokumentierungszeit. Speziell das Erstellen der Sequenzdiagramme nahm viel Zeit in Anspruch und war nicht immer ganz trivial.

Während der ganzen Zeit arbeiteten Martin und ich gemeinsam am selben Problem. Das Pair-Programming empfand ich als sehr spannend und interessant. Die dabei entstanden Zwischendiskussionen verhalfen uns oft in einer kurzen Zeit zu einem grösseren Fortschritt. Ich behaupte, dass wir durch das Pair-Programming in der Lage waren, komplexere Probleme zu analysieren und zu lösen als dies in Einzelarbeit möglich gewesen wäre. Allgemein war die Zusammenarbeit mit Martin sehr angenehm und lehrreich. Weitere Projekte mit Martin kann ich mir ohne weiteres gut vorstellen. Die super Unterstützung durch Herrn Joller trug zu einer guten Zeit während dem Projekt bei. Wir hatten immer die Möglichkeit Fragen zu stellen und erhielten stets gute Ideen und Vorschläge. Herr Joller liess uns einen grossen Freiheitsgrad für unser Vorgehen. So war es möglich, dass wir sehr selbständig arbeiten konnten. Auch bezüglich der Dokumentation wurden keine unangenehmen Rahmenbedingungen gestellt oder zusätzliche Artefakte verlangt. Dies war äusserst motivierend.

Die Arbeit war spannend, da das Endresultat nicht bereits im Vorhinein detailliert vorgegeben war. Unsere Ideen hatten direkten Einfluss auf das Ergebnis. Diese Möglichkeit der Beeinflussung fand ich super. Mit dem Ergebnis der Arbeit bin ich sehr zufrieden. Ich bin der Meinung, dass wir eine gute DHT realisieren konnten. Die gesamte Arbeit verlief reibungslos. Ich war stets motiviert und konnte viel lernen. Weitere Projekte in Zusammenarbeit mit Martin und Herrn Joller kann ich mir gut vorstellen und würde ich mich darauf freuen.

11 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	42.2	40
2	42.2	59.5
3	42.2	48
4	42.2	60.5
5	42.2	57
6	42.2	48.5
7	42.2	51.5
8	42.2	51
9	42.2	26.75
10	42.2	55
11	42.2	55
12	42.2	51
13	42.2	55
14	42.2	53
15	42.2	31
16	42.2	29
17	42.2	6
Total	**720.8**	**777.75**

Tabelle 11-1: Geleistet Arbeitsstunden

Das folgende Diagramm zeigt die oben dargestellten Daten grafisch dar.

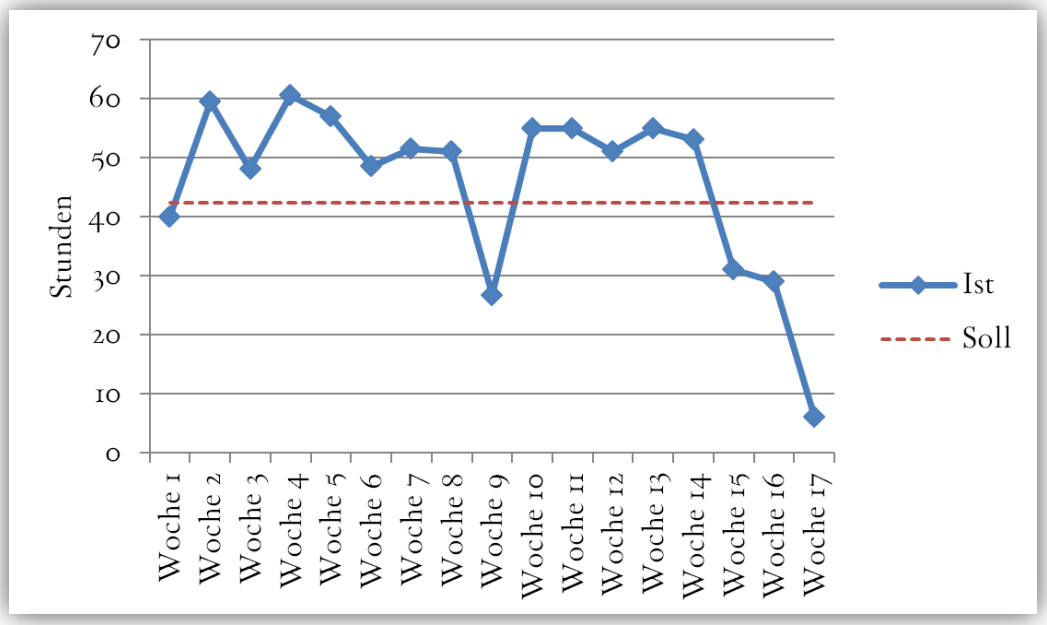


Abbildung 11-1: Geleistete Arbeitsstunden

12 Glossar

DHT	Distributed Hash Table (verteilte Hashtabelle), eine Datenstruktur in der Informatik.
BST	Binary-Search-Tree bzw. binärer Suchbaum.
ACT	Asynchronous Completion Token
DynDNS	Ist ein System, das in Echtzeit Domain-Name-Einträge aktualisieren kann. Für die Arbeit könnte der Service www.dyndns.com verwendet werden. [4]
RPC	Remote Procedure Call ist eine Technik zur Realisierung von Interprozesskommunikation.
Stub	Dient als Proxy bzw. Stellvertreten für den Aufruf von entferntem Code.
URL	Uniform Resource Locator identifizieren und lokalisieren Ressourcen in Computernetzwerken.
WCF	Das Hauptanwendungsgebiet von Windows Communication Foundation liegt in der Entwicklung Service-orientierter Architekturen. WCF ist ein Teil des .Net Frameworks. [5]
SOAP	Das Simple Object Access Protocol ist ein Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls durchgeführt werden können. [6]

13 Abbildungsverzeichnis

14 Tabellenverzeichnis

15 Referenzen

[1] http://en.wikipedia.org/wiki/File:Binary_search_tree.svg, 14.03.2011
[2] http://commons.wikimedia.org/wiki/File:Sorted_binary_tree_inorder.svg, 15.03.2011
[3] http://de.wikipedia.org/wiki/Happened-Before, 06.04.2011
[4] http://de.wikipedia.org/wiki/DynDNS, 10.04.2011
[5] http://de.wikipedia.org/wiki/Windows_Communication_Foundation, 11.04.2011
[6] http://de.wikipedia.org/wiki/SOAP, 16.04.2011