Diplomarbeit

Synchronisation von verteilten Verzeichnisstrukturen

Thorsten Schütt
Fakultät IV
Technische Universität Berlin
Matr. Nr. 180414

Berlin, März 2002

Betreuer
Prof. Dr. Sergei Gorlatch, TUB
Prof. Dr. Alexander Reinefeld, HUB und ZIB

Danksagung

Ich möchte mich bei Prof. Reinefeld und Prof. Gorlatch bedanken. Außerdem danke ich meinem Betreuer Florian Schintke für seine konstruktive Kritik an meiner Arbeit und Donald E. Knuth für TEX.

Schließlich möchte ich meinen Eltern danken, ohne deren Unterstützung diese Arbeit nicht möglich gewesen wäre.

Zusammenfassung

Das Management von großen Datenmengen spielt eine immer wichtigere Rolle, wie aktuelle Entwicklungen in der Hochenergiephysik [2] zeigen. Für das DataGrid-Projekt zum Beispiel ist es notwendig, große Datenmengen auf mehrere Rechenzentren in Europa zu verteilen und die Daten untereinander zu synchronisieren. Auch innerhalb von Clustern gewinnen mit zunehmender Anzahl der Knoten Werkzeuge zur effizienten Synchronisation und Verteilung von Daten an Bedeutung.

Im Rahmen dieser Arbeit wurde ein effizientes Verfahren zur Synchronisation von verteilten Verzeichnisstrukturen entwickelt und implementiert. Mit diesem Verfahren ist es möglich, unabhängige Änderungen an beliebigen Repositories gleichzeitig durchzuführen. Das Verfahren benötigt keine zentrale Instanz, wodurch eine gegenüber vielen existierenden Verfahren verbesserte Skalierbarkeit erreicht werden konnte. Dabei wurden Erkenntnisse aus der Graphentheorie eingesetzt und weiterentwickelt, um die Netzwerktopologie und -bandbreiten zwischen den Rechnern zu berücksichtigen. Durch die Verwendung einer Offline-Synchronisation werden Änderungen erst dann an andere Rechner propagiert, wenn der Nutzer dies anstößt. Das kann zum Beispiel nach einer abgeschlossenen Transaktion, die Änderungen an mehreren Dateien beinhaltet, angemessen sein.

Abstract

Current developments in high energy physics [2] show that the management of large datasets plays an important role. For the DataGrid project it is necessary to distribute large datasets over several computing centers all over Europe and to synchronize these datasets. Within clusters tools for efficient synchronization and distribution of data become more important, too.

In this thesis, a method to synchronize distributed directory structures was developed and implemented which makes it possible to perform independent changes to arbitrary repositories simultaneously. This method needs no central instance and therefore the presented system achieves a better scalability than many existing systems. Knowledge from graph theory was used and improved to take the network topology and the network bandwidth between the computers into account. By using offline synchronization, changes will only be propagated when the user initiates it. This can be reasonable after a completed transaction which consists of changes on several files.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Thema dieser Arbeit

Im europäischen Zentrum für Nuklearforschung (Cern) in Genf entsteht derzeit ein neuer Teilchenbeschleuniger für Experimente der Hochenergiephysik, der in den Jahren 2005/2006 in Betrieb genommen werden soll. Von diesem Zeitpunkt an werden bei den vier Experimenten (Alice, Atlas, CMS, LHCb) [2], die auf dem Beschleuniger angeordnet sein werden, jeweils Meßdaten von mehreren Petabytes pro Jahr zu analysieren sein.

Im von der Europäischen Union geförderten DataGrid-Projekt [7] entsteht für diese Experimente eine Middleware zur Verteilung von Daten und Analyseprozessen über mehrere Rechenzentren in Europa. Um den Zugriff auf die Daten zu beschleunigen, und um sich gegen Datenverluste abzusichern, werden in den Rechenzentren Kopien der Meßdaten vorgehalten.

Bei der Auswertung der Experimente überwiegt die Anzahl lesender Zugriffe auf die Meßdaten. Wenn durch schreibende Zugriffe Dateien verändert werden, müssen alle Kopien dieser Dateien miteinander synchronisiert werden, damit alle Forscher die gleichen Dateien nutzen können. Die Synchronisation für dieses Szenario kann effizient gestaltet werden, indem die Dateien nicht sofort nach jeder Änderung synchronisiert werden, sondern mehrere Veränderungen in einer Synchronisation zusammengefaßt werden. Die Kopien befinden sich in diesem Fall für einen kurzen Zeitraum in unterschiedlichen Zuständen. Diese Form der Synchronisation wird Offline-Synchronisation genannt und läßt sich ohne Eingriffe in den Kern eines Betriebsystems implementieren. Solche Systeme können dadurch ohne Hilfe des Administrators vom Benutzer installiert werden, was deren Nutzung vereinfacht.

Probleme entstehen bei der Offline-Synchronisation, wenn mehrere Kopien einer Datei, die seit der letzten Synchronisation unterschiedlich verändert wurden, synchronisiert werden sollen. Dabei können zwei Arten von Problemen auftreten. Entweder muß dem System die Semantik der Datei bekannt sein, wodurch es die verschiedenen Versionen der Datei zu einer Version zusammenführen kann, oder es wird ein Konflikt gemeldet, der vom Benutzer gelöst werden muß. Ein Beispiel für den ersten Fall wäre ein verteilter Terminkalender, zu dem Einträge sowohl auf einem PC als auch auf einem PDA hinzu-

gefügt wurden. Wenn der Synchronisationssoftware die Semantik von Terminen bekannt ist, kann sie beide Versionen des Terminkalenders vereinen.

In dieser Arbeit wird eine effiziente Methode zur Synchronisation von Repositorysystemen vorgestellt. Ein Repositorysystem soll hier eine Menge sogenannter Repositories sein, die die gleichen Dateien als Kopien enthalten. Ein Repository ist dabei allgemein eine beliebige Menge von Dateien. Die Repositories eines Repositorysystems sollen über mehrere Rechner verteilt sein können. Veränderungen sollen nicht auf ein bestimmtes Repository beschränkt sein, sondern an beliebigen Repositories vorgenommen werden können. Konflikte müssen vom Benutzer gelöst werden, da sich selbst unter Berücksichtigung der Semantik Konflikte nicht immer eindeutig lösen lassen.

Das entwickelte Verfahren basiert auf der Synchronisation von jeweils zwei Repositories miteinander. Es können gleichzeitig mehrere dieser zweiseitigen Synchronisationen durchgeführt werden, wenn sie unterschiedliche Repositories betreffen. Um das gesamte System zu synchronisieren, müssen die zweiseitigen Synchronisationen so komponiert werden, daß jedes Repository direkt oder indirekt mit jedem anderen synchronisiert wird. Die Komposition entspricht einem All-to-All Broadcast. In der Graphentheorie wird diese Form der Kommunikation als Gossip bezeichnet [11].

Für die Laufzeit der Synchronisation ist es wichtig, daß die Netzwerktopologie zwischen den Repositories berücksichtigt wird. Deshalb wurden im Rahmen dieser Arbeit mehrere Methoden entwickelt, mit denen sich die Gesamtsynchronisation an verschiedene Netzwerktopologien und -bandbreiten anpassen läßt.

1.2 Aufbau dieser Arbeit

Zunächst wird in Kapitel 2 das Thema dieser Arbeit präzisiert und auf Randbedingungen eingegangen. Anschließend werden neue Lösungsansätze zur effizienten Synchronisation von Repositorysystemen diskutiert. Außerdem werden existierende Lösungen vorgestellt.

In Kapitel 3 werden Verfahren zur effizienten Durchführung von Punkt-zu-Punkt-Synchronisationen vorgestellt. Anhand von Messungen wird gezeigt, daß die Auswahl der Übertragungs- und Kompressionsverfahren von der Netzwerkbandbreite abhängig gemacht werden muß.

Kapitel 4 stellt Algorithmen für All-to-All Broadcasts vor, die an vollständige Graphen, Ketten und Ringe angepaßt sind. In Kapitel 5 werden Broadcast-Verfahren für Tori und hierarchische Topologien vorgestellt, die auf den Algorithmen aus Kapitel 4 basieren. Die Effizienz der Verfahren wird anhand eines Modells zur Laufzeitabschätzung verglichen. Zusätzlich wird eine XML-basierte Sprache definiert, mit der die verschiedenen Topologien beschrieben werden können.

Im Rahmen dieser Arbeit ist eine Implementierung der in den Kapiteln 3 - 5 präsentierten Ideen entstanden. In Kapitel 6 werden das Design und Implementierungsdetails vorgestellt, und in Kapitel 7 werden Installation, Konfiguration und Betrieb der Implementierungen beschrieben.

Kapitel 2

Synchronisation verteilter Verzeichnisstrukturen

In diesem Kapitel werden einige Designentscheidungen motiviert, die bei der Entwicklung der hier vorgestellten Methode zur Synchronisation getroffen wurden. Dazu werden einige Merkmale erläutert, anhand derer sich Synchronisationssysteme unterscheiden lassen. Anschließend werden existierende Systeme vorgestellt. Aus den existierenden Systemen werden die Merkmale herausgearbeitet, die ein System erfüllen muß, das ein breites Anwendungsgebiet abdecken soll.

2.1 Unterscheidungsmerkmale

2.1.1 Kommunikationsarten

Es werden verschiedene Kommunikationsarten anhand der Anzahl der Rechner, die ein gesendetes Datum empfangen, unterschieden.

Broadcasting: Beim Broadcasting wird jedes gesendete Datum von allen Rechnern, die an dasselbe Netzwerk wie der Sender angeschlossen sind, empfangen. Sowohl das Ethernet-Protokoll [12], das in LANs (Local Area Network) verbreitet ist, als auch das IP-Protokoll [24], das die Grundlage für das Internet bildet, unterstützen durch besondere Adressierung Broadcasting.

Multicasting: Beim Multicasting wird jedes Datum an eine genau spezifizierte Gruppe von Rechnern gesendet. Im IP-Protokoll ist Unterstützung von Multicasting durch besondere Adressen und zusätzliche Protokolle vorgesehen. Für die Synchronisation von Repositories ist diese Form der Kommunikation geeignet, da die Veränderungen jedes Repositories direkt an alle anderen Repositories gesendet werden können.

Unicasting: Unicasting bedeutet, daß es für jedes gesendete Datum genau einen Empfänger gibt. Diese Kommunikationsform wird beim TCP [25]-Protokoll, das zur

Familie der Internet-Protokolle gehört, eingesetzt. Punkt-zu-Punkt-Verbindungen basieren auf Unicasting.

Für diese Arbeit ist Broadcasting unter Sicherheitsaspekten nicht geeignet, da nicht immer alle Rechner eines Subnetzes zu einem Repositorysystem gehören und dadurch auch nicht beteiligte Rechner die Daten empfangen können. Multicasting kann auch nicht verwendet werden, da effiziente Multicasting-Implementierungen nicht weit verbreitet sind.

2.1.2 Offline-/Onlinesynchronisation

Synchronisationssysteme können anhand des Zeitpunktes unterschieden werden, zu dem sie Veränderungen an den Repositories austauschen. Folgendes soll in dieser Arbeit unter Online- und Offlinesynchronisation verstanden werden:

Onlinesynchronisation: Bei der Onlinesynchronisation werden Änderungen an Dateien sofort in allen Repositories sichtbar. Dafür ist es notwendig, daß ständig eine Kommunikationsverbindung zwischen den Repositories besteht (online). Da Veränderungen der Dateien sofort propagiert werden müssen, sind zum Erkennen der Veränderungen im Allgemeinen Eingriffe in den Kern der Betriebssysteme nötig. Eingriffe in den Kern dürfen allerdings nur von Administratoren durchgeführt werden, weshalb solche Onlinesynchronisationssysteme nicht vom Benutzer installiert werden können.

Systeme zur Onlinesynchronisation sind meistens als Dateisystem implementiert und deshalb für Benutzer bei der Verwendung transparent.

Offlinesynchronisation: Bei der Offlinesynchronisation werden die Änderungen an Dateien erst in alle Repositories propagiert, wenn die Synchronisation explizit angestoßen wird. Deshalb wird keine dauerhafte Kommunikationsverbindung zwischen den Repositories (offline) benötigt. Die Repositories haben unmittelbar nach der Synchronisation den gleichen Inhalt. Zwischen den Synchronisationen können die Inhalte der Repositories voneinander abweichen.

Systeme, die diese Synchronisationsform nutzen, können im Allgemeinen auch Benutzer installieren.

2.1.3 Master-Slave

Einige Synchronisationssysteme erlauben es dem Benutzer, nur an einem besonderen Repository Veränderungen vorzunehmen. Diese werden Master-Slave-System genannt. Benutzerfreundlicher ist es, wenn an beliebigen Repositories Veränderungen vorgenommen werden dürfen.

2.1.4 Unterstützte Dateitypen

Wenn zwei Replikate einer Datei unterschiedlich verändert wurden, liegt ein Konflikt vor. Wenn dem Synchronisationssystem die Semantik dieser Dateien bekannt ist, können beide Versionen der Datei zusammengefaßt werden. Solche Systeme können zum Beispiel die Adressdatenbank eines PDAs mit der eines PCs synchronisieren, auch wenn in beide Datenbanken neue Adressen eingefügt wurden.

Systeme, die keine Informationen über die Semantik der Datei besitzen, können solche Konflikte nicht lösen. Die Konflikte müssen in diesem Fall vom Benutzer gelöst werden.

2.2 Einordnung existierender Systeme

Im folgenden Abschnitt werden existierende Synchronisationssysteme vorgestellt und anhand der in Abschnitt 2.1 eingeführten Kriterien klassifiziert. Zunächst werden die Systeme nach Online- und Offlinesynchronisation aufgeteilt.

2.2.1 Onlinesynchronisation

Ein System mit Onlinesynchronisation kann realisiert werden, indem die Dateien auf einem File-Server gespeichert werden. Die Dateien auf dem File-Server bilden ein Repository. Die restlichen Repositories entstehen, wenn die Dateien auf den Clients in die lokale Verzeichnisstruktur eingeblendet (mount) werden. Physikalisch sind die Dateien nur einmal auf dem Server gespeichert. Auf den Clients existieren nur virtuelle Repositories. Da von den Dateien nur eine physikalische Kopie existiert, ist eine Synchronisation nicht nötig.

Solche Systeme haben zwei Schwachstellen, die beide auf die zentrale Speicherung der Dateien zurückzuführen sind: (a) bei einem Ausfall des File-Servers kann in keinem Repository mehr auf die Dateien zugegriffen werden, (b) solche Systeme skalieren nicht, da alle Dateizugriffe über den File-Server laufen müssen.

Ein Beispiel für solch ein System ist ein NFS [31]-Server. NFS ist ein im Unix-Bereich weit verbreitetes Protokoll für File-Server. Bei NFS können keine Konflikte auftreten, da nie zwei Versionen einer Datei existieren können.

Coda [28] implementiert wie NFS einen File-Server, versucht aber, die genannten Nachteile zu mildern. Bei Coda wird durch Client-seitiges Caching der Zugriffe auf den Server versucht, die Skalierbarkeit zu erhöhen. Die Entwickler gingen beim Design von Coda davon aus, daß lesende Zugriffe auf die Dateien des Servers überwiegen. Durch File-Caches auf den Clients können viele Dateizugriffe ohne die Hilfe des Servers durchgeführt werden. Damit wird die Last am Server reduziert und die Skalierbarkeit erhöht. Bei einem Ausfall des Servers können lesende Zugriffe auf Dateien aus dem Cache bedient werden. Schreibende Zugriffe werden bis nach dem Ausfall des Servers lokal protokolliert. Bei Wiederherstellung der Verbindung zum Server werden die protokollierten Schreibzugriffe zum Server übertragen. Für den Fall, daß es dabei zu Konflikten zwischen

Schreibzugriffen kommt, existiert die Möglichkeit „application specific resolver“ zu entwickeln und in Coda einzubinden. Diese können für die Dateien jeweils eines Dateityps Konflikte lösen.

Coda wurde an der Carnegie Mellon University (CMU) entwickelt und befindet sich dort im täglichen Einsatz. Das gleichfalls an der CMU entwickelte Intermezzo [3] ist eine weiterentwickelte Version von Coda mit weiter verbesserten Skalierungseigenschaften.

NFS, Coda und Intermezzo benutzen sternförmige Punkt-zu-Punkt-Verbindungen zur Kommunikation zwischen den Repositories. Bei allen drei Systemen können an jedem Repository Veränderungen vorgenommen werden. Sie unterscheiden sich aber bei der Behandlung von Konflikten.

Wenn bei Coda oder Intermezzo der Server nicht erreichbar ist, können auf den Clients unterschiedliche Versionen derselben Datei entstehen. Coda bietet mit dem „application specific resolver“ die Möglichkeit, Konflikte für einige Dateitypen zu lösen, während bei Intermezzo alle Konflikte vom Benutzer gelöst werden müssen.

2.2.2 Offlinesynchronisation

Viele Programme, die mit Offlinesynchronisation arbeiten, unterstützen nur zwei Repositories pro Repositorysystem und sind applikationsspezifisch. Diese Programme können zum Beispiel die auf einem Personal Digital Assistant (PDA) gespeicherten Adressen mit den auf einem PC gespeicherten Adressen synchronisieren.

Im Rahmen dieser Arbeit ist ein Synchronisationssystem entwickelt worden, das zum Beispiel im DataGrid-Projekt eingesetzt werden kann. Für dieses Anwendungsgebiet ist es notwendig, daß das System Repositorysysteme mit vielen Repositories synchronisieren kann und applikationsunabhängig arbeitet. Im folgenden werden deshalb nur Programme vorgestellt, die die Semantik der Dateien in den Repositories nicht beachten und Unicasting zur Kommunikation zwischen den Repositories verwenden. Programme, die andere Kommunikationsarten verwenden, konnten nicht gefunden werden.

`reconcile`, das am Mitsubishi Electric Research Laboratory (MERL)[1] entwickelt wurde, kann mehr als zwei Repositories miteinander synchronisieren, wobei diese sowohl auf Windows- als auch auf Unix-Rechnern gespeichert sein können. Mit `reconcile` wurden nur interne Tests am MERL durchgeführt. Das Programm soll kommerziell vertrieben werden, aber darüber gibt es bisher keine weiteren Informationen. Über den internen Aufbau des Programms gibt es keine Veröffentlichungen.

