



TECHNISCHE UNIVERSITÄT ILMENAU

Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Verteilte System und Betriebssysteme

Hauptseminar

Skalierbarkeit in verteilten Systemen – Architekturen für verteilte Dateisysteme

Markus Brückner

Betreut durch Prof. Dr.-Ing. habil. Winfried Kühnhauser

14. Juli 2004

Inhaltsverzeichnis

1	Einleitung	3
1.1	Der Begriff <i>Skalierbarkeit</i>	3
1.2	Problemstellungen verteilter Dateisysteme	4
2	Das Coda Dateisystem	4
2.1	Grundstruktur	4
3	Ansätze zur Steigerung der Skalierbarkeit	7
3.1	Lastskalierbarkeit	7
3.1.1	Replikation	7
3.1.2	Caching	8
3.1.3	Callbacks	10
3.1.4	Hints	10
3.2	Administrative Skalierbarkeit	11
3.2.1	Zentrale Nutzerverwaltung	11
3.2.2	Gruppen/Rechtevererbung	13
3.2.3	Negative Rechte	14
4	Zusammenfassung	14
	Literatur	16

1 Einleitung

Die Entwicklung weitreichender und schneller Kommunikationsnetze ermöglicht heutzutage eine weltweite Zusammenarbeit von Menschen an gemeinsamen Projekten ohne Rücksicht auf ihren tatsächlichen Aufenthaltsort. Eine Grundvoraussetzung dafür ist ein einfacher und effizienter Zugriff auf gemeinsame Daten. Diese Arbeit soll einen der möglichen Ansätze zur Lösung dieses Problems näher beleuchten: die verteilten Dateisysteme.

Grundsätzlich existieren für den Zugriff auf gemeinsame Daten unterschiedliche Ansätze, angefangen von einfachem Austausch über E-Mail oder Webseiten bis hin zu komplexen Datenbanken mit starker Strukturierung der zu speichernden Informationen. Nachteil dieser Methoden ist ihre mangelnde Applikationstransparenz. Eine Anwendung, welche auf derartige Austauschmöglichkeiten zurückgreifen möchte, benötigt entweder weitreichendes Wissen über den Datenspeicher oder die Mithilfe des Nutzers.

Im allgemeinen Fall verwenden Applikationen das Dateisystem als persistenten Datenspeicher. Typischerweise als Baum von Verzeichnissen mit Dateien als Blättern organisiert, bietet es hinreichende Möglichkeiten, Datenbestände zu organisieren und zu strukturieren. Trotzdem treffen Dateisysteme normalerweise keinerlei Annahmen über die Feinstruktur von Daten, wie sie in den einzelnen Dateien vorliegt. Aufgrund dieser Vielfältigkeit sind Dateisysteme der Speicher der Wahl für nahezu alle Betriebssysteme und werden von einem überwiegenden Teil der vorhandenen Applikationen über standardisierte Schnittstellen verwendet. Ein applikationstransparenter Speicher für gemeinsame Daten muss daher seine Dienste über diese Schnittstellen anbieten. Diesen Anspruch erfüllen die verteilte Dateisysteme im Gegensatz zu vielen anderen Diensten.

Stellvertretend für die verschiedensten Ansätze für verteilte Dateisysteme wird in dieser Arbeit das Coda Dateisystem wie in [Sat90a] beschrieben stehen. Anhand dieses Beispiels werden Ansätze zur Skalierbarkeit über große Nutzerzahlen hinweg vorgestellt. Soweit der Umfang dieser Arbeit es zulässt, werden auch von Coda nicht verfolgte Ansätze vorgestellt.