Bei `unison`[2] können wie bei `reconcile` die Repositories auf Windows- und Unix-Rechnern gespeichert sein. Es werden nur Repositorysysteme, die aus zwei Repositories bestehen, unterstützt.

Das Programm `rsync` kann effizient zwei Repositories miteinander synchronisieren, wobei nur an einem Änderungen vorgenommen werden dürfen. Bei veränderten Dateien wird nur die Differenz zwischen der alten und der neuen Version übertragen. Der Algorithmus, den `rsync` nutzt, wird im Abschnitt 3.2.3 vorgestellt und wurde in [33] untersucht.

[1] http://www.merl.com/projects/reconcile/
[2] http://www.cis.upenn.edu/~bcpierce/unison/

2.3 Schlußfolgerungen für ein neues Tool

Aus den vorgestellten Systemen lassen sich einige Eigenschaften ableiten, die für den Einsatz in einem breiten Anwendungsgebiet notwendig sind.

Kommunikation: Alle vorgestellten Systeme verwenden Unicast zur Kommunikation. Dieses sollte aus den in Abschnitt 2.1 genannten Gründen übernommen werden.

Administration: Installation und Betrieb sollen ohne Unterstützung durch den jeweiligen Administrator möglich sein, was besonders bei vielen über mehrere Administrationsdomänen verteilten Repositories wichtig ist.

Online-/Offline-Synchronisation: Um ein System zur Online-Synchronisation implementieren zu können, wären Eingriffe in den Kern des Betriebsystems nötig, damit Veränderungen sofort erkannt werden können. Da dafür root-Rechte nötig sind, sollte Offlinesynchronisation implementiert werden.

Konfliktlösung: Konflikte könnten nur vom System gelöst werden, wenn die Semantik der Dateien bekannt ist. Aber selbst wenn die Semantik bekannt wäre, können Fälle auftreten, in denen Konflikte nicht eindeutig gelöst werden können. Konflikte müssen deshalb gemeldet und vom Benutzer gelöst werden.

Master-Slave: Von den vorgestellten Systemen ist nur `rsync` darauf angewiesen, daß Veränderungen nur an einem Repository vorgenommen werden. Um das System benutzerfreundlich zu gestalten, sollten an jedem Repository Änderungen vorgenommen werden dürfen.

Für die Effizienz des Synchronisationssystems ist die Kommunikation zwischen den Repositories entscheidend. Dafür muß der lokale und der globale Aufwand der Kommunikation berücksichtigt werden. Um den lokalen Aufwand gering zu halten, synchronisieren sich die Rechner paarweise, dadurch hat jeder Rechner zu jedem Zeitpunkt maximal eine bidirektionale Netzwerkverbindung.

Zur Reduzierung des globalen Aufwands führen alle Repositories gleichzeitig einen Broadcast mit Hilfe eines Broadcast-Trees unter Berücksichtigung der Netzwerktopologie aus. Mit dem Broadcast teilt jedes Repository die Veränderungen, die an ihm durchgeführt wurden, den anderen mit. Bei dieser Kommunikationsstruktur handelt es sich um einen All-to-All-Broadcast. Effiziente Kommunikationsstrategien für diese Struktur werden in der Literatur zum Beispiel unter dem Stichwort Gossip-Problem [11] diskutiert. Das Gossip-Problem wird in Kapitel 4 genauer vorgestellt.

Kapitel 3

Punkt-zu-Punkt-Synchronisation

Bei der Synchronisation zweier Repositories sind zwei Punkte von Bedeutung: (a) es wird eine Methode benötigt, die die Dateien erkennt, die seit der letzten Synchronisation verändert wurden, und (b) es müssen effiziente Verfahren zur Übertragung der Veränderungen zwischen den Repositories verwendet werden. In den folgenden Abschnitt sollen diese Punkte detaillierter diskutiert werden.

3.1 Erkennen von veränderten Dateien

An Repositories können drei Arten von Veränderungen vorgenommen werden:

- Eine Datei wird einem Repository hinzugefügt.
- Eine Datei wird aus einem Repository entfernt.
- Der Inhalt einer existierenden Datei wird in einem Repository verändert.

Im folgenden werden zwei Verfahren zum Erkennen von Veränderungen an Repositories vorgestellt. Beide Verfahren speichern dazu Informationen über den Zustand des Repositories nach der letzten Synchronisation ab.

3.1.1 Berechnung von Prüfsummen

Das erste Verfahren verwendet zur Erkennung der Veränderungen am Repository Prüfsummen. Eine Prüfsumme einer Datei ist ein Wert, der mit Hilfe einer Formel aus dem Inhalt der Datei berechnet wird. Mit Prüfsummen läßt sich bis auf eine geringe Fehlerwahrscheinlichkeit feststellen, ob zwei Dateien unterschiedlichen Inhalt haben. Nach jeder Synchronisation werden zu jedem Repository Metadaten gespeichert. Diese bestehen aus den Namen aller Dateien und Verzeichnisse in dem Repository. Zusätzlich wird zu jeder Datei eine Prüfsumme des Inhaltes der Datei vermerkt.

Veränderungen an Repositories werden erkannt, indem für jede Datei im Repository die Prüfsumme berechnet wird und dieser Wert mit dem Wert aus den Metadaten verglichen wird. Wenn sich die beiden Prüfsummen unterscheiden, wurde die Datei seit der

letzten Synchronisation verändert. Dateien wurden dem Repository hinzugefügt, wenn sie sich im Repository befinden, aber ihr Name nicht in den Metadaten gespeichert ist. Eine Datei wurde aus dem Repository gelöscht, wenn sich ihr Name in den Metadaten befindet, aber im Repository keine Datei mit diesem Namen existiert.

Für dieses Verfahren muß bei jeder Synchronisation für die Berechnung der Prüfsummen der Inhalt aller Dateien im Repository gelesen werden. Wie das zweite Verfahren zeigen wird, ist es möglich die Veränderungen zu erkennen, ohne jede Datei komplett zu lesen.

3.1.2 Nutzung vorhandener Metadaten

Für das zweite Verfahren wird für jede Datei des Repositories ihr Name in den Metadaten gespeichert. Zusätzlich wird der Zeitpunkt der letzten Synchronisation des Repositorysystems vermerkt.

Unix-Betriebssysteme speichern für jede Datei den Zeitpunkt, an dem diese das letzte Mal verändert wurde. Um Dateien zu erkennen, die seit der letzten Synchronisation des Repositorysystems verändert wurden, muß für jede Datei der Zeitpunkt der letzten Änderung der Datei mit dem Zeitpunkt der letzten Synchronisation verglichen werden. Bei Dateien, die seit der letzten Synchronisation verändert wurden, liegt der Zeitpunkt der letzten Veränderung nach dem in den Metadaten gespeicherten Zeitpunkt der letzten Synchronisation. Gelöschte und hinzugefügte Dateien werden wie beim ersten Verfahren erkannt, indem die Namen der Dateien im Repository mit den in den Metadaten gespeicherten Namen verglichen werden.

Benutzern ist es mit Hilfe des Befehls `utime` möglich, den Zeitpunkt der letzten Veränderung von Dateien auf beliebige Werte zu setzen. Wird dieser Zeitpunkt in die Zukunft gesetzt, wird eine unter Umständen unveränderte Datei als verändert erkannt. Wird aber dieser Zeitpunkt vor den Zeitpunkt der letzten Synchronisation gesetzt, kann die jeweilige Datei verändert worden sein, ohne daß dieses Verfahren die Veränderung erkennen kann. Im ersten Fall werden unter Umständen zwei gleiche Dateien synchronisiert, im zweiten Fall können Inkonsistenzen zwischen den Repositories entstehen.

Die Uhren der Rechner, auf denen Repositories gespeichert sind, müssen nicht synchron laufen, da nie zwei Zeitpunkte verglichen werden, die mit unterschiedlichen Uhren gemessen wurden. Dadurch muß auch nicht berücksichtigt werden, ob sich die Rechner, auf denen die Repositories gespeichert sind, in unterschiedlichen Zeitzonen befinden.

Dieses Verfahren kann nicht alle Veränderungen erkennen, wenn der Benutzer den Modifikationszeitpunkt von Dateien direkt verändert. Der Aufwand dieses Verfahrens ist aber geringer als die Berechnung von Prüfsummen, und deshalb ist dieses Verfahren auch das geeignetere für diese Anwendung.

3.2 Effiziente Übertragungsverfahren

Wenn einem Repository eine Datei neu hinzugefügt wurde, kann diese komprimiert an andere Repositories versendet werden. Für den Fall, daß eine Datei verändert wurde,

muß nicht die komplette Datei übertragen werden sondern nur die Differenz zur alten Version.

Die Differenz zwischen zwei Dateien ist eine Folge von Anweisungen, mit denen sich die erste Datei in die zweite überführen läßt. Die Differenz kann daraus bestehen, die zweite Datei auf die Erste zu kopieren. Es ist auch möglich, daß die Differenz aus der Anweisung besteht, ein Byte an die erste Datei anzuhängen. In diesem Fall ist die zweite Differenz vorzuziehen, da sie sich kompakter speichern läßt. In Abschnitt 3.2.3 wird der Rsync-Algorithmus [33] vorgestellt, der die Differenz zwischen zwei Versionen einer Datei berechnet, ohne die gesamte Datei zu übertragen, wobei beide Versionen der Datei in unterschiedlichen Repositories gespeichert sind. Anschließend werden mehrere Übertragungsverfahren bei unterschiedlichen Netzwerkbandbreiten verglichen.

3.2.1 `diff` und `patch`

In Open-Source-Projekten ist es üblich, nicht nur die aktuelle Version des Quellcodes, sondern auch die Differenz der aktuellen Version zur letzten zu veröffentlichen. Die Differenz wird mit dem Programm `diff` berechnet. Die Differenzen zwischen zwei Versionen sind in vielen Fällen kleiner als der gesamte Quellcode einer Version. Wenn ein Benutzer den Quellcode der letzten Version besitzt, kann er mit Hilfe von `patch` die Differenz in seinen Quellcode integrieren und besitzt dann die aktuelle Version. In diesem Fall muß der Benutzer statt des kompletten Quellcodes der aktuellen Version nur die Differenz aus dem Internet herunterladen.

Der komprimierte Quellcode des Linux-Kernels [32] ist ca. 24 MB groß. Die Patches, die die Differenz zwischen den letzten beiden Versionen des Linux-Kernels enthalten, sind typischerweise kleiner als 2 MB.

Angenommen zwei Repositories enthalten verschiedene Versionen des Quellcodes des Linux-Kernels und sollen synchronisiert werden. Dem Synchronisationssystem sei das Repository mit der aktuelleren Version bekannt. Ein Verfahren könnte den kompletten Quellcode in dem Repository mit der neueren Version komprimieren und an das andere Repository senden. Dabei müßten ca. 24 MB an Daten (siehe oben) übertragen werden. Ein zweites Verfahren könnte mit Hilfe des Rsync-Algorithmuses die Differenz der beiden Versionen berechnen und diese an das Repository mit der älteren Version übertragen. Beim zweiten Verfahren müßten ca. 2 MB Daten übertragen werden. Mit dem zweiten Verfahren könnte in diesem Fall das zu übertragende Datenvolumen um 92% reduziert werden.

Für das Programm `diff` ist es im Allgemeinen nötig, daß sich beide Dateien auf demselben Rechner befinden. Für diese Arbeit ist dieses aber nicht möglich, da in jedem Repository immer nur eine Version gespeichert ist. Deshalb wird im folgenden der Rsync-Algorithmus vorgestellt, der dieses Problem löst.

3.2.2 Differenzen zwischen zwei Versionen einer Datei übertragen

Es wird nun ein Verfahren gesucht, das die Differenz zwischen zwei Dateien ermittelt, die auf verschiedenen Rechnern gespeichert sind, ohne daß einer der beteiligten Rechner

beide Dateien komplett vorzuliegen hat.

Im folgenden wird angenommen, daß sich auf dem ersten Rechner die Originaldatei und auf dem zweiten die veränderte befindet.

Das gesuchte Verfahren überträgt wenige Daten vom ersten Rechner zum zweiten, aus denen dieser erkennen kann, wie die Originaldatei verändert wurde. Im zweiten Schritt berechnet der zweite Rechner eine Folge von Anweisungen, mit denen sich die Originaldatei in die veränderte Datei überführen läßt. Die Anweisungen entsprechen der Differenz zwischen beiden Versionen. Anschließend werden die Anweisungen vom zweiten zum ersten Rechner übertragen und dort auf die Originaldatei angewendet.

Da im ersten Schritt möglichst wenig Daten übertragen werden sollen und über die Struktur der Dateien keine Informationen vorliegen, sie also binär behandelt werden, bietet es sich an, Prüfsummen[1] einzusetzen. Prüfsummen sind nur wenige Bytes groß und im Vergleich zu den Dateien relativ klein.

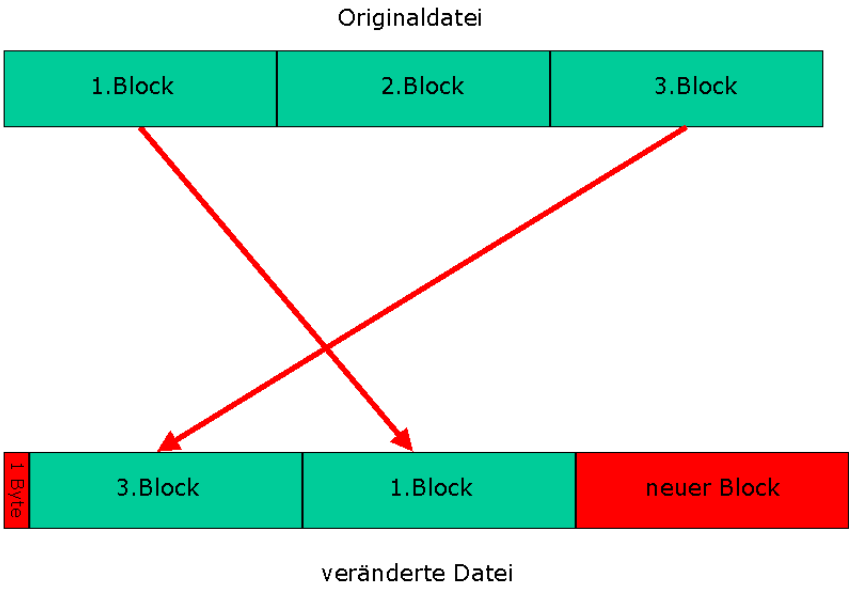

Abbildung 3.1: Beispiel für die Veränderung einer Datei

In Abbildung 3.1 wird ein Beispiel für die Veränderung einer Datei gezeigt. Die oberen Rechtecke repräsentieren die Originaldatei auf dem ersten Rechner und die unteren die veränderte Datei auf dem zweiten Rechner.

[1] engl.: checksum

Im folgenden wird der Rsync-Algorithmus [33] erarbeitet. Dabei wird das Verfahren zum Erkennen der Veränderungen an der Originaldatei schrittweise verbessert.

Unterteilung in Blöcke

Als erste Möglichkeit könnte über die gesamte Datei eine Prüfsumme berechnet und diese zum zweiten Rechner gesendet werden. Dieser kann nur feststellen, ob die Datei verändert wurde oder nicht. Wenn sie verändert wurde, müßte die komplette veränderte Datei an den ersten Rechner übertragen werden.

Also unterteilt der erste Rechner die Originaldatei in k Byte große Blöcke, berechnet für jeden dieser Blöcke eine Prüfsumme und sendet die Prüfsummen an den zweiten Rechner. Der zweite Rechner unterteilt die Datei ebenfalls in Blöcke von jeweils k Bytes und berechnet für jeden Block eine Prüfsumme. Die Prüfsummen beider Rechner werden anschließend miteinander verglichen. Dadurch kann erkannt werden, wenn sich die Reihenfolge der Blöcke gegenüber der Originaldatei geändert hat oder wenn neue Blöcke hinzugekommen sind. Wenn sich wie in Abbildung 3.1 Blöcke nicht um ein Vielfaches von k verschoben haben, kann die Verschiebung nicht erkannt werden. Für das Beispiel aus der Abbildung müßten alle Blöcke neu übertragen werden.

Prüfsummen an beliebigen Stellen

Der zweite Rechner könnte die Verschiebung erkennen, wenn bei jedem Byte in der veränderten Datei beginnend für die nächsten k Bytes eine Prüfsumme berechnet wird. Bei einer Dateilänge von l wäre der Aufwand für die Berechnung der Prüfsummen in $O(k \cdot l)$. Dieser Ansatz ist nicht praktikabel, da der Rechenaufwand zu hoch wäre.

„Rollende“ Prüfsummen

Die Lösung für das Problem sind sogenannte „rollende“ Prüfsummen[2]. Diese haben die Eigenschaft, daß, wenn sie für die ersten k Bytes ab dem i-ten Byte berechnet wurden, sie sich mit wenigen Operationen für die ersten k Bytes ab dem $i+1$-ten Byte berechnen lassen. Dafür wird das i-te Byte aus der Prüfsumme herausgerechnet und der Wert des $i+k+1$-ten Bytes zur Prüfsumme hinzugefügt. Mit „normalen“Prüfsummen wie MD5 [27] ist dies nicht möglich. Der Aufwand zum Berechnen der „rollenden“Prüfsummen über die gesamte Datei ist in $O(l)$.

Für den Rsync-Algorithmus wird $r(i,k)$ aus Abbildung 3.2 als „rollende“Prüfsumme benutzt, wobei i dem Offset in der Datei und k der Länge der Blöcke entspricht. Zusätzlich wird mit M die Größe eines Maschinenwortes benötigt.

3.2.3 Rsync-Algorithmus

Der Rsync-Algorithmus basiert auf „rollenden“ Prüfsummen und wurde in [33] vorgestellt.

[2] engl. rolling checksum

$$
\begin{aligned}
r_1(i,k) &= (\sum_{j=0}^{k-1} a_{j+i}) \mod M \\
r_2(i,k) &= (\sum_{j=0}^{k-1} (k-j) a_{j+i}) \mod M \\
r(i,k) &= r_1(i,k) + r_2(i,k)
\end{aligned}
$$

Abbildung 3.2: Formeln für die Berechnung der „rollenden“ Prüfsummen in `rsync`

1. Prüfsummen berechnen:

Zunächst unterteilt der erste Rechner die Datei in k Bytes große Blöcke. In [33] wird der Einfluß von k auf die Größe der Differenzen untersucht, hier wird darauf aber nicht genauer eingegangen. Für jeden Block wird eine „rollende“ Prüfsumme und eine MD4 [26]-Prüfsumme berechnet. Die zweite Prüfsumme wurde zur Sicherheit eingeführt, da die Wahrscheinlichkeit, daß die „rollende“ Prüfsummen für zwei Blöcke mit unterschiedlichem Inhalt denselben Wert hat, zu groß ist. In diesem Fall würden zwei Blöcke als gleich erkannt werden, obwohl sie unterschiedlichen Inhalt haben.

2. Prüfsummen übertragen:

Anschließend werden alle Prüfsummen, die im vorherigen Schritt berechnet wurden, an den zweiten Rechner gesendet.

3. Prüfsummen vergleichen:

Der zweite Rechner beginnt am Anfang der Datei und betrachtet immer einen Block von k Bytes, dieses Fenster wird im weiteren Verlauf immer um ein Byte in Richtung Ende der Datei verschoben. Für jeden Block wird die „rollende“ Prüfsumme berechnet und mit der empfangenen verglichen. Wenn eine Übereinstimmung gefunden wurde, wird für diesen Block eine MD4-Prüfsumme berechnet und ebenfalls mit der empfangenen verglichen. Wenn beide Prüfsummen für einen Block übereinstimmen, wird angenommen, daß dieser in beiden Dateien vorkommt.

4. Differenzen übertragen:

Danach sind dem zweiten Rechner die Blöcke bekannt, die in beiden Dateien vorkommen, und alle Bereiche in der Datei, die verändert wurden. Die Differenz beider Dateien besteht aus den Informationen über die Verschiebung der unveränderten Blöcke und dem Inhalt der veränderten Bereiche. Die Differenz wird an den ersten Rechner zurückgesendet, die dieser dann in die Originaldatei einarbeitet.

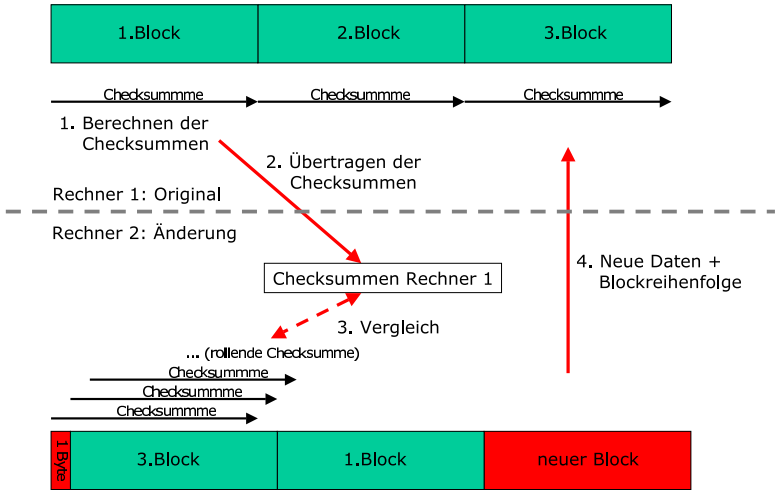

Abbildung 3.3: Rsync am Beispiel einer Veränderung an einer Datei

Trotz der Optimierung durch die Einführung der „rollenden“ Prüfsummen kann es in manchen Fällen effizienter sein, die komplette Datei zu übertragen, als die Differenzen zu berechnen. Dies gilt insbesondere, wenn die veränderte Datei stark von der Originaldatei abweicht und zwischen beiden Rechnern eine hohe Netzwerkbandbreite zur Verfügung steht. Im nächsten Abschnitt wird gezeigt, daß der Rsync-Algorithmus bei Netzwerkbandbreiten von 100 Mbits keine Vorteile gegenüber dem kompletten Übertragen der veränderten Datei hat.

3.2.4 Vergleichsmessungen

Um die Eigenschaften des Rsync-Algorithmus zu untersuchen, wurden im Rahmen dieser Arbeit Versuche mit verschiedenen Netzwerkbandbreiten und unterschiedlichen Übertragungsverfahren durchgeführt. Für die Versuche wurde in den Repositories der Quellcode des Linux-Kernels in verschiedenen Versionen gespeichert. In [33] wurde ebenfalls der Quellcode des Linux-Kernels für die Test-Daten verwendet. Es wurde aber nur untersucht, wieviel Daten zur Synchronisation übertragen werden müssen und nicht, wieviel Zeit für die Synchronisation benötigt wird.

Szenarien

Das `rsync`-Programm implementiert neben dem Rsync-Algorithmus auch das einfache Kopieren von Dateien über Netzwerkverbindungen. Optional kann die Kommunikation

Tabelle 3.1: Daten der verwendeten Linux-Kernel

Version	Größe in Bytes	Größe in Bytes (komprimiert)	Anzahl der Dateien
2.2.13	71838208	15079540	5243
2.2.14	75778560	15918652	5455
diff-2.2.13-2.2.14	7269094	1668147	–

zwischen den beiden Rechnern komprimiert werden. Alle Versuche wurden mit `rsync` durchgeführt. Für die Versuche wurden drei Szenarien bei drei unterschiedlichen Netzwerkbandbreiten untersucht, zusätzlich wurden die Messungen sowohl mit als auch ohne Kompression der Kommunikation durchgeführt.

- Im Szenario `COPY` befindet sich auf einem Rechner der Quellcode des Linux-Kernels in der Version 2.2.13. Auf dem anderen Rechner befinden sich keine Dateien. Die Dateien müssen von einem Rechner zu dem anderen kopiert werden.
- Im Szenario `IDENTICAL` befindet sich auf beiden Rechnern der Quellcode des Linux-Kernels in der Version 2.2.13 und die beiden Verzeichnisse werden mit Hilfe des Rsync-Algorithmus synchronisiert. In diesem Fall werden auf beiden Seiten die Prüfsummen berechnet und auch verglichen, aber es werden keine veränderten Blöcke ausgetauscht, da es keine Veränderungen gibt. Mit diesem Szenario soll der Verwaltungsaufwand des Rsync-Algorithmuses ermittelt werden.
- Im Szenario `PATCH` befindet sich auf einem Rechner der Quellcode in der Version 2.2.13 und auf dem anderen in der Version 2.2.14. Der offizielle Patch von der ersten Version auf die zweite ist komprimiert 1668147 Bytes und unkomprimiert 7269094 Bytes groß. Es werden auf beiden Seiten die Prüfsummen berechnet und verglichen. Zusätzlich zu den Prüfsummen müssen noch die veränderten Blöcke übertragen werden, deren Datenvolumen ungefähr der Größe der angegebenen Patches entspricht.

Für die Messungen wurden insgesamt 3 Rechner verwendet. Rechner A ist ein P4 1700 MHz, Rechner B ein PIII 800 MHz und Rechner C ein PII 400 MHz. Bei allen drei Rechnern ist der Hauptspeicher so groß, daß während der Testläufe keine Zugriffe auf die Festplatten nötig sind. Dadurch können Auswirkungen der Festplattenleistungen auf die Meßwerte verhindert werden. Rechner A und B sind über ein FastEthernet-Netzwerk verbunden. Beide Rechner können gleichzeitig maximal 100 MBits senden und empfangen. Rechner C ist mit einer asymmetrischen DSL-Leitung (Digital Subscriber Line) mit Rechner A verbunden und kann 864K Bits von diesem empfangen und 160 KBits an diesen senden.

Mit der asymmetrischen Netzwerkverbindung sollen symmetrische Verbindungen von 160 KBits und 864 KBits simuliert werden. Im `COPY` Szenario wurde das erreicht, indem

Tabelle 3.2: Laufzeiten bei verschiedenen Bandbreiten ohne Kompression

Szenario	160 Kbits	864 Kbits	100 Mbits
`COPY`	-	699.7s	13.6s
`IDENTICAL`	58.2s	58.0s	23.0s
`PATCH`	252.8s	111.1s	26.8s

Tabelle 3.3: Laufzeiten bei verschiedenen Bandbreiten mit Kompression

Szenario	160 Kbits	864 Kbits	100 Mbits
`COPY`	1021.9s	218.4s	20.6s
`IDENTICAL`	58.2s	50.4s	22.9s
`PATCH`	150.4s	73.8s	25.2s

zunächst auf Rechner A das Verzeichnis leer war und anschließend der Versuch mit dem leeren Verzeichnis auf Rechner C wiederholt wurde. Beim `IDENTICAL`-Szenario hat sich aber gezeigt, daß in beide Richtungen ungefähr die gleiche Menge an Daten versandt werden mußte. Die Messungen auf echten symmetrischen Verbindungen mit 864 KBits und 160 KBits werden deshalb bei `IDENTICAL` abweichen. Bei den anderen Szenarien sind die Datenflüsse asymmetrisch, deshalb sind die Ergebnisse mit den simulierten Verbindungen auf echte übertragbar.

Ergebnisse

Die Ergebnisse der durchgeführten Messungen sind in den Tabellen 3.2 und 3.3 dargestellt. Bei 100 MBits Netzwerkbandbreite zeigt es sich, daß Algorithmen, die die zu übertragene Datenmenge reduzieren sollen, keinen Vorteil sondern Nachteile bringen. Das schnellste Verfahren bei dieser Bandbreite ist das einfache Kopieren der Daten. Im `PATCH`-Szenario kann der komplette Quellcode schneller übertragen werden als die Differenzen mit Hilfe des Rsync-Algorithmus.

Bei den niedrigen Bandbreiten ist das Übertragen der Differenzen schneller als das einfache Kopieren der Dateien. Ohne Kompression ist der Rsync-Algorithmus in den durchgeführten Versuchen fast 7x schneller als das Kopieren. Von der Kompression profitieren alle Verfahren bei niedrigen Bandbreiten. Der Vorteil des Rsync-Algorithmus reduziert sich dabei auf einen Faktor von nur noch $3 - 6$.

Aus den Versuchen kann für diese Arbeit abgeleitet werden, daß die Auswahl der Übertragungsverfahren von der Netzwerkbandbreite zwischen den Repositories abhängig gemacht werden muß.

Kapitel 4

Synchronisation in vollständigen Graphen

In diesem Kapitel werden das Gossip[1]-Problem und seine Lösungen vorgestellt. Es geht auf folgende Frage zurück, die 1972 in [1] beschrieben wurde:

> There are n ladies, and each one of them knows an item of scandal which is not known to any of the others. They communicate by telephone, and whenever two ladies make a call, they pass on to each other, as much scandal as they know at the time. How many calls are needed before all ladies know all the scandal?

Da jede Dame eine Information an alle anderen sendet, ist die Gesamtheit aller Anrufe zu einem All-to-All Broadcast isomorph. Da bei der Synchronisation von Repositorysystemen die Veränderungen jedes Repository an alle anderen gesendet werden müssen, können die Lösungen des Gossip-Problems hierfür verwendet werden.

In der Literatur [17, 11] werden verschiedenen Arten von Anrufen diskutiert. Dabei wird zwischen full- und half-duplex Verbindungen unterschieden. Bei der half-duplex Variante können bei jeder Verbindung Informationen nur in eine Richtung gesendet werden, während bei full-duplex Daten in beide Richtungen ausgetauscht werden können. Diese beiden Varianten werden als `H1` und `F1` bezeichnet. `H1` bedeutet, daß alle Verbindungen half-duplex (`H`) sind und es für jede Nachricht genau einen (`1`) Empfänger gibt. In Kapitel 2 wurde begründet, daß das hier vorgestellte Synchronisationssystem zur Kommunikation das TCP-Protokoll verwenden soll. Im Weiteren werden deshalb nur Lösungen des Gossip-Problems für das `F1` Modell genauer vorgestellt.

4.1 Einleitung

Eine mathematische Formalisierung für das oben beschriebene Problem im `F1`-Modell ist folgende Definition:

[1] deutsch: Tratsch, Klatsch

Definition 4.1 (Gossiping F1) *Es sei $G = (E, V)$ ein Graph, wobei V Individuen mit Informationen sind ($|V| = n$), die niemand sonst besitzt, und $E \subset V \times V$ die möglichen Kommunikationsverbindungen zwischen den Individuen beschreibt.*

Gesucht wird eine Reihenfolge, in der die Individuen miteinander kommunizieren, die allen Teilnehmern alle Informationen bekannt macht.

Das Problem wird durch eine Sequenz S von Paaren $(i, j) \in E$ gelöst. Jedes Paar (i, j) wird als Verbindung zwischen i und j betrachtet, bei der beide die ihnen zu diesem Zeitpunkt bekannten Informationen dem anderen mitteilen. Zur Verdeutlichung der Bedeutung der Paare werden sie im folgenden leicht verändert dargestellt: $i \leftrightarrow j$. Nachdem S abgearbeitet wurde, müssen jedem Individuum alle Informationen bekannt sein.

Das Abarbeiten des Sequenz wird Gossiping genannt.

In dieser Definition werden die Möglichkeiten zur Kommunikation eingeschränkt, indem nur entlang der Kanten des Graphen G kommuniziert werden darf. In der oben vorgestellten Aufgabe [1] waren Verbindungen zwischen allen Teilnehmern erlaubt, in dem Fall wäre G ein vollständiger Graph.

4.2 Beispiel

Am Beispiel eines Graphen mit vier Knoten wird nun gezeigt, wie Gossiping funktioniert.

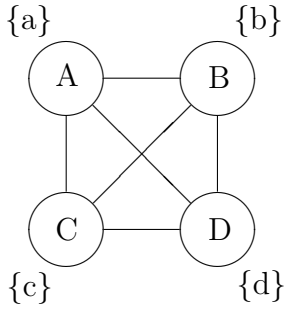

Abbildung 4.1: 4 Knoten mit 4 Informationen im vollständigen Graphen

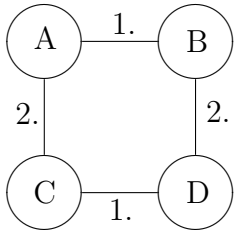

Abbildung 4.2: Darstellung der Lösung des Gossip-Problems

In Abbildung 4.1 wird ein vollständiger Graph mit vier Knoten dargestellt, für den das Gossip-Problem gelöst werden soll. Jeder Knoten besitzt eine nur ihm bekannte Information, die durch einen Kleinbuchstaben dargestellt wird. Eine Lösungssequenz für dieses Beispiel lautet $S = [A \leftrightarrow B, C \leftrightarrow D, A \leftrightarrow C, B \leftrightarrow D]$. In Abbildung 4.2 sind nur die Kanten der Lösung eingezeichnet.

Die Kanten werden zu möglichst großen Schritten bzw. möglichst wenigen Kommunikationsrunden zusammengefaßt. In jedem Schritt ist jeder Knoten an maximal einer Kommunikationsverbindung beteiligt. Dadurch können alle Kommunikationsverbindungen eines Schritts parallel erfolgen, ohne daß ein Knoten an zwei Verbindungen gleichzeitig beteiligt ist. Die Anzahl der Schritte entspricht der Länge des kritischen Pfads.

Die Schritte werden in den Lösungssequenzen über die Mengennotation kenntlich gemacht. Die Lösung S kann in zwei Schritte unterteilt werden. $S' = [\{A \leftrightarrow B, C \leftrightarrow D\}, \{A \leftrightarrow C, B \leftrightarrow D\}]$. In der Abbildung ist zusätzlich jede Kante mit dem Schritt, in dem sie aktiv ist, annotiert.

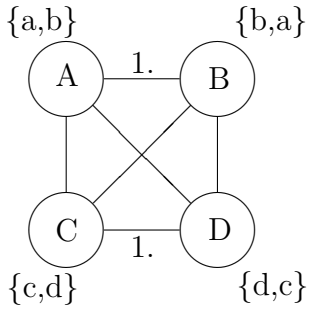

Abbildung 4.3: Konfiguration nach dem ersten Schritt

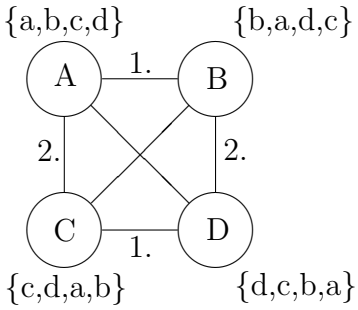

Abbildung 4.4: Konfiguration nach dem zweiten Schritt

Abbildung 4.1 stellt die Anfangskonfiguration des Beispiels dar. In den Abbildungen 4.3 und 4.4 wird die Verteilung der Informationen nach dem ersten bzw. zweiten Schritt gezeigt. Im ersten Schritt tauschen A & B und C & D ihre Informationen aus und im zweiten A & C und B & D. Neben jedem Knoten sind die ihm bekannten Informationen nach dem jeweiligen Schritt angegeben. Nach dem zweiten Schritt haben alle Knoten alle Informationen erhalten.

Bei diesem Beispiel ist zu beachten, daß A und D nicht direkt miteinander kommunizieren. Die Informationen zwischen den beiden Knoten werden indirekt über B und C ausgetauscht. Indem die Transitivität der Synchronisation ausgenutzt wird, muß also nicht zwischen allen Knoten direkt kommuniziert werden.

4.3 Erzeugung von Lösungen für das Gossip-Problem

Beim Gossip-Problem werden Lösungssequenzen zum Gossiping gesucht, bei denen die Anzahl der Kommunikationsrunden minimal ist. Für vollständige Graphen mit n Knoten wurde in [15] bewiesen, daß die minimale Anzahl der Schritte:

- $\lceil 2n \rceil$ für n gerade und
- $\lceil 2n \rceil + 1$ für n ungerade beträgt.

Das Finden von Lösungen für das Gossip-Problem mit minimaler Anzahl an Kommunikationsrunden im `F1` Modell ist NP-vollständig. Ein Beweis dafür wird in [17] beschrieben. Für bestimmte Klassen von Graphen G läßt sich das Problem jedoch mit polynomiellem Aufwand lösen. In [11] werden optimale Lösungen unter anderem für vollständige Graphen, Hypercubes, Ketten und Ringe beschrieben.

Im nächsten Abschnitt werden Algorithmen aus [11] vorgestellt, die für das Gossip-Problem Lösungen mit minimaler Anzahl an Schritten erzeugen, wenn zwischen allen

Knoten Kommunikationsverbindungen erlaubt sind. Dabei wird unterschieden, ob die Anzahl der Knoten n gerade oder ungerade ist. Der erste Algorithmus löst das Problem für n gerade.