1.1 Der Begriff *Skalierbarkeit*

Der Begriff *Skalierbarkeit* bezeichnet allgemein die Fähigkeit eines Systems über große Lastbereiche hinweg einen Dienst anzubieten, ohne dass grundsätzliche Änderungen an der Systemstruktur notwendig werden. Dabei sind grundsätzlich zwei Arten der Skalierbarkeit zu unterscheiden. Auf der einen Seite die *Lastskalierbarkeit*, die Fähigkeit über große Lastbereiche hinweg konstante Antwortzeiten, Durchsatzmengen und ähnliche Lastparameter aufzuweisen. Die *administrative Skalierbarkeit* andererseits bezeichnet die Fähigkeit, große Subjekt- und Objektmengen in einem System zu vereinigen ohne dieses dabei aufgrund der Komplexität unbedienbar zu machen. Speziell im Bereich der verteilten Dateisysteme mit typischerweise hohen Nutzerzahlen und Datenmengen kommt der administrativen Skalierbarkeit eine wichtige Rolle zu. Da für die in solchen Systemen abgelegten Daten dieselbe Vertraulichkeit und Sicherheit gefordert wird wie für lokale Datenträger, wird ein extrem komplexes System schnell zum Sicherheitsrisiko und sein Einsatz damit im großen Maßstab unmöglich. Diese Arbeit wird daher versuchen, beide Arten vom Skalierbarkeit gleichberechtigt nebeneinanderzustellen und für jede Prinzipien zur Steigerung aufzuzeigen.

1.2 Problemstellungen verteilter Dateisysteme

Abhängig von der Art des betrachteten Systems treten unterschiedliche Problemstellungen auf, welche zur Optimierung beachtet werden müssen. So muss beispielsweise bei einem DNS-Server, welcher in sehr kurzer Zeit Antworten liefern soll stark auf die Größe des verwendeten DNS-Teilbaumes geachtet werden, während die verfügbare Netzbandbreite eine eher untergeordnete Rolle spielt. Wird hingegen ein FTP-Server betrachtet, welcher sehr große Datenmengen pro Anfrage ausliefert, so fallen Suchzeiten und damit die Menge an verwalteten Daten weniger ins Gewicht. In diesem Fall wird die verfügbare Netzbandbreite viel eher zum Problem. Die verteilten Dateisysteme bieten hier eine umfangreiche Sammlung von Problemstellungen, die gesonderter Beachtung bedürfen. Als Hauptprobleme lassen sich folgende Aspekte ausmachen:

Nutzeranzahl Mit steigender Nutzerzahl steigt sowohl die Last im System durch parallele Zugriffe als auch der Aufwand zur Administration des Systems (speziell bei der Nutzerverwaltung).

Datenmenge Größere Datenmengen, sowohl in einzelnen Objekten als auch im System insgesamt führen zu einer erhöhten Netzwerklast. Die Menge der Daten im System korreliert oft mit der Nutzeranzahl, so dass hier beide Faktoren zusammenwirken.

Zugriffsverhalten Abhängig vom Nutzungsprofil (wenige langfristige parallele Zugriffe oder viele kurze Operationen) bieten sich unterschiedliche Strategien zur Lastverteilung an. Ein verteiltes Dateisystem sollte die Möglichkeit bieten, detaillierte Anpassungen an das Nutzungsprofil einzelner Objekte vorzunehmen.

verfügbare Netzbandbreite und -latenz Bei sinkender Bandbreite und/oder steigender Latenz werden Maßnahmen nötig, die die Performanz des Dateisystems erhalten.

Es ist erkennbar, dass haben diese Problemstellungen sowohl Einfluss auf die Last-, als auch auf die administrative Skalierbarkeit haben. Während Lastskalierbarkeit als offensichtliches Ziel der Optimierung eines Systems erscheint, offenbart sich eine fehlende administrative Skalierbarkeit erst bei sehr großen Nutzerzahlen und Datenmengen. Aus diesem Grund wird diese Arbeit auch auf Möglichkeiten zu Behebung dieser Probleme eingehen.

2 Das Coda Dateisystem

Das Coda Dateisystem wurde an der Carnegie Mellon University als Nachfolger des Andrew File System (AFS) [SHN⁺85] entwickelt. Bereits AFS war auf den Betrieb mit mehreren tausend Nutzern und großen Datenmengen eingerichtet. Bei der Weiterentwicklung zu Coda lag das Hauptaugenmerk auf der Verfügbarkeit der Daten und weniger auf einer Leistungssteigerung. Im folgenden wird der Grundaufbau des Systems kurz vorgestellt, um später auf Details unter dem Aspekt Skalierbarkeit näher eingehen zu können.