4.3.1 Lösungen für vollständige Graphen mit geradem n

Algorithmus 4.1 (Konstruktion von Lösungen des Gossip-Problems für gerade n und vollständige Graphen G): *Es sei $n = 2m$. Die Knoten werden in 2 Mengen Q, R von je m Knoten aufgeteilt. Die Elemente von Q und R werden auf folgende Weise indiziert: $Q[i], R[i], 0 \leq i \leq m-1$.*

for all $i \in \{0, \ldots, m-1\}$ do in parallel
exchange information between $Q[i]$ and $R[i]$;
for $t = 1$ to $\lceil 2m \rceil$ do
for all $i \in \{0, \ldots, m-1\}$ do in parallel
exchange information between $Q[i]$ and $R[(i + 2^t - 1) mod\, m]$;

Die Schritte werden durch die parallelen for-Schleifen ausgedrückt. Also führt dieser Algorithmus $1 + \lceil 2m \rceil = \lceil 2(2m) \rceil = \lceil 2n \rceil$ Schritte aus. Damit wurde eine Lösung erzeugt, die die minimale Anzahl an Schritten benötigt.

In Abbildung 4.5 wird das Verfahren für $n = 8$ graphisch dargestellt. Die Knoten werden mit `A` bis `H` bezeichnet. Die Knoten `A - D` bilden die Menge Q und die Knoten `E - H` die Menge R. In der Abbildung sind die Informationen, die am Anfang den Knoten `A` und `D` bekannt sind, durch kleine Kreise kenntlich gemacht. Die Pfeile stellen die Kommunikationsverbindungen dar. An jedem Pfeil sind die Informationen eingezeichnet, die über die entsprechende Verbindung übertragen werden.

Die Abbildung 4.5 veranschaulicht, daß im ersten Schritt die Knoten der Menge Q mit ihrem direkten Gegenüber kommunizieren. Danach wird zu dem Gegenüber ein Offset hinzuaddiert, der sich aus der Folge $\{1, 3, 7, 15, \ldots, 2^t - 1\}$ ergibt. Bei geeigneter graphischer Darstellung kann leicht gesehen werden, daß die Knoten und Verbindungen einen Hypercube bilden.

4.3.2 Lösungen für vollständige Graphen für ungerade n

Algorithmus 4.2 Konstruktion von Lösungen des Gossip-Problems für ungerade $n (n = 2m + 1)$ und vollständige Graphen:
Die Knoten werden mit $node_1, \ldots, node_n$ bezeichnet.

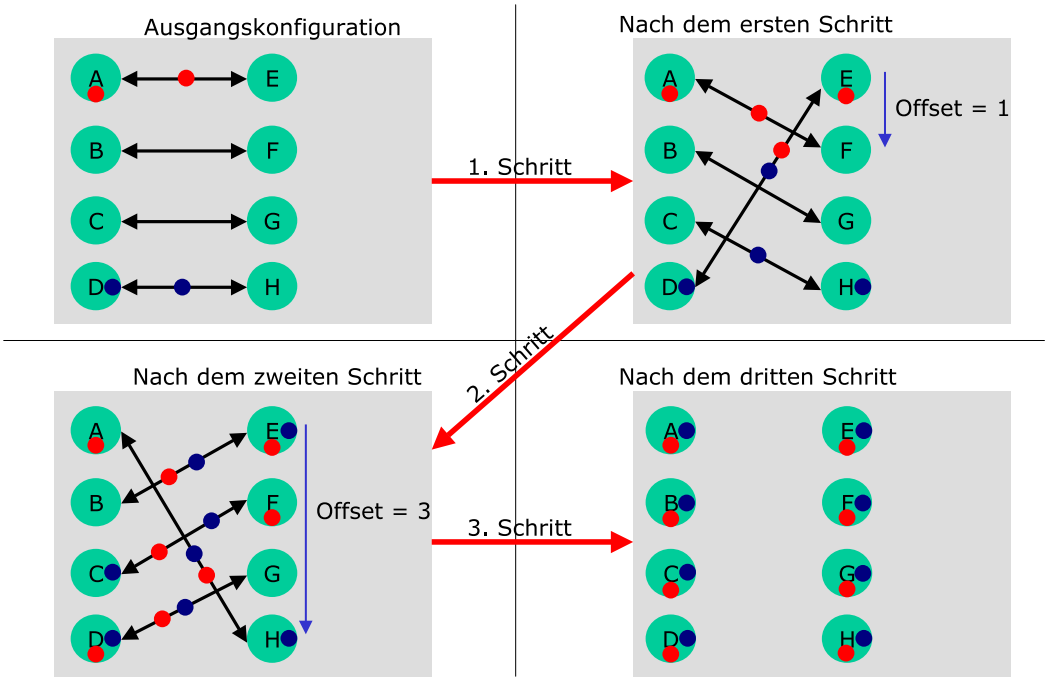

Abbildung 4.5: Die Lösung des Algorithmus für das Gossip-Problem mit 8 Knoten

for all $i \in \{2, \ldots, m+1\}$ *do in parallel*
    exchange information between $node_i$ *and* $node_{i+m}$;
if $m+1$ *even do*
    gossip in $\{node_1, node_2, \ldots, node_{m+1}\}$;
else
    gossip in $\{node_1, node_2, \ldots, node_{m+2}\}$;
fi
for $i \in \{2, \ldots, m+1\}$ *do in parallel*
    exchange information between $node_i$ *and* $node_{i+m}$;

Um die Anzahl der Schritte zu berechnen, die der Algorithmus 4.2 benötigt, muß eine Fallunterscheidung gemacht werden. Der erste Fall behandelt alle n für die $m+1$ gerade ist. Dann wird das Problem in $\lceil 2(m+1)\rceil + 2 = \lceil 2\frac{n+1}{2}\rceil + 2 = \lceil 2(n+1)\rceil + 1 = \lceil 2n\rceil + 1$ Schritten gelöst. Im zweiten Fall ist $m+1$ ungerade. Dann werden $\lceil 2(m+2)\rceil + 2 = \lceil 2\frac{n+3}{2}\rceil + 2 = \lceil 2(n+3)\rceil + 1 = \lceil 2n\rceil + 1$ Schritte benötigt [11].

Die Knoten werden in zwei Gruppe aufgeteilt, wobei eine Gruppe um einen Knoten größer als die andere ist. Jeder Knoten der kleineren Gruppe tauscht mit seinem Gegenüber der größeren Gruppe Informationen aus. Im zweiten Schritt wird auf den Algorithmus 4.1 zurückgegriffen. Durch die Fallunterscheidung wird sichergestellt, daß die Anzahl der Knoten, auf die der Algorithmus 4.1 angewendet wird, gerade ist. Anschließend wird der erste Schritt wiederholt.

4.3.3 Lösungen für Ketten

Eine Kette ist ein zusammenhängender Graph mit n Knoten und $n-1$ Kanten. Zur Lösung des Gossip-Problems werden in dieser Topologie

- $n-1$ Schritte für n gerade und
- n Schritte für n ungerade benötigt.

Die Knoten einer Kette werden mit $\{v_0, v_1, \ldots, v_{n-1}\}$ bezeichnet und die Kanten mit $\{e_0, e_1, \ldots, e_{n-2}\}$, wobei die Kante e_i die Knoten v_i und v_{i+1} verbindet.

Algorithmus 4.3 Konstruktion von Lösungen des Gossip-Problems für Ketten mit n Knoten:

```
even_edges = {e_0, e_2, e_4, ...}
odd_edges = {e_1, e_3, e_5, ...}
if even(n) then
    count_steps = n - 1
else
    count_steps = n
fi
for i = 0 to count_steps-1 do
    if even(i) then
        for all e ∈ even_edges do in parallel
            exchange information via edge e
    else
        for all e ∈ odd_edges do in parallel
            exchange information via edge e
    fi
```

4.3.4 Lösungen für Ringe

Ein Ring ist eine Kette, bei der der erste Knoten zusätzlich mit dem letzten verbunden ist. Ein Ring mit n Knoten hat n Kanten. Lösungen für Gossip-Problem in dieser Topologie benötigen

- $\frac{n}{2}$ Schritte für n gerade und
- $\lceil \frac{n}{2} \rceil + 1$ Schritte für n ungerade.

Die Knoten und Kanten eines Ringes werden entsprechend denen einer Kette beschrieben, zusätzlich verbindet die Kante e_{n-1} die Knoten v_{n-1} und v_0. Der Algorithmus zum Erzeugen der Lösungen ist bis auf die Berechnung von *count_steps* mit dem für Ketten identisch. Die Berechnung von *count_steps* muß entsprechend angepaßt werden.

Kapitel 5

Synchronisation unter Berücksichtigung der Topologie

Lösungen des Gossip-Problems mit minimaler Anzahl an Kommunikationsrunden wurden in Kapitel 4 für Ketten, Ringe und vollständig verbundene Netzwerke vorgestellt. In dieser Arbeit soll aber ein Synchronisationssystem vorgestellt werden, das die Laufzeit minimiert. Deshalb werden in diesem Kapitel verschiedene Methoden vorgestellt, die auf Basis der Lösungen aus Kapitel 4 die Laufzeit optimieren.

Im ersten Abschnitt wird ein Modell vorgestellt, mit dem sich die Effizienz der vorgestellten Methoden vergleichen läßt. Die Methoden zur Laufzeitoptimierung werden im zweiten Abschnitt vorgestellt und miteinander verglichen. Dabei werden die Begriffe Rechner und Repository synonym verwendet. In der Diskussion wird davon ausgegangen, daß jedes Repository auf einem anderen Rechner gespeichert ist. Die vorgestellten Verfahren funktionieren aber auch, wenn mehrere Repositories auf einem Rechner gespeichert sind. Im letzten Abschnitt wird eine XML [4]-basierte Sprache vorgestellt, mit der Netzwerktopologien beschrieben werden können. Sie dient in der Implementation des im Rahmen dieser Arbeit entstandenen Synchronisationssystems dazu, die Topologie des Netzwerks zwischen den Rechnern, auf denen die Repositories gespeichert sind, zu beschreiben.

5.1 Modell zur Laufzeitabschätzung

Das im folgenden vorgestellte Modell soll die Möglichkeit bieten, verschiedene Kommunikationsstrategien zur Synchronisation miteinander zu vergleichen. Dazu wird mit Hilfe des Modells die Laufzeit verschiedener Strategien für konkrete Szenarien abgeschätzt. Das Modell hat nicht das Ziel, die Laufzeiten exakt vorherzusagen. Es soll nur dazu dienen, verschiedene Strategien zu vergleichen und gute von schlechten Lösungen zu unterscheiden.

Um für eine Lösungssequenz die Laufzeit abzuschätzen, wird sie wie in Kapitel 4 in einzelne Kommunikationsrunden unterteilt. Für jede Runde wird die Laufzeit berechnet und anschließend die Summe über die Laufzeiten aller Schritte gebildet. Dabei

wird davon ausgegangen, daß sich die Ausführungszeiten der Schritte nicht überschneiden. In der in Kapitel 6 vorgestellten Implementierung wurde diese Einschränkung aus Leistungsgründen aufgehoben. Dort führt jeder Knoten die Kanten in der durch die Lösungssequenz gegebenen Reihenfolge aus, dabei können sich einzelne Schritte überschneiden.

Sowohl die Topologie zwischen den Repositories als auch die zu simulierende Kommunikationsstrategie wird in Form von Funktionen angegeben. Die Funktion $V_{send}(i, step) : \text{Node} \times \mathbb{N} \to \mathbb{N}$ gibt das Datenvolumen in Bytes an, das der Knoten i im Schritt $step$ versendet. $edges(step) : \mathbb{N} \to P(\text{Node} \times \text{Node})$ liefert die Menge der Kanten, über die im Schritt $step$ kommuniziert wird. Mit der Funktion $bandwidth(i, j) : \text{Node} \times \text{Node} \to \mathbb{N}$ wird die Bandbreite der Netzwerkverbindung von Knoten i zu Knoten j in $\frac{Byte}{s}$ beschrieben.

$$t = \sum_{i=0}^{steps-1} MAX_{(j,k)\in edges(i)} MAX\left(\frac{V_{send}(j,i)}{bandwith(j,k)}, \frac{V_{send}(k,i)}{bandwidth(k,j)}\right)$$

Für jede Kante eines Schritts wird die Dauer berechnet, für die Daten über sie übertragen werden. Das Maximum über diese Zeiten ist die Laufzeit des jeweiligen Schritts. Die Gesamtlaufzeit ergibt sich aus der Summe über die Laufzeiten der Schritte.

Bei der Synchronisation werden hauptsächlich kontinuierliche Datenströme versendet, die vom Empfänger nicht beantwortet werden müssen, wodurch die Latenzzeiten des Netzwerkes nur eine untergeordnete Rolle spielen und für die Laufzeitabschätzungen nicht berücksichtigt werden müssen.

5.2 Anpassungen an Hierarchien und Gitter

Beim in Kapitel 4 vorgestellten Gossip-Problem werden den Verbindungen konstante Kosten zugeordnet. Bei der Synchronisation hängen die Kosten einer Verbindung aber von der zur Verfügung stehenden Bandbreite und dem zu übertragenden Datenvolumen ab. Für flache Netzwerktopologien wie Ketten, Ringe und Switche, in denen alle Netzwerkverbindungen dieselbe Bandbreite haben, können die vorgestellten Lösungen verwendet werden. Um aber hierarchische Netzwerke effizient synchronisieren zu können, müssen andere Verfahren entwickelt werden.

In [14] werden Implementierungen von kollektiven Operationen für die Message-Passing-Bibliothek MPI [18] in hierarchischen Netzwerken vorgestellt. Unter anderem wird eine Implementierung für die `allgather`-Funktion vorgestellt. Diese Funktion ist mit dem hier untersuchten All-to-All-Broadcast vergleichbar. Es wurden aber nur Hierarchien mit zwei Ebenen untersucht. Für diese Arbeit können die dort vorgestellten Methoden nicht direkt übernommen werden.

In Kapitel 4 wurden für das Gossip-Problem die erlaubten Netzwerkverbindungen zwischen den Teilnehmern mit einem Graphen beschrieben. Im folgenden wird angenommen, daß sich zwischen den Rechnern, auf denen die Repositories gespeichert sind, eine

hierarchische Netzwerktopologie befindet. Alle drei im folgenden vorgestellten Algorithmen zerlegen die Topologien in Subtopologien und lösen für diese das Gossip-Problem, dazu wird auf die in Kapitel 4 vorgestellten Verfahren zurückgegriffen. Da das Gossip-Problem NP-vollständig ist und die vorgestellten Algorithmen polynomiellen Aufwand haben, werden die erzeugten Lösungen nicht immer eine optimale Laufzeit haben. Deshalb werden die Algorithmen auch Approximationen genannt.

In diesem Abschnitt werden drei Beispiele für Synchronisationen vorgestellt. Aus den Beispielen wird jeweils eine Approximation abgeleitet.

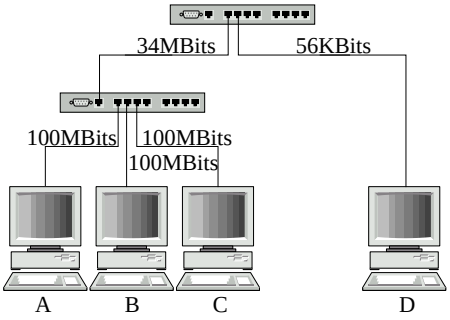

Abbildung 5.1: Beispieltopologie, in der vier Rechner untereinander mit 100 MBits bzw. 56 KBits verbunden sind.

Im ersten Beispiel soll die Topologie in Abbildung 5.1 betrachtet werden, in der die Rechner `A` bis `C` über einen Switch mit 100 MBits verbunden sind, während Rechner `D` über ein Modem mit 56 KBits mit dem restlichen Netzwerk verbunden ist. In diesem Beispiel gibt es 4 Repositories, also könnte die Lösung für den 4er-Graphen aus Kapitel 4 verwendet werden ($S = [A \leftrightarrow B, C \leftrightarrow D, A \leftrightarrow C, B \leftrightarrow D]$). Wenn auf Rechner D die Veränderungen insgesamt 30 MB umfassen, würden diese zweimal über die Modem-Verbindung übertragen werden.

Zunächst wird mit dem Modell die Laufzeit für die Synchronisation dieses Beispiels abgeschätzt, dafür müssen die oben beschriebenen Funktionen definiert werden. Für V_{send} ergeben sich folgende Werte:

$$V_{send}(D, 0) = 30\ MB V_{send}(D, 1) = 30\ MB V_{send}(C, 1) = 30\ MB$$

Für nicht angegebene Argumente soll die Funktion den Wert 0 annehmen. Die Bandbreite beträgt 56 KBits für Verbindungen, an denen D beteiligt ist, und für alle anderen 100 MBits.

$$\begin{aligned} t &= \frac{V_{send}(D,0)}{bandwith(D,C)} + MAX(\frac{V_{send}(D,1)}{bandwidth(D,B)}, \frac{V_{send}(C,1)}{bandwidth(C,A)}) \\ &= \frac{30MB}{56KBit/s} + MAX(\frac{30MB}{100MBit/s}, \frac{30MB}{56KBit/s}) \\ &= 2 * 3840s = 2.1h \end{aligned}$$

Das Modell schätzt die Laufzeit der Synchronisation auf 2.1 Stunden.

5.2.1 Einfache Approximation

Für die Approximationen werden die Topologien als Hierarchie von Netzwerkbausteinen betrachtet. Eine Hierarchie ist entweder ein Rechner oder ein Strukturelement (Switch, Ring oder Kette), an das weitere Hierarchien angeschlossen sind.

Im obigen Beispiel ist das Wurzelelement ein Switch, an den der Rechner `D` mit 56 KBits und der Switch mit den Rechnern `A` bis `C` mit 100 MBits angeschlossen sind. Bei diesem Beispiel muß es das Ziel sein, wenig Daten über die langsame Modem-Verbindung zu übertragen. Dafür bietet sich folgende Approximation an:

Algorithmus 5.1 *Einfache Approximation:*
synchronize*(Hierarchie):*
forall $x \in$ *childs(Hierarchie) do-in-parallel* **synchronize***(x)*
$nodes = \bigcup_{x \in childs(Hierarchie)} proxy(x)$
gossip*(nodes)*
forall $x \in$ *childs(Hierarchie) do-in-parallel* **synchronize***(x)*

Diese Approximation ist eine Verallgemeinerung der in [14] vorgestellten Implementierung für die Funktion `allgather`. Die Approximation arbeitet wie folgt:

1. Für alle Kinder des Wurzelements wird die Funktion **synchronize** rekursiv aufgerufen.

2. Für jedes Kind x des Wurzelelementes wird ein Stellvertreter gewählt. Der Stellvertreter ist ein Rechner aus der Hierarchie, die x als Wurzelelement hat. Die Stellvertreter werden miteinander synchronisiert, wobei auf die Lösungen des Gossip-Problems zurückgegriffen wird.

3. Wieder wird für alle Kinder des Wurzelements die Funktion **synchronize** rekursiv aufgerufen.

Das Ergebnis der Approximation ist dann eine Komposition von Lösungen des Gossip-Problems. Auf dieses Beispiel angewendet bedeutet dies, daß sich die Rechner `A` bis `C` synchronisieren ($[A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A]$). Das zweite Kind des Wurzelelementes besteht nur aus dem Rechner `D`, deshalb muß diese Gruppe nicht synchronisiert werden ($[]$). Als Stellvertreter der beiden Gruppen werden die Rechner `A` und `D` gewählt. Diese synchronisieren sich miteinander ($[A \leftrightarrow D]$). Schließlich werden noch einmal die zuerst genannten Rechnergruppen synchronisiert ($[A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A]$). Da in diesem Beispiel nur Switche und Knoten auftreten, werden beim Aufruf der Funktion `gossip` die Lösungen für vollständige Graphen verwendet. Für Ringe, Ketten und Busse würden entsprechende Lösungen verwendet werden.

Wenn diese Approximation auf das Beispiel aus Abbildung 5.1 angewendet wird, ergibt sich diese Lösungssequenz: $S = [A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A, A \leftrightarrow D, A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A]$. Für diese Lösung wird nun die Laufzeit der Synchronisation ermittelt. Es

gilt wieder, daß alle Funktionen 0 als Ergebnis haben, wenn nichts anderes angegeben wird.

$$V_{send}(D,3) = 30MB; V_{send}(A,4) = 30MB; V_{send}(B,5) = 30MB$$

Die Funktion *bandwidth* kann von der ersten Laufzeitabschätzung übernommen werden. Die Berechnung wurde wieder vereinfacht, indem Terme mit dem Wert 0 weggelassen wurden.

$$t = 1 \cdot \frac{30MB}{56KBit/s} + 2 \cdot \frac{30MB}{100MBit/s} = 3840s + 4.8s = 1.07h$$

Durch diese Anpassung konnte die Laufzeit von $2.1h$ auf $1.07h$ annähernd halbiert werden, während die Anzahl der Kommunikationsrunden von zwei auf sieben gestiegen ist. In diesem Beispiel hat die Anpassung gute Ergebnisse geliefert, die vermutlich nicht weiter verbessert werden können. Im nächsten Beispiel reicht diese Anpassung nicht aus, um gute Laufzeiten zu erzielen.

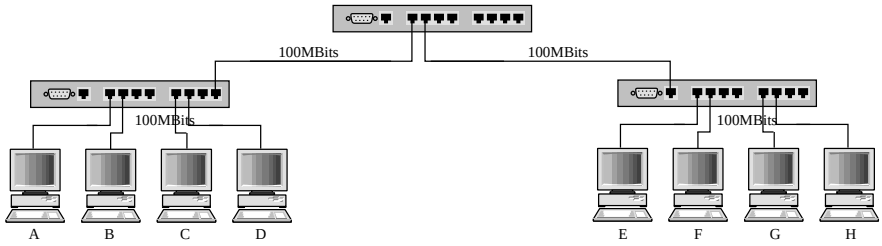

Abbildung 5.2: 2 Gruppen von je 4 Rechnern sind mit einer 100 MBits-Leitung verbunden

Für das nächste Beispiel wird die Topologie aus Abbildung 5.2 betrachtet. In einer dreistufige Hierarchie sind acht Rechner verbunden. Die drei Netzwerkkomponenten stellen jeweils einen Switch dar. Alle Netzwerkverbindungen haben eine Bandbreite von 100 MBits. Auf Rechner A seien Änderungen mit einem Volumen von 30 MB durchgeführt worden. Die Repositories auf den anderen Rechnern seien unverändert.

Wenn die erste Approximation für diese Topologie angewendet wird, ergibt sich mit dem in Abschnitt 5.1 vorgestellten Modell eine Laufzeit von 12s, dabei wurde als Lösungssequenz $[A \leftrightarrow B, \{A \leftrightarrow C, B \leftrightarrow D\}, A \leftrightarrow E, E \leftrightarrow F, \{E \leftrightarrow G, F \leftrightarrow H\}]$[1] verwendet. Durch Berücksichtigung der Verteilung der Veränderungen an den Repositories über die Rechner kann eine bessere Lösung konstruiert werden.

5.2.2 An die Verteilung der Veränderungen angepaßte Approximation

Um die Laufzeit zu reduzieren, müssen die Veränderungen früh in die zweite Gruppe von Rechnern übertragen werden. Dadurch können die Veränderungen in beiden Gruppen

[1] Es wurden nur Verbindungen angegeben, über die Daten übertragen wurden.

parallel verteilt werden. In diesem Beispiel muß zunächst A mit E synchronisiert werden. Anschließend können die Veränderungen in beiden Gruppen gleichzeitig verteilt werden. Dadurch ergibt sich die Lösungssequenz $[A \leftrightarrow E, \{A \leftrightarrow B, E \leftrightarrow F\}, \{A \leftrightarrow C, B \leftrightarrow D, E \leftrightarrow G, F \leftrightarrow H\}]$ und eine Laufzeit von 7.2s, während die erste Approximation 12s gebraucht hätte.

Algorithmus 5.2 *An die Verteilung angepaßte Approximation:*
synchronize*(Hierarchie):*
    forall x in childs(Hierarchie) do-in-parallel
        synchronize_activeNodes*(x)*
    nodes $= \bigcup_{x \in childs(Hierarchie)}$ *active_proxy(x)*[2]
    gossip*(nodes)*
    forall x in childs(Hierarchie) do-in-parallel
        synchronize*(x)*

Der Unterschied zu der ersten Approximation besteht darin, daß zunächst nur die Repositories miteinander synchronisiert werden, die verändert wurden. Dadurch muß dann auch die Wahl der Stellvertreter angepaßt werden. Stellvertreter einer Hierarchie, in der Veränderungen vorgekommen sind, müssen selber eines der Repositories sein, das verändert wurde.

5.2.3 An die Verteilung der Veränderungen angepaßte parallelisierte Approximation

Wenn die Parameter im letzten Beispiel ein wenig geändert werden, indem auf Rechner `E` eine zusätzliche Veränderung von 30MB eingefügt wird, benötigen beide vorgestellte Approximationen 12s zur Synchronisation. Mit der dritten nun vorgestellten Approximation wird es möglich sein, das Repositorysystem in 9.6s zu synchronisieren. Dazu wurde folgende Lösungssequenz verwendet: $[\{(A, B), (E, F)\}, \{(A, E), (B, C), (F, G)\}, \{(E, F), (A, B), (C, D), (G, H)\}, \{(A, C), (B, D), (E, G), (F, H)\}]$.

Die Sequenz wurde mit dieser Approximation erzeugt:

Algorithmus 5.3 *An die Verteilung angepaßte parallelisierte Approximation:*
synchronize*(Hierarchie):*
    forall x in childs(Hierarchie) do-in-parallel
        synchronize_activeNodes2(x)
    nodes $= \bigcup_{x \in childs(Hierarchie)}$ *active_proxy(x)*
    do in parallel{
        forall x in childs(Hierarchie) do-in-parallel{
            x' = excludeNodes(x, nodes)
            synchronize(x')
        }

[2] Mit dem Zusatz **_activeNodes** bzw. **active_** sind Knoten gemeint, auf denen Veränderungen vorhanden sind.

```
}and{
        gossip(nodes)
}
forall x in childs(Hierarchie) do-in-parallel
        synchronize(x)
```

1. Für alle Kinder des Wurzelements wird die Funktion **synchronize_activeNodes2** rekursiv aufgerufen. Dabei werden wie in der zweiten Approximation zunächst nur die Repositories mit Veränderungen synchronisiert. Zusätzlich wird versucht, daß es in jeder Hierarchie zwei Rechner gibt, die Stellvertreter für diese Hierarchie werden können.

2. Synchronisation zwischen den Stellvertretern der Kinder des Wurzelelementes, parallel dazu Synchronisation der Hierarchien, die durch die Kinder des Wurzelelementes gebildet werden, ohne die Stellvertreter.

3. Synchronisation in den Gruppen.

Bei der Lösung, die die zweite Approximation für dieses Beispiel geliefert hat, müssen im dritten Schritt sehr viele Daten synchronisiert werden, während im zweiten Schritt sehr wenige Verbindungen parallel arbeiten. Der zweite Schritt wird vermutlich einen hohen Anteil an der Laufzeit haben, da die Verbindungen zwischen den Gruppen tendenziell eher eine geringere Bandbreite besitzen als die Verbindungen innerhalb einer Gruppe, deshalb wird für die dritte Approximation ein Teil des dritten Schritts parallel zum zweiten Schritt ausgeführt. Die Daten, die in der Gruppe bekannt sind, können parallel zur Kommunikation zwischen den Gruppen verteilt werden. Dazu sind in jeder Gruppe zwei Knoten nötig, die alle Informationen dieser Gruppe kennen. Der eine Knoten übernimmt die Stellvertreterrolle und wird für die Kommunikation zwischen den Gruppen genutzt, während der andere die Daten innerhalb der Gruppe verteilt. Dadurch müssen im letzten Schritt nur die neu empfangenen Daten aus den anderen Gruppen verteilt werden. Diese Approximation erzielt gute Ergebnisse, wenn in mehreren Gruppen Veränderungen vorhanden sind, aber es im Verhältnis mehr Repositories ohne als mit Veränderungen gibt.

5.2.4 Ergebnis

In der Implementation (siehe Kapitel 6) wurden nur die letzten beiden Approximationen implementiert und mit Hilfe des Modells wird zur Laufzeit die effizientere der beiden Approximationen ausgewählt. Der Aufwand für das Erzeugen der Lösungen liegt in $O(n^2)$. Da das allgemeine Gossip-Problem NP-vollständig ist und die hier vorgestellten Lösungen polynomiellen Aufwand besitzen, sind die Methoden nicht immer optimal. Aber die Approximationen können nicht aufwendiger gestaltet werden, da dann die Laufzeit für große Systeme die Berechnung der Lösungen dominieren würde.

5.2.5 Anpassung an mehrdimensionale Gitter

Für die Synchronisation von Repositorysystemen, die in einem mehrdimensionalen Gitter angeordnet sind, wurde ein flexibler Ansatz gewählt, so daß Topologien wie Hypercubes und Tori [6] nicht gesondert betrachtet werden müssen, sondern Spezialfälle für mehrdimensionale Gitter sind.

Das Besondere an diesem Ansatz ist die Möglichkeit, für jede Dimension des Gitters eine andere Topologie des Verbindungsnetzwerk zu wählen. Wenn für alle Dimensionen Ringe gewählt werden, entsteht ein Torus. Ein Hypercube entsteht, wenn für Dimensionen Ketten gewählt und in jeder Dimension nur zwei Koordinaten verwendet werden.

Da für diese Arten von Topologien nur eine effiziente Methode gefunden wurde, können keine Vergleiche zwischen unterschiedlichen Ansätzen durchgeführt werden.

Algorithmus 5.4 *Kommunikationsstrategien für mehrdimensionale Gitter:*
Gegeben sei ein d dimensionales Gitter. Dann ist $n = \prod_{i=0}^{d-1} d_i$ *die Anzahl der Rechner in dem Gitter und die i-te Dimension hat die Größe* d_i*. Das Gitter wird synchronisiert, indem nacheinander die Dimensionen durchlaufen werden und für jede Dimension die Zeilen, die parallel zu dieser Dimension liegen, synchronisiert werden.*

Für den 2-dimensionalen Fall ergibt sich dann folgende Lösung:

1. *Synchronisiere jede Zeile.*
2. *Synchronisiere jede Spalte.*

Formale Beschreibung der Konstruktion der Lösungssequenz:

> Gegeben seien d Funktionen $row_i(j) : \mathbb{N} \to P(V)$ mit $0 \leq i < d$ und $0 \leq j < \frac{\prod_{k=0}^{d-1} d_k}{d_i}$. $row_i(j)$ liefert die Knoten der Zeile j, die parallel zur Dimension i verläuft.
>
> Dann gilt für die Lösungssequenz:
>
> $$S = \bigcup_{i \in [0 \ldots d-1]} \bigcup_{j \in [0 \ldots \frac{\prod_{k=0}^{d-1} d_k}{d_i} - 1]} gossip_i(row_i(j))$$

Wenn die Dimension i aus Ketten besteht, löst die Funktion $gossip_i$ das Gossip-Problem für Ketten. Für andere Dimensionen und Verbindungsnetzwerke werden analog die Funktionen angepaßt.

Die Anzahl der Schritte, die bei dieser Komposition ausgeführt werden müssen, ergibt sich, indem über alle Dimensionen die Schritte, die für diese Dimension benötigt werden, aufsummiert werden.

5.3 Topologiebeschreibung in XML

Um ein Repositorysystem effizient synchronisieren zu können, muß die Topologie des Netzwerkes zwischen den Repositories berücksichtigt werden. Diese Topologie muß dem Synchronisationssystem vom Benutzer mitgeteilt werden.

Mit den bisher besprochenen Mitteln läßt sich die Synchronisation an Switche, Ketten, Ringe, mehrdimensionale Gitter und Hierarchien davon anpassen. Da sich Hierarchien als Baum darstellen lassen, bietet es sich für die Beschreibung der Topologien an, eine XML [4]-basierte Sprache zu verwenden. Im folgenden wird eine im Rahmen dieser Arbeit entwickelte Sprache vorgestellt, mit der sich die oben genannten Topologien beschreiben lassen. XML-basierte Sprache werden in Form einer DTD [4][3] beschrieben, dabei handelt es sich um eine Definition der Grammatik für diese Sprache. Für die hier vorgestellte Sprache ist die DTD in Anhang A angegeben.

5.3.1 Hierarchien in XML

Traditionell ist die Wurzel einer Hierarchie in graphischen Darstellungen oben. In Abbildung 5.3 wird gezeigt, wie die traditionelle Darstellung in eine XML-Repräsentation überführt werden kann. Dazu muß die graphische Darstellung nur um 90**?** gegen den Uhrzeigersinn gedreht werden. In der Abbildung ist die Beziehung von XML und bisheriger Darstellung kenntlich gemacht.

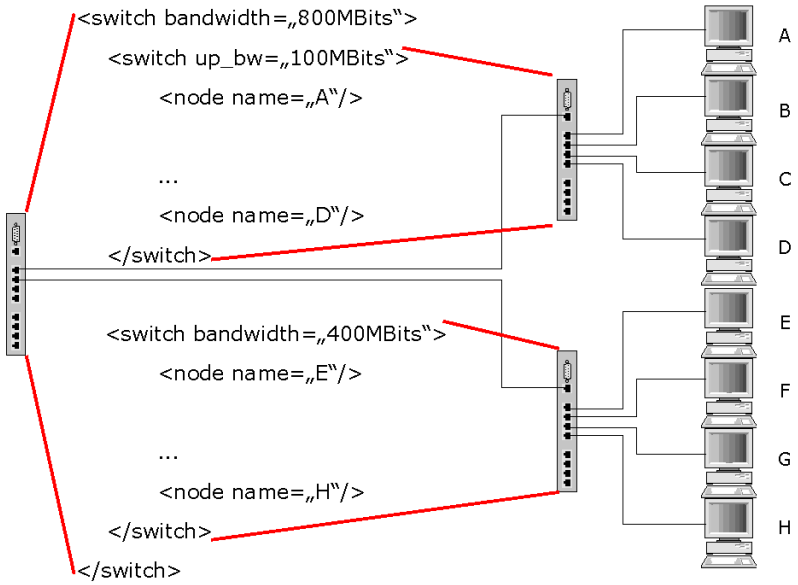

Abbildung 5.3: Gegenüberstellung von XML und Topologie

Anhand des Beispiels mit der Modemleitung aus Abbildung 5.1, soll gezeigt werden, wie verschiedene Netzwerkbandbreiten spezifiziert werden können.

```
<nsync_config>
  <repositorysystem name="DialUp-Test">
```

[3] document type definition

```
    <switch down_bw_default="34MBits" name="Internet">
      <switch bandwidth="800MBits"
        down_bw_default="100MBits" name="lokal">
        <node name="Rechner A" host="a.domain1.de"/>
        <node name="Rechner B" host="b.domain1.de"/>
        <node name="Rechner C" host="c.domain1.de"/>
      </switch>
      <node name="Rechner D" host="d.domain2.de" up_bw="56KBits"/>
    </switch>
  </repostorysystem>
</nsync_config>
```

Bandbreiten können in den Einheiten `Bits`, `KBits`, `MBits` und `GBits` angegeben werden. `1 KBits` entspricht $1000\frac{Bit}{s}$. Die Bandbreiten werden mit den Attributen `up_bw`, `bandwidth` und `down_bw_default` definiert. Mit `up_bw` am Rechner `D` wird die Bandbreite zum Switch `Internet` definiert. Die Rechner `A` bis `C` haben kein Attribut `up_bw`, deshalb wird für die Bandbreite zum Switch `lokal` der Wert `down_bw_default` des Switches verwendet. Die Bandbreite zwischen dem Switch `lokal` und dem Switch `Internet` beträgt 34 MBits. Mit `bandwidth` wird die interne Bandbreite des Switches angegeben. Dieser Wert wird benutzt, um die Topologie unter Umständen noch weiter zu optimieren, indem mehrere Komponenten zusammengefaßt werden. Keiner dieser Werte muß angegeben werden, die Beispielimplementierung kann auch ohne diese Werte die Repositories synchronisieren. Aber je mehr Informationen zur Verfügung gestellt werden, desto besser kann die Synchronisation an die Topologie angepaßt werden.

Jede beschriebene Hierarchie muß genau eine Wurzel haben. In diesem Fall ist die Wurzel der Switch `Internet`. Das Internet kann in den meisten Fällen durch einen Switch repräsentiert werden und sollte ganz oben in der Hierarchie stehen. Da sich mehrere parallele Übertragungen bei den hier auftretenden Bandbreiten über das Internet nur dann gegenseitig behindern, wenn sie die gleiche Leitung zum Internet benutzen, ist die Darstellung als Switch gerechtfertigt.

Mit dem Attribut `name` kann jedem Netzwerkelement ein beliebiger Name gegeben werden, der von der Beispielimplementation nur zum Hinzufügen von weiteren Repositories zu einem Repositorysystem verwendet wird, aber dem Benutzer das Lesen von Konfigurationen erleichtert. Das Hinzufügen von Repositories wird in Kapitel 7 beschrieben. Das Attribut `host` gibt zu jedem Rechner seine IP-Adresse an. Das Domain Name System (DNS) [19] bietet eine Abbildung zwischen IP-Adressen und für Menschen lesbareren Adressen. Dadurch ist es möglich, Rechnern Namen wie `www.zib.de` zu geben. Solche Namen können auch im Attribut `host` angegeben werden.

5.3.2 Gitter in XML

Hierarchien lassen sich in XML gut beschreiben, da sie sich durch Bäume repräsentieren lassen. Nicht ganz so elegant gelingt die Repräsentation von Gittern mit beliebig vielen Dimensionen. Im folgenden Beispiel wird eine Mischung aus Gitter und Torus be-

schrieben. Die Rechner sind in einem Quader von $2 \times 3 \times 2$ Rechnern angeordnet. Diese Topologie wird in Abbildung 5.4 graphisch dargestellt.

```
<nsync_config>
  <repositorysystem name="Torus">
    <mesh>
      <topology>
        <ring name="erste Dimension">
          <chain name="zweite Dimension">
            <switch name="dritte Dimension"></switch>
          </chain>
        </ring
      </topology>
      <dimension name="erste Dimension als Ring">
        <dimension name="zweite Dimension als Kette">
          <dimension name="dritte Dimension als Switch">
            <node name="A"/>
            <node name="B"/>
          </dimension>
          <dimension name="dritte Dimension als Switch">
            <node name="C"/>
            <node/>
          </dimension>
          <dimension name="dritte Dimension als Switch">
            <node name="D"/>
            <node/>
          </dimension>
        </dimension>
        <dimension>
          <dimension>
            <node name="E"/>
            <node/>
          </dimension>
          <dimension>
            <node/>
            <node/>
          </dimension>
          <dimension>
            <node/>
            <node/>
          </dimension>
        </dimension>
      </dimension>
    </mesh>
```

```
  </repositorysystem>
</nsync_config>
```

Gitter werden mit dem `mesh`-Tag beschrieben. Ein `mesh`-Tag besteht aus einem `topology`- und einem `dimension`-Tag. Mit dem `topology`-Tag wird beschrieben, welche Verbindungsnetzwerke für die verschiedenen Dimensionen des Gitters verwendet werden. Im obigen Beispiel werden in der ersten Dimension Ringe, in der zweiten Ketten und in der dritten Switche verwendet. Das `dimension`-Tag leitet den Beginn der Definition der einzelnen Knoten in dem Gitter ein. Jede Dimension besteht entweder aus mehreren `dimension`- oder `node`-Tags. Nur in der innersten Dimension werden Knoten eingetragen.

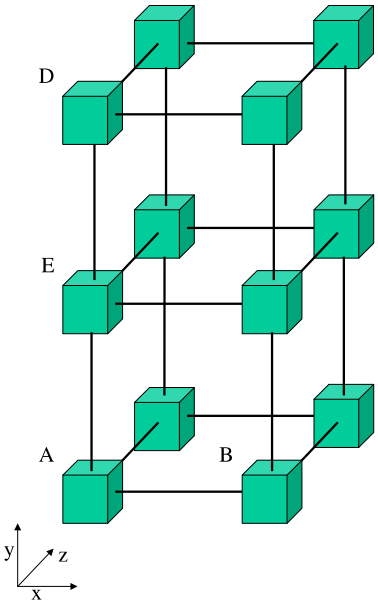

Abbildung 5.4: Graphische Darstellung des Gitters

In dem Beispiel von $2 \times 3 \times 2$ Rechnern werden im äußersten `dimension`-Tag zwei neue `dimension`-Tags definiert, in denen jeweils drei weitere definiert werden. In den innersten Tags werden dann die Knoten beschrieben. In der obigen Topologie-Beschreibung (siehe Abbildung 5.4[4]) bedeutet dies, daß die Rechner A und B über einen Switch, die Rechner A, C und D über eine Kette und die Rechner A und E über einen Ring verbunden sind.

[4]Zur Übersichtlichkeit wurden die verschiedenen Netzwerke in den Dimensionen nicht kenntlich gemacht.

5.3.3 Unterstützte Netzwerkkomponenten

In der Hierarchie können neben Gittern auch `switch`-, `bus`-, `chain`- und `ring`-Elemente verwendet werden. Das `switch`-Element repräsentiert einen Switch oder Crossbar. Ein `bus`-Element entspricht einem Bus oder Hub. Die Kinder eines `chain`-Elements sind über eine Kette verbunden und die Kinder eines `ring`-Elements über einen Ring.

Zusätzlich gibt es noch die Möglichkeit `router` zu definieren. Diese verhalten sich auch wie ein Rechner mit einem Repository, können aber noch weitere Kinder haben. Damit ist es möglich für eine Gruppe von Rechnern vorzugeben, welcher Rechner die Kommunikation zu höheren Ebenen in der Hierarchie übernehmen soll.

```
<nsync_config>
  <repositorysystem name="Torus">
    <switch name="Internet">
      <switch name="Switch A">
        <node name="A"/>
        <node name="B"/>
        <node name="C"/>
      </switch>
      <router name="Router">
        <switch name="Switch B">
          <node name="D"/>
          <node name="E"/>
        <switch>
      <router/>
    </switch>
  </repositorysystem>
</nsync_config>
```

In diesem Fall darf kein Rechner, der an den Switch `A` angeschlossen ist, direkt mit dem Rechner `D` oder `E` kommunizieren. Der Router und die Rechner `D` und `E` sind gemeinsam an den Switch `B` angeschlossen, aber der Router wird bei der Synchronisation immer als Stellvertreter für diese Gruppe gewählt.

Wie oben bereits gesagt, können mit dieser Sprache nicht alle Topologien beschrieben werden. Aber die üblichen Netzwerke lassen sich mit den vorgestellten Bausteinen repräsentieren. Andere Topologien müssen approximiert werden.

Kapitel 6

Implementierung: Synchronisationsablauf

Im Rahmen dieser Arbeit ist das Programm `nsync` entstanden, das die in den Kapiteln 2, 3, 4 und 5 vorgestellten Konzepte implementiert. Dieses Programm wird in den nächsten beiden Kapiteln präsentiert. In diesem Kapitel wird auf Details der Implementierung eingegangen und in Kapitel 7 die Installation und Bedienung beschrieben.

Der Synchronisationsablauf wird in vier Phasen unterteilt:

1. **Startphase:** Die Konfiguration des Repositorysystems wird auf dem Server aus einer Datei eingelesen und für jedes Repository eine Instanz von `nsync` auf dem jeweiligen Rechner gestartet.

2. **Planungsphase:** Wenn es Konflikte zwischen den Veränderungen an den Repositories gibt, werden diese erkannt und dem Benutzer mitgeteilt.

3. **Ausführungsphase:** Die Veränderungen an den Repositories werden innerhalb des Repositorysystems ausgetauscht und dieses dadurch synchronisiert.

4. **Endphase:** Die Metadaten über den Inhalt der Repositories werden aktualisiert und die Instanzen von `nsync` beendet.

6.1 Startphase

Der Synchronisationsablauf beginnt mit dem Aufruf von `nsync`. Diese Instanz von `nsync` wird im weiteren Server genannt. Es wird eine Konfigurationsdatei benötigt, in der die einzelnen Repositorysysteme beschrieben sind. Aus dieser werden die benötigten Informationen gelesen, dabei entsteht eine Repräsentation der Netzwerktopologie zwischen den Repositories und eine Liste der Repositories.

Der Server öffnet einen freien TCP-Port und startet einen weiteren Ausführungsstrang (Thread), der Nachrichten auf diesem Port entgegennimmt. Anschließend wird für jedes Repository eine Instanz von `nsync` mit `ssh` auf dem dazugehörigen Rechner gestartet. Diese Instanzen werden im weiteren Clients genannt.

Jeder Client liest die Metadaten, die zu seinem Repository gehören und lokal auf den Rechnern gespeichert sind, ein und vergleicht sie mit den vorhandenen Dateien. Dabei entsteht eine Liste mit allen Dateien und Verzeichnissen, die verändert wurden (siehe dazu Abschnitt 3.1). Zusätzlich wird das Datenvolumen der Änderungen an jedem Repository abgeschätzt, dazu werden die Größen aller veränderten Dateien je Repository aufsummiert. Die Größe der Veränderungen kann nur abgeschätzt werden, da zu diesem Zeitpunkt nicht bekannt ist, ob für die Übertragung ein Kompressionsverfahren oder `rsync` benutzt werden kann. Für die Bestimmung der Kommunikationsstrategie reicht aber eine Näherung aus.

Nach dem Einlesen der Metadaten öffnet jeder Client einen TCP-Port und sendet dem Server eine Nachricht, in der die Nummer dieses Ports und die approximierte Größe der Veränderungen seines Repository steht. Der Server hat den Clients dafür seine Port-Nummer und seine IP-Adresse beim Start als Kommandozeilenparameter mitgeteilt.

Nachdem sich alle Clients beim Server angemeldet haben, hat dieser eine Liste mit den IP-Adressen und Ports der Clients. Außerdem ist ihm bekannt, an welchen Repositories Veränderungen vorgenommen wurden und wie groß diese ungefähr sind. Mit diesen Informationen kann der Server eine Kommunikationsstrategie für die Synchronisation dieses Repositorysystems berechnen, dazu wird auf die Approximationen aus dem letzten Kapitel zurückgegriffen. Die dabei entstandene Lösungssequenz wird zusammen mit den Adressen und Ports der Clients vom Server an alle Clients gesendet.

6.1.1 Konfigurationsdatei und Graphgenerierung

Die Konfigurationsdatei ist, wie in Kapitel 5 erläutert, XML-basiert [4]. Für den Zugriff auf die XML-Daten wird ein DOM(Document Object Model)-Interface [34] verwendet, daß von der `xerces`-Bibliothek zur Verfügung gestellt wird. Die Bibiliothek überprüft dabei, ob die XML-Daten der DTD aus Anhang A entsprechen (validierender Parser). Das DOM-Interface präsentiert die XML-Datei als Baum von Objekten. Dieser Baum wird in eine interne Repräsentation übersetzt, die es erlaubt, die Approximationen aus Kapitel 5 effizient anzuwenden. In Abbildung 6.1 sind die Beziehungen zwischen den Klassen, die für die interne Repräsentation der Topologie verwendet werden, in einem UML [20]-Klassendiagramm dargestellt.

6.2 Planungsphase

In der Planungsphase wird die vom Server berechnete Lösungssequenz einmal abgearbeitet. Dabei werden bei jedem Verbindungsaufbau zwischen zwei Repositories die Namen der in den Repositories veränderten Dateien und Verzeichnisse ausgetauscht. Die Veränderungen schließen auch die mit ein, die bis zu diesem Zeitpunkt von anderen Repositories empfangen wurden. Während der Abarbeitung der Lösungssequenz entsteht in jedem Repository eine Liste von Operationen (Datei löschen, senden oder empfangen), die durchgeführt werden müssen, um das gesamte System zu synchronisieren. Außerdem können Kanten aus dem Graphen entfernt werden, über die keine Daten

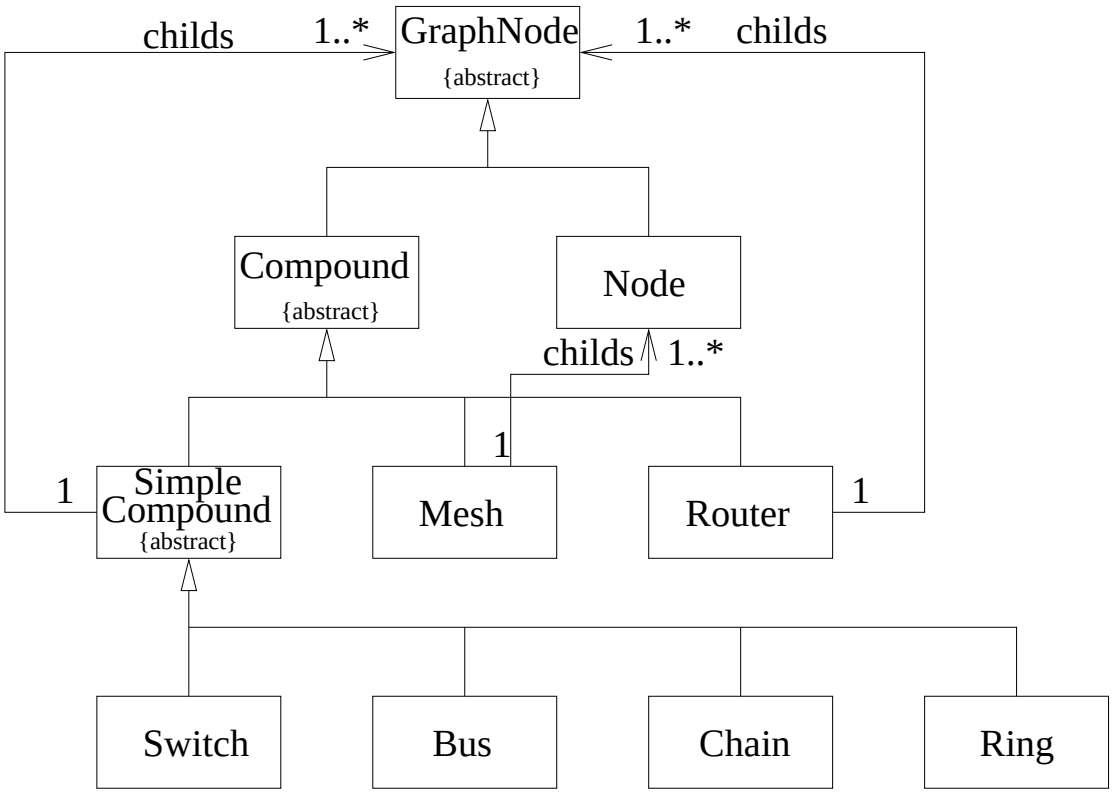

Abbildung 6.1: UML Klassendiagramm der internen Repräsentation der Topologie

übertragen werden müssen. Das eigentliche Ziel der Planungsphase ist aber das Erkennen von Konflikten, das heißt Dateien beziehungsweise Verzeichnissen, die in mindestens zwei Repositories unterschiedlich verändert wurden. Das Verfahren dafür wird genauer in Abschnitt 6.2.1 vorgestellt. Die Konflikte werden wie die Operationen in einer Liste gespeichert.

Am Ende der Planungsphase sendet jeder Client alle Konflikte, die ihm bekannt geworden sind, an den Server oder meldet, daß keine Konflikte erkannt wurden. Der Server wartet, bis ihm von jedem Client das Ende der Planungsphase mitgeteilt wurde. Wenn Konflikte aufgetreten sind, werden diese dem Benutzer mitgeteilt, und der Server sendet den Clients eine Nachricht, daß die Synchronisation abgebrochen wird. Wenn es zu keinen Konflikten kam, wird den Clients mitgeteilt, daß sie mit der Ausführungsphase beginnen können.

6.2.1 Erkennen von Konflikten/Operationen

Vor der Planungsphase werden auf jedem Rechner die Namen aller Dateien, die verändert wurden, in eine Liste eingetragen. Zusätzlich wird zu jeder Datei zunächst das Repository vermerkt, in dem sie vom Benutzer verändert wurde. Zu diesem Zeitpunkt sind in den

Repositories nur die lokalen Veränderungen bekannt, deshalb wird zu jeder Datei das eigene Repository vermerkt. Die Liste könnte dann wie in Abbildung 6.1 aussehen. In diesem Beispiel werden die Rechnernamen, auf denen die Repositories gespeichert sind, zur Identifizierung des Ursprungs der Veränderung verwendet, in der Implementierung wird stattdessen für jedes Repository ein eindeutiger Index gespeichert.

Dateiname	Ursprung der Veränderung
`foo/donald`	`hosta.domain.de`
`foo/dagobert`	`hosta.domain.de`
`foo/gustav`	`hosta.domain.de`
`foo/mickey`	`hosta.domain.de`

Tabelle 6.1: Lokale Tabelle der Veränderungen auf `hosta`

Um die Liste der Operationen zu erstellen, werden in der Planungsphase für jedes Element in der Lösungssequenz Listen der auf den beteiligten Rechnern veränderten Dateien miteinander verglichen. Mit Hilfe des Index/Rechnernamens kann bei Dateinamen, die in beiden Listen vorkommen, überprüft werden, ob es sich um einen Konflikt handelt.

Wenn die Datei auf beiden Rechnern auf dieselbe Art verändert wurde, liegt kein Konflikt vor. Solche Fälle können zum Beispiel auftreten, wenn der Benutzer Konflikte löst. Der Benutzer hat in mehreren Repositories dieselbe Datei so verändert, daß alle anschließend denselben Inhalt haben. In diesem Fall handelt es sich um keinen Konflikt. Um zu verhindern, daß in einer solchen Situation fälschlicherweise ein Konflikt gemeldet wird, werden MD5-Checksummen [27] beider Dateien berechnet und diese verglichen. Nur wenn auch die Checksummen unterschiedlich sind, handelt es sich um einen Konflikt, in diesem Fall wird der Dateiname, wie oben erwähnt, in einer Liste für Konflikte gespeichert. Falls ein Dateiname nur in der Liste eines Repositories vorkommt, wird dieser zu den Operationen hinzugefügt, die in der Ausführungsphase durchgeführt werden.

Dateiname	Ursprung der Veränderung
`foo/donald`	`hostb.domain.de`
`foo/daisy`	`hostb.domain.de`
`foo/minnie`	`hostb.domain.de`
`foo/gundel`	`hostb.domain.de`

Tabelle 6.2: Lokale Tabelle der Veränderungen auf `hostb`

Am Beispiel von `hosta.domain.de` und `hostb.domain.de` wird dieses Verfahren verdeutlicht. Die Tabellen für die beiden Rechner sind in den Abbildungen 6.1 und 6.2 dargestellt. Die Datei `foo/donald` wurde auf beiden Rechnern verändert, deshalb wird mit einer Checksumme überprüft, ob sich der Inhalt der beiden Dateien unterscheidet. Wenn sich die beiden Checksummen unterscheiden, wird `foo/donald` zu der Liste mit den Kon-

flikten hinzugefügt, ansonsten muß die Datei nicht beachtet werden, da der Inhalt beider Replikate gleich ist. Die Dateien `foo/dagobert`, `foo/gustav` und `foo/mickey` müssen von `hosta` nach `hostb` übertragen werden und die Dateien `foo/daisy`, `foo/minnie` und `foo/gundel` in die andere Richtung. Sie werden entsprechend zu der Liste mit den Operationen hinzugefügt.

Dateinamen, die nur in der empfangenen Liste vorkamen, werden in die lokale Liste mit aufgenommen. Dadurch wird sichergestellt, daß Veränderungen an alle Repositories propagiert werden und Konflikte zwischen Repositories erkannt werden können, die nicht direkt miteinander kommunizieren. In diesem Beispiel bedeutet das, daß auf beiden Rechnern die Liste der Veränderungen nach dem Abgleich wie Tabelle 6.3 aussieht (auf `hostb` wird für `foo/donald` als Ursprung `hostb.domain.de` gespeichert).

Dateiname	Ursprung der Veränderung
`foo/donald`	`hosta.domain.de`
`foo/dagobert`	`hosta.domain.de`
`foo/gustav`	`hosta.domain.de`
`foo/mickey`	`hosta.domain.de`
`foo/daisy`	`hostb.domain.de`
`foo/minnie`	`hostb.domain.de`
`foo/gundel`	`hostb.domain.de`

Tabelle 6.3: Lokale Tabelle der Veränderungen auf `hosta` nach dem Abgleich

6.3 Ausführungsphase

Nachdem den Clients mitgeteilt wurde, daß keine Konflikte aufgetreten sind, arbeiten sie die Liste der Operationen ab. In dieser Liste steht nur, was für eine Änderung am Verzeichnisbaum aufgetreten ist, aber nicht, welches Verfahren zur Synchronisation der Veränderungen angewendet werden soll. Auf dem Rechner, auf dem die Änderung vorliegt, wird entschieden, wie die Änderung dem anderen Repository mitgeteilt werden kann. Dafür stehen folgende Möglichkeiten zur Verfügung:

- unkomprimiertes Senden der Datei
- komprimiertes Senden der Datei mit bzip2
- synchronisieren mit `rsync`
- anlegen eines Verzeichnisses
- löschen einer Datei/eines Verzeichnisses
- ändern der Zugriffsrechte einer Datei

Nachdem jeder Client seine Liste mit den Operationen abgearbeitet hat, werden die Metadaten über die Repositories aktualisiert. Anschließend meldet jeder Client dem Server, daß er die Synchronisation beendet hat, und beendet sich selbst.

6.3.1 Veränderungen am Dateisystem während der Synchronisation

Gesondert ist der Fall zu betrachten, wenn der Benutzer während der Synchronisation Änderungen in den Repositories durchführt. Es können Veränderungen an Dateien vorgenommen werden, ohne daß dieses von `nsync` erkannt werden kann. In diesem Fall kann es zu Inkonsistenzen zwischen den Repositories kommen, die nicht synchronisiert werden, da sie `nsync` nicht bekannt sind. Wenn `nsync` und ein Benutzer gleichzeitig in eine Datei schreiben, können Dateien entstehen, die weder der Datei, die `nsync` schreiben wollte, entsprechen noch der Datei, die der Benutzer schreiben wollte.

Im Posix-Standard[1] [21] ist keine Möglichkeit definiert, die exklusiven Zugriff auf Dateien ermöglicht. Es gibt lock-Funktionen, die aber nur funktionieren, wenn alle beteiligten Applikationen kooperieren. Allgemein kann aber nicht davon ausgegangen werden, daß alle Applikationen die lock-Mechanismen berücksichtigen. Deshalb kann nur eine Näherung für exklusiven Zugriff auf Dateien erreicht werden. Diese Näherung kann die meisten Zugriffe auf Dateien erkennen, aber wenn Zugriffe zu ungünstigen Zeitpunkten stattfinden, können diese nicht erkannt werden.

Beim Start ermittelt jeder Client die aktuelle lokale Uhrzeit (Startzeit). Wenn `nsync` eine Datei geschrieben hat, wird der Zeitpunkt der letzten Änderung an dieser Datei auf die Startzeit zurückgesetzt. Bevor in eine Datei geschrieben wird, überprüft `nsync`, ob der Zeitpunkt der letzten Änderung an dieser Datei nach der Startzeit liegt. In diesem Fall wurde während der Synchronisation von einer anderen Applikation schreibend auf diese Datei zugegriffen. Dateien, die während der Synchronisation verändert wurden, werden zunächst ignoriert. In den Metadaten wird die Startzeit als Zeitpunkt der letzten Synchronisation vermerkt, dadurch werden Dateien, die während einer Synchronisation verändert werden, bei der nächsten Synchronisation als verändert erkannt und erneut synchronisiert.

Damit können die meisten Zugriffe auf Dateien durch Benutzer während einer Synchronisation erkannt werden. Wenn Benutzer und `nsync` nach der Datumsüberprüfung gleichzeitig auf eine Datei zugreifen, kann dies nicht erkannt werden.

In Abbildung 6.2 wird der gleichzeitige Zugriff auf eine Datei von `nsync` und einem Benutzer dargestellt. `nsync` prüft zunächst mit dem Befehl `stat`, ob nach dem Beginn der Synchronisation von einem Benutzer auf diese Datei zugegriffen wurde. Danach öffnet der Benutzer die Datei zum Schreiben, dabei wird die Dateilänge auf 0 zurückgesetzt, und `nsync` öffnet die Datei zum Lesen. Der Benutzer schreibt einen Block in die Datei und schließt sie anschließend. Nachdem die Daten vom Benutzer geschrieben wurden, liest `nsync` diese aus, schließt die Datei und setzt die Zeit des letzten Zugriffs zurück. In diesem Beispiel liest `nsync` Daten, die vom Benutzer während der Synchronisation

[1] Der Posix-Standard definiert eine API (Application Programming Interface) für Basisoperationen eines Betriebssystem, die von den meisten Unices unterstützt wird.

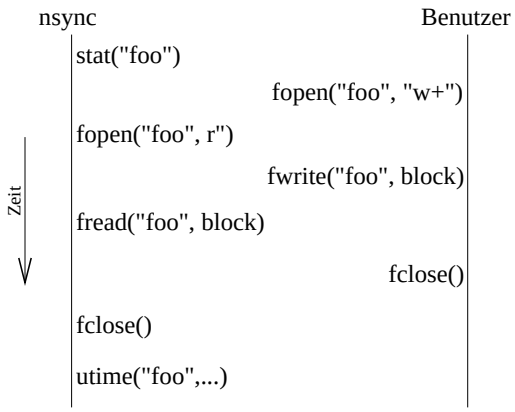

Abbildung 6.2: Zeitlicher Ablauf von 2 gleichzeitigen Zugriffen auf eine Datei

geschrieben wurden, ohne daß dieses von `nsync` bemerkt werden kann. Dabei können von `nsync` unbemerkte Inkonsistenzen zwischen den Repositories entstehen.

6.4 Endphase

Der Server wartet, bevor er sich beendet, bis ihm von allen Clients mitgeteilt wurde, daß die Synchronisation durchgeführt wurde.

Der Server wurde im gesamten Ablauf nur zur Berechnung der Kommunikationsstrategie und für Kontrollaufgaben verwendet. Er ist nicht am Datenaustausch zwischen den Clients beteiligt, deshalb ist die Ausführungszeit der Synchronisation nicht durch die Prozessorleistung bzw. Netzwerkbandbreite des Servers begrenzt.

6.5 Datenübertragungsprotokoll

In Kapitel 2 wurde beschlossen, zur Kommunikation das TCP-Protokoll [25] einzusetzen. Statt TCP könnte auch das UDP-Protokoll [23] verwendet werden, das auch zur Familie der Internet-Protokolle gehört und deshalb ebenfalls auf den meisten Rechnern vorhanden ist. In [29] werden die beiden Protokolle wie folgt beschrieben:

> *Transmission Control Protocol (TCP).* A connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process.
>
> *User Datagram Protocol (UDP).* A connectionless protocol for user processes. Unlike TCP, which is reliable, there is no guarantee that UDP datagrams ever reach their intended destination.

Für `nsync` ist entscheidend, daß mit TCP ein kontinuierlicher Datenstrom und mit UDP Datenpakete übertragen werden. Außerdem wird bei UDP nicht garantiert, daß die

Pakete ihr Ziel erreichen. Für TCP wird dieses auch nicht garantiert, aber es sind Mechanismen vorgesehen, die in einem gewissen Rahmen den Empfang ausgesendeter Pakete sicherstellen sollen. Solange es keine Hardware- oder Softwareausfälle gibt, kann beim TCP-Protokoll davon ausgegangen werden, daß gesendete Daten auch beim Empfänger ankommen.

Abbildung 6.3: Protokollaufbau

Für diese Implementierung wäre eine Mischung aus beiden Protokollen wünschenswert gewesen (*reliable* und *datagram* basiert). Die Datenpaket-Abstraktion ermöglicht es ohne großen Aufwand, Befehle bzw. Inhalte von Dateien zwischen den Rechnern auszutauschen. Wenn wie bei TCP nur ein Datenstrom zur Verfügung steht, müssen Methoden entwickelt werden, um die Befehle bzw. Inhalte der Dateien im Strom voneinander abzugrenzen.

Bei der Implementierung von `nsync` wurde das TCP-Protokoll gewählt und darauf eine weitere Schicht entwickelt, die den Austausch von Paketen zwischen den beiden Kommunikationsteilnehmern ermöglicht. Des weiteren bietet diese Schicht die Möglichkeit, innerhalb einer TCP-Verbindung mehrere logische Kommunikationskanäle zu definieren.

Dafür wird vor jedes Datenpaket ein Header (siehe Abbildung 6.3) gesetzt, der die Länge des Paketes, eine Channel-ID und eine Message-ID enthält. Mit der Channel-ID werden die Pakete den Kanälen zugeordnet und mit der Message-ID wird der Typ der Nachricht bestimmt. Jeder Message-ID ist eine C++-Klasse [30] zugeordnet, dadurch ist die Kommunikation typsicher.

Dieses Design basiert auf dem MPI-Standard [18], wobei die logischen Kanäle den `message tags` bei MPI entsprechen. Bei diesem Ansatz müssen auf den Rechnern keine MPI-Implementierungen für Weitverkehrsnetze installiert sein und trotzdem stehen ähnliche Eigenschaften wie bei MPI zur Verfügung.

Das Protokoll ist mit einer objektorientierten Schnittstelle gekapselt, dadurch ist es möglich, die Kommunikation von `nsync` auf einem abstrakten Niveau unabhängig von den tatsächlichen genutzten Protokollen zu implementieren. In den Abbildungen 6.4 und 6.5 ist der Quelltext zum Versenden und Empfangen von Nachrichten dargestellt.

6.6 Benchmarks

Um die Skalierbarkeit der Implementierung zu überprüfen, wurden einige Versuche mit verschieden großen Repositorysystemen durchgeführt. Für die Versuche enthielt ein Repository den Quellcode des Linux-Kernels in der Version 2.2.13 (vgl. Kapitel 3), dieser umfaßt ca. 72 MB, während die anderen Repositories leer waren. In den Tests mußte

```
// Nachricht erzeugen und Felder belegen
Message *msg = new AnnouncePortMessage(my_port);
// Nachricht über Kanal channel versenden
channel->sendMessage(msg); //Speicher der Nachricht freigeben
delete msg;
```

Abbildung 6.4: Nachricht versenden

```
// Nachricht empfangen
Message *msg = channel->getNextMessage();
// überprüfen, ob es eine AnnouncePort-Nachricht ist
if(dynamic_cast<AnnouncePortMessage *>(msg) != NULL){
  AnnouncePortMessage *apmsg;
  // Typ-Umwandlung
  apmsg = dynamic_cast<AnnouncePortMessage *>(msg);
  // Felder der Nachricht auslesen
  his_port = apmsg->getPort();
}else{
  /* ... */
}
delete msg;
```

Abbildung 6.5: Nachricht empfangen

der Quellcode an die leeren Repositories verteilt werden. Die Laufzeiten wurden mit dem Modell aus Kapitel 5 vorhergesagt und anschließend mit den gemessenen Werten verglichen.

Für die Tests standen 4 Rechner mit 800 MHz PIII-Prozessoren und ein Rechner mit einem P4-Prozessor mit 1700 MHz zur Verfügung. Der letztgenannte wurde nur für den Versuch mit 5 Repositories verwendet. Die Rechner sind über ein geswitchtes Netzwerk mit 100 MBits Bandbreite verbunden. Alle Rechner waren wie bei den Versuchen in Kapitel 3 mit genügend Hauptspeicher versehen, so daß die Festplatten keinen Einfluß auf die Messungen hatten.

Anzahl der Repositories	gemessene Laufzeit	vorhergesagte Laufzeit	Effizienz
2	14.6s	5.7s	39.0%
3	28.9s	11.5s	39.8%
4	26.9s	11.5s	42.8%
5	32.7s	17.2s	52.6%

Tabelle 6.4: Ergebnisse der Versuche mit `nsync`

Die Laufzeiten, die mit dem Modell aus Kapitel 5 ermittelt wurden (siehe Tabelle 6.4), waren immer kürzer als die gemessenen Zeiten, da im Modell angenommen wird, daß die gesamte Netzwerkbandbreite genutzt werden kann. Bei den ersten drei Meßpunkten (siehe Abbildung 6.6) konnte ca. 40% der vorhergesagten Leistung erreicht werden, während beim letzten Meßpunkt ca. 50% erreicht wurden. Das Abweichen des letzten Wertes kann durch den P4 zustande gekommen sein, da dieser leistungsfähiger als die PIIIs ist, aber im wesentlichen skalierte `nsync` im Rahmen der begrenzten Versuchsgröße wie erwartet.

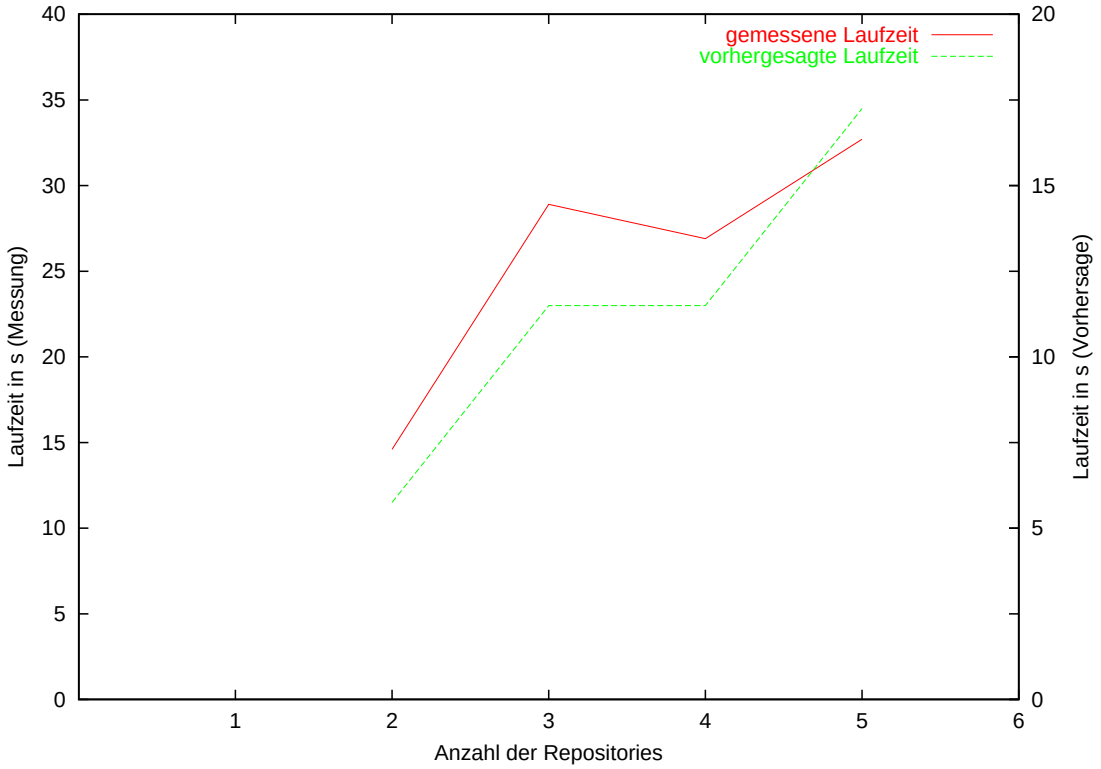

Abbildung 6.6: Messungen mit verschieden großen Repositorysystemen

Kapitel 7

Implementierung: Benutzerinterface und Konfiguration

In diesem Kapitel wird eine Einführung in das Benutzerinterface von `nsync` gegeben, dabei wird die Installation, Konfiguration und Bedienung vorgestellt.

7.1 Installation

Es wird davon ausgegangen, daß `nsync` auf einem Unix-System installiert werden soll. Vor der Installation muß vom Benutzer überprüft werden, ob die folgenden Komponenten installiert sind.

- ssh: http://www.openssh.org
- libbz2-Bibliothek: http://sources.redhat.com/bzip2/
- Xerces-Bibliothek http://xml.apache.org/
- librsync-Bibliothek: http://rproxy.samba.org

Bei `ssh` kann davon ausgegangen werden, daß es installiert ist. Die `libbz2`-Bibliothek sollte auf aktuellen Systemen ebenfalls vorhanden sein. Die `librsync`-Bibliothek implementiert den Rsync-Algorithmus und befindet sich noch in der Entwicklung, deshalb wird sie vom Benutzer installiert werden müssen. Außerdem werden GNU `Make` und ein C++ Compiler (gcc) benötigt.

Wenn diese Komponenten vorhanden sind, kann die Installation wie folgt durchgeführt werden:

```
> tar xvzf nsync-x.y.z.tar.gz
> cd nsync-x.y.z
> make
```

Danach sollte sich in `$HOME/bin` eine ausführbare Datei namens `nsync` befinden.

```
<?xml version="1.0" ?>
<!DOCTYPE dsync_config SYSTEM "config.dtd">
<nsync_config>
  <repositorysystem name="demo-all">
    <switch>
      <node host="hosta.domain.de"
        path="/home/f/foo/path_a" username="foo"/>
      <node host="hostb.domain.de"
        path="/home/bar/path_b" username="bar"/>
    </switch>
  </repositorysystem>
  <repositorysystem name="demo-source" pattern=".*\.[ch]">
    <switch>
      <node host="hosta.domain.de"
        path="/home/f/foo/path_a" username="foo"/>
      <node host="hostb.domain.de"
        path="/home/bar/path_b" username="bar"/>
    </switch>
  </repositorysystem>
</nsync_config>
```

Abbildung 7.1: Konfigurationsbeispiel

7.2 Konfiguration

Zur Spezifikation der Repositorysysteme verwendet `nsync` eine Datei, die unter `$HOME/.nsync/config.xml` gespeichert sein muß. Da die Konfigurationsdatei, wie in Kapitel 6 beschrieben, nur vom Server gelesen wird, muß die Konfigurationsdatei nur auf den Rechnern vorhanden sein, auf denen `nsync` direkt vom Benutzer aufgerufen wird. Wenn zum Beispiel mit `nsync` vom Arbeitsplatzrechner HTML-Dateien auf mehrere Web-Server verteilt werden sollen und `nsync` nur auf dem Arbeitsplatzrechner direkt vom Benutzer aufgerufen wird, muß die Konfigurationsdatei nicht auf den Web-Servern gespeichert sein. Die Konfigurationsdatei basiert auf der Topologiebeschreibungssprache aus Kapitel 5. Da es eine XML-Datei ist, muß ein eindeutiges Wurzelelement existieren, für `nsync` ist dieses das `nsync_config`-Element.

In Abbildung 7.1 wird ein Repositorysystem mit dem Namen „demo-all“ beschrieben. Die Topologie des Netzwerkes zwischen den Repositories wird, wie in Kapitel 5 erklärt, beschrieben. Mit jedem `node`-Tag wird ein Repository deklariert. Diese Namensgebung suggeriert fälschlicherweise, daß auf einem Rechner nicht mehr als ein Repository gespeichert sein darf. Mit Knoten sind hier aber nicht Rechner sondern Repositories gemeint. Wenn in dem obigen Beispiel der Wert von `host` immer auf `hosta.domain.de` gesetzt ist, wird damit ein Repositorysystem deklariert, bei dem beide Repositories auf demselben Rechner gespeichert sind.

Neben den in Kapitel 5 vorgestellten Attributen besitzen `nodes` die Attribute `path` und `username`. Mit `path` wird das Verzeichnis angegeben, indem das Repository gespeichert ist. Mit `username` wird der Name des Besitzer dieses Verzeichnisses angegeben. Dieser Name wird auch für `ssh` benutzt, um auf dem Rechner eine Instanz von `nsync` zu starten.

Für jedes Repositorysystem kann ein regulärer Ausdruck [22] angegeben werden, mit dem die Synchronisation auf Dateien beschränkt wird, die dem Ausdruck entsprechen. In der Konfiguration in Abbildung 7.1 sind zwei Repositorysysteme beschrieben, wobei das zweite nur Dateien synchronisiert, die auf `.c` oder `.h` enden. Mit `demo-all` werden alle Dateien synchronisiert und mit `demo-source` wird nur C-Quellcode synchronisiert.

Bei der Synchronisation werden grundsätzlich alle Unterverzeichnisse des Repositories untersucht. Für jedes Verzeichnis und jede Datei, die dabei gefunden werden, wird der Pfad relativ zur Wurzel des Repositories mit dem angegeben regulären Ausdruck verglichen. Dadurch ist es möglich, zum Beispiel nur C-Quellcode synchronisieren, der im Wurzelverzeichnis gespeichert ist. Mit `^[!/]*\.[ch]$` als regulären Ausdruck wird C-Quellcode, der sich in einem Unterverzeichnis des Repositories befindet, ignoriert. Der reguläre Ausdruck stimmt mit alle Zeichenketten überein, die kein `/` enthalten und mit `.c` oder `.h` enden.

Wird in der Konfigurationsdatei kein regulärer Ausdruck angegeben, wird `^.*$` angenommen. Dieser Ausdruck stimmt mit allen Pfaden überein.

7.3 Bedienung

Mit folgendem Befehl wird das Repositorysystem „demo-all“ synchronisiert.

```
> nsync -r demo-all
```

Bei der ersten Synchronisation eines Repositorysystems wird angenommen, daß alle Repositories leer waren und alle Dateien und Verzeichnisse, die sich jetzt in den Repositories befinden, als Veränderungen hinzukamen. Dabei kann es zu Konflikten kommen, wenn sich in mindestens zwei Repositories Dateien mit gleichem Namen und unterschiedlichem Inhalt befinden. Wenn `nsync` Konflikte entdeckt hat, werden diese dem Benutzer mitgeteilt, wobei der Dateiname und die Repositories, in denen die Datei verändert wurde, angegeben werden. Die Konflikte müssen dann vom Benutzer mit Hilfe der üblichen Unix Hilfsmittel gelöst werden. Dazu muß der Benutzer die verschiedenen Versionen der Datei zu einer zusammenfassen bzw. die neueste Version auswählen. Anschließend muß diese Version in alle Repositories, die den Konflikt gemeldet haben, kopiert werden. Wenn mehrere Konflikte vorlagen, müssen nicht alle sofort gelöst werden. Ungelöste Konflikte werden beim nächsten Aufruf von `nsync` erneut gemeldet.

Wenn neue Repositories zu einem System hinzugefügt werden sollen, geschieht dieses mit dem Parameter `--add`. Zuerst müssen die neuen Repositories in die Konfigurationsdatei (siehe Abbildung 7.2) eingefügt werden. In diesem Beispiel wurden `neu1` und `neu2` hinzugefügt. Die neuen Repositories müssen ein `name`-Tag besitzen, mit dem sie eindeutig identifiziert werden können. Anschließend werden sie dann mit folgendem Befehl in das Repositorysystem integriert:

```
...
  <repositorysystem name="demo-source" pattern=".*\.[ch]">
    <switch>
      <node host="hosta.domain.de"
        path="/home/f/foo/path_a" username="foo"/>
      <node host="hostb.domain.de"
        path="/home/bar/path_b" username="bar"/>
      <node host="hostc.domain.de" name="neu1"
        path="/home/bar/path_c" username="braz"/>
      <node host="hostd.domain.de" name="neu2"
        path="/home/bar/path_d" username="ofo"/>
    </switch>
  </repositorysystem>
...
```

Abbildung 7.2: XML-Beispiel für das Hinzufügen von Repositories

```
> nsync -r demo-source --add neu1,neu2
```

Beim Hinzufügen der Repositories werden die Namen der Repositories durch „,“ getrennt an der Kommandozeile angegeben. Dabei muß der Benutzer besonders sorgfältig sein, da `nsync` in diesem Fall eine Kopie eines bereits vorhandenen Repositories in das neue kopiert. Dateien, die mit dem regulären Ausdruck für dieses Repositorysystem übereinstimmen und in den neuen Repositories gespeichert waren, aber nicht in den vorhandenen, werden gelöscht! Danach enthalten alle Repositories die gleichen Dateien.

Kapitel 8

Zusammenfassung

8.1 Ergebnisse

In dieser Arbeit wurde eine neue effiziente Methode zur Synchronisation von Verzeichnisstrukturen, die auf verschiedenen Rechnern gespeichert sind, vorgestellt. Der Unterschied zu existierenden Verfahren besteht in der Möglichkeit, daß die Netzwerktopologie zwischen den Rechnern, auf denen die Repositories gespeichert sind, bei der Synchronisation mitberücksichtigt werden kann. Dazu werden Punkt-zu-Punkt-Synchronisationen zu einer Synchronisation des gesamten Systems komponiert. Für die Kompositionen wurde auf Basis der Lösungen des Gossip-Problems ein Verfahren entwickelt, das neben der Topologie auch die Bandbreiten der Netzwerkverbindungen und die Verteilung der Veränderungen über die Repositories in eine effiziente Kommunikationsstrategie einfließen läßt. Dabei wird versucht, die Kommunikationslast über alle beteiligten Rechner zu verteilen, während bei einer zentralen Speicherung der Daten der Server zum Flaschenhals werden könnte. Deshalb ist es mit diesem Verfahren auch möglich, große Systeme von Repositories zu synchronisieren.

Für die Datenübertragungen bei der Punkt-zu-Punkt-Synchronisation werden sowohl Kompressionsalgorithmen, als auch der Rsync-Algorithmus eingesetzt. Dieser ermöglicht es, die Differenz zwischen zwei Dateien zu übertragen, ohne daß beide Dateien komplett vorliegen haben. Dadurch ist es möglich, bei veränderten Dateien die zu übertragenden Daten gegenüber einer herkömmlichen Datenkompression zu reduzieren. Die Wahl des Übertragungsverfahren wird von der Netzwerkbandbreite zwischen den Rechnern abhängig gemacht.

Zum Erkennen der veränderten Dateien wurde eine Methode entwickelt, die nur sehr wenig Rechenaufwand benötigt. Während das Erkennen der Veränderungen mit Checksummen viel Rechenzeit und Zugriff auf die komplette Datei benötigt, werden hier nur der Zeitpunkt der letzten Veränderung der Datei und einige Metadaten benötigt. Außerdem ist es nicht nötig, jede Datei komplett zu lesen.

Die Beispielimplementierung deckt ein weites Einsatzspektrum ab. Sie kann sowohl bei Entwicklung von Software auf mehreren Systemen zur Synchronisation des Quelltextes zwischen den Rechnern verwendet werden, als auch zur Verteilung von Datensätzen

in großen Clustersystemen, um lokalen Zugriff auf die Daten zu ermöglichen.

8.2 Ausblick

Die Implementierung kann noch in einigen Punkten verbessert werden. So kann `nsync` im Moment noch nicht damit umgehen, falls ein Knoten während der Synchronisation ausfällt. Für den Fall, daß ein Knoten schon beim Starten der Synchronisation nicht erreichbar ist, existiert bereits eine Lösung, sie muß allerdings noch implementiert werden. Wenn Knoten nicht erreichbar sind, werden sie beim Erzeugen der Kommunikationsstrategie ausgelassen und bei der nächsten Synchronisation mitsynchronisiert. `nsync` könnte auch um Funktionen zum Erkennen der Netzwerktopologie zwischen den Rechnern erweitert werden.

Die Verfahren zur Datenübertragung können noch verbessert werden. Einerseits sollten aus dem Programm `rsync` einige Optimierungen übernommen werden, andererseits könnte `bzip2` parallelisiert werden, um von SMP[1]-Knoten profitieren und dann auch Datenkompression bei schnelleren Verbindungen sinnvoll einsetzen zu können.

Beim Verteilen der Lösungssequenz vom Server könnte der Broadcast zusätzlich anhand der Topologie optimiert werden. Zur Zeit sendet der Server jedem Client direkt die Sequenz.

Im Rahmen des DataGrid-Projektes [7] könnte `nsync` zu einem Replika- und Migrationssystem erweitert werden, indem ein weiteres Tool die Migrationsanfragen entgegennimmt und entsprechend die Konfigurationsdatei anpaßt. Datensätze könnten so gezielt in verschiedene Tier 0-, 1-, oder 2-Zentren verteilt werden. Veränderungen an den Datensätzen in einem Zentrum können dadurch an alle anderen Zentren propagiert werden, die diesen Datensatz ebenfalls gespeichert haben.

[1] symmetric multiprocessor

Anhang A

DTD für die Topologiebeschreibungssprache

Diese DTD (Document Type Definition) [4] definiert die Sprache, mit der die Topologien und die Repositories beschrieben werden. Genaue Erläuterungen zu den einzelnen Parametern gibt es in Kapitel 5.

```
<?xml encoding="ISO-8859-1" ?>

<!ELEMENT nsync_config (repository)+>
<!ELEMENT repositorysystem   (node | switch | bus | chain |
                         ring | mesh)>
<!ELEMENT switch       (node | switch | bus | chain | ring |
                         mesh | router)*>
<!ELEMENT bus          (node | switch | bus | chain | ring |
                         mesh | router)*>
<!ELEMENT chain        (node | switch | bus | chain | ring |
                         mesh | router)*>
<!ELEMENT ring         (node | switch | bus | chain | ring |
                         mesh | router)*>
<!ELEMENT router       (node | switch | bus | chain | ring |
                         mesh | router)>
<!ELEMENT mesh         (topology, dimension)>
<!ELEMENT topology     (       switch | bus | chain | ring)>
<!ELEMENT dimension    (dimension* | node*)>
<!ELEMENT node         EMPTY>

<!ATTLIST nsync_config >
<!ATTLIST repositorysystem
  name        CDATA #REQUIRED
  pattern     CDATA #IMPLIED
>
```

```
<!ATTLIST switch
  name              CDATA #IMPLIED
  bandwidth         CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
  down_bw_default   CDATA #IMPLIED
>
<!ATTLIST bus
  name              CDATA #IMPLIED
  bandwidth         CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
  down_bw_default   CDATA #IMPLIED
>
<!ATTLIST chain
  name              CDATA #IMPLIED
  bandwidth         CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
>
<!ATTLIST ring
  name              CDATA #IMPLIED
  bandwidth         CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
>
<!ATTLIST mesh
  name              CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
>
<!ATTLIST topology
  name              CDATA #IMPLIED
>
<!ATTLIST dimension
  name              CDATA #IMPLIED
>
<!ATTLIST node
  name              CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
  host              CDATA #REQUIRED
  path              CDATA #REQUIRED
  username          CDATA #REQUIRED
  ssh_proxy         CDATA #IMPLIED
>
<!ATTLIST router
  name              CDATA #IMPLIED
  up_bw             CDATA #IMPLIED
  host              CDATA #REQUIRED
```

```
  path              CDATA #REQUIRED
  username          CDATA #REQUIRED
  ssh_proxy         CDATA #IMPLIED
>
```

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit alleine und ohne fremde Hilfe angefertigt habe. Alle verwendeten Hilfsmittel sind dokumentiert.

Thorsten Schütt

Literaturverzeichnis

[1] Brenda Baker und Robert Shostak: *Gossips and Telephones.* Discrete Mathematics, 2:191–193, 1972.

[2] S. Bethke, M. Calvetti, H.F. Hoffmann, D. Jacobs, M. Kasemann und D. Linglin: *Report of the Steering Group of the LHC Computing Review.* Technischer Bericht, CERN European Origanization for Nuclear Research, February 2001.

[3] P. Braam, M. Callahan und Phil Schwan: *The InterMezzo Filesystem.* In: *Proceedings of the O'Reilly Perl Conference*, 1999.

[4] T. Bray, J. Paoli und C. Sperberg-MacQueen: *Extensible markup language (xml) 1.0 (w3c recommendation).* http://www.w3.org/TR/REC-xml, Februar 1998.

[5] M. Burrows und D. J. Wheeler: *A block-sorting lossless data compression algorithm.* Technischer Bericht 124, Digital Systems Research Center, 1994.

[6] David E. Culler und Jaswinder Singh: *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1999.

[7] Mauro Draoli, Gianfranco Mascari und Roberto Puccinelli: *DataGrid - Project Presentation*, 2001.

[8] A. Hajnal, E. Milner und E. Szemeredi: *A cure for the telephone disease.* Canad. Math. Bull., 15:447–450, 1972.

[9] S. Hedetniemi, S. Hedetniemi und A. Liestman: *A Survey of Gossiping and Broadcasting in Communication Networks.* Networks, 18:129–134, 1988.

[10] J. Howard: *Reconcile Users' Guide.* Technischer Bericht, Mitsubishi Electric Research Laboratory, 1999.

[11] J. Hromkovi, C. Klasing, B. Monien und R. Peine: *Dissemination of information in interconnection networks.* Combinatorial Network Theory, Seiten 125–212, 1995.

[12] The Institute of Electrical and Electronics Engineers Inc. (IEEE): *IEEE Standard for Information technology – Telecommunications and information*

exchange between systems – Local and metropolitan area networks – Specific requirements – Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, IEEE 802.3-2000, 2000.

[13] Nicholas Karonis: *Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance.* In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Seiten 377 – 384. IEEE, Mai 2000.

[14] Thilo Kielmann, Henri E. Bal, Sergei Gorlatch, Kees Verstoep und Rutger F.H. Hofman: *Network Performance-aware Collective Communication for Clustered Wide Area Systems.* Parallel Computing, 2001.

[15] W. Knödel: *New gossips and telephones.* Discrete Mathematics, 1975.

[16] David W. Krumme: *Reordered Gossip Schemes.* DMATH: Discrete Mathematics, 156, 1996.

[17] David W. Krumme, George Cybenko und K. N. Venkataraman: *Gossiping in Minimal Time.* SIAM J. Comput., 21(1):111–139, 1992.

[18] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard.* Technischer Bericht UT-CS-94-230, 1994.

[19] P. V. Mockapetris: *RFC 1034: Domain names — concepts and facilities*, November 1987.

[20] OMG: *Unified Modeling Language (UML), version 1.4*, 2001.

[21] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[22] *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 1).* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1993.

[23] J. Postel: *RFC 768: User Datagram Protocol*, August 1980.

[24] J. Postel: *RFC 791: Internet Protocol*, September 1981.

[25] J. Postel: *RFC 793: Transmission Control Protocol*, September 1981.

[26] R. L. Rivest: *RFC 1186: MD4 Message Digest Algorithm*, Oktober 1990. Status: INFORMATIONAL.

[27] R. L. Rivest: *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992. Status: INFORMATIONAL.

[28] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel und David C. Steere: *Coda: A Highly Available File System for a Distributed Workstation Environment.* IEEE Transactions on Computers, 39(4):447–459, 1990.

[29] Richard W. Stevens: *UNIX network programming.* Prentice-Hall, 1990.

[30] Bjarne Stroustrup: *Die C++ Programmiersprache.* Addison-Wesley, 2000.

[31] Sun Microsystems, Inc.: *RFC 1094: NFS: Network File System Protocol specification*, März 1989.

[32] L. Torvalds: *Linux Kernel.* http://www.kernel.org/.

[33] A. Tridgell: *Efficient Algorithms for Sorting and Synchronization.* Doktorarbeit, Australian National University, 1999.

[34] W3C DOM Working Group: *Document Object Model (DOM) Level 1 Specification*, 1998.