2.1 Grundstruktur

Die Struktur von Coda orientiert sich sehr stark an seinem direkten Vorgänger AFS. Tatsächlich teilen sich sogar die Implementierungen von AFS und Coda der Carnegie Mellon University große Teile ihres Quellcodes. Zentrales Element bei Coda ist ein lokaler persistenter Cache, welcher auf Dateiebene arbeitet. Öffnet ein Nutzer eine Datei, so wird diese in den Cache übertragen

und fortan als Arbeitskopie verwendet. Ein Konsistenzprotokoll stellt dabei sicher, dass die Datei gleich der Version auf dem Server bleibt. Abweichend von der UNIX-Dateisystemsemantik *as soon as possible*, welche jegliche Änderung auf eine geöffnete Datei sofort für andere Nutzer sichtbar macht, arbeitet Coda sessionorientiert. Die Änderungen werden somit erst beim Schließen einer Datei sichtbar.

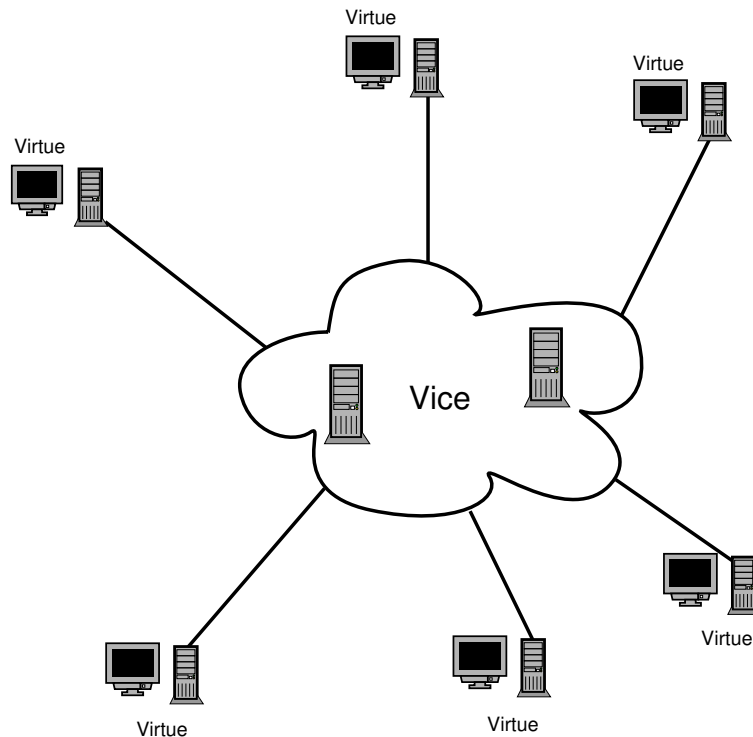


Abbildung 1: Die Grundstruktur eines Coda-Systems. Das Vice fasst die Coda-Server zusammen, welche per Definition vertrauenswürdig sind. Als Virtue werden die Clientrechner bezeichnet, die sich untereinander nicht vertrauen und denen auch vom Vice nicht vertraut wird, solange sich ein Client nicht authentisiert hat.

Die in Abbildung 1 gezeigte Struktur einer Coda-Installation lässt zwei große Teile erkennen. Auf der einen Seite das sogenannte *Vice*, die Sammlung aller Coda-Server. Diese vertrauen sich untereinander per Definition, was großen Einfluss auf das Authentisierungsverfahren in einem Vice hat. Als zweiter Teil existieren die sogenannten *Virtue*, die nicht vertrauenswürdigen Clients. Diese vertrauen sich weder untereinander noch wird ihnen vom Vice vertraut.

Bereits den Entwicklern von AFS war bekannt, dass für eine transparente Einbindung eines verteilten Dateisystems in eine UNIX-Umgebung Änderungen am Betriebssystemkern notwendig sind. So verwenden zwar die meisten Anwendungsprogramme zum Dateisystemzugriff Schnittstellen, welche ihnen von der C-Bibliothek des Systems zur Verfügung gestellt werden. Ein Austausch dieser Bibliothek ermöglicht daher die notwendigen Änderungen für den Zugriff auf ein verteiltes Dateisystem. Dazu ist jedoch mindestens ein erneutes Linken der Anwendungsprogramme notwendig, was oft nicht möglich ist. Die einfachste Möglichkeit ist daher, den Betriebssystemkern um die notwendigen Fähigkeiten zu erweitern. Leider unterscheiden sich bereits in der UNIX-Welt – in der AFS und Coda ursprünglich entwickelt wurden – die Kerne so stark, dass eine Portierbar-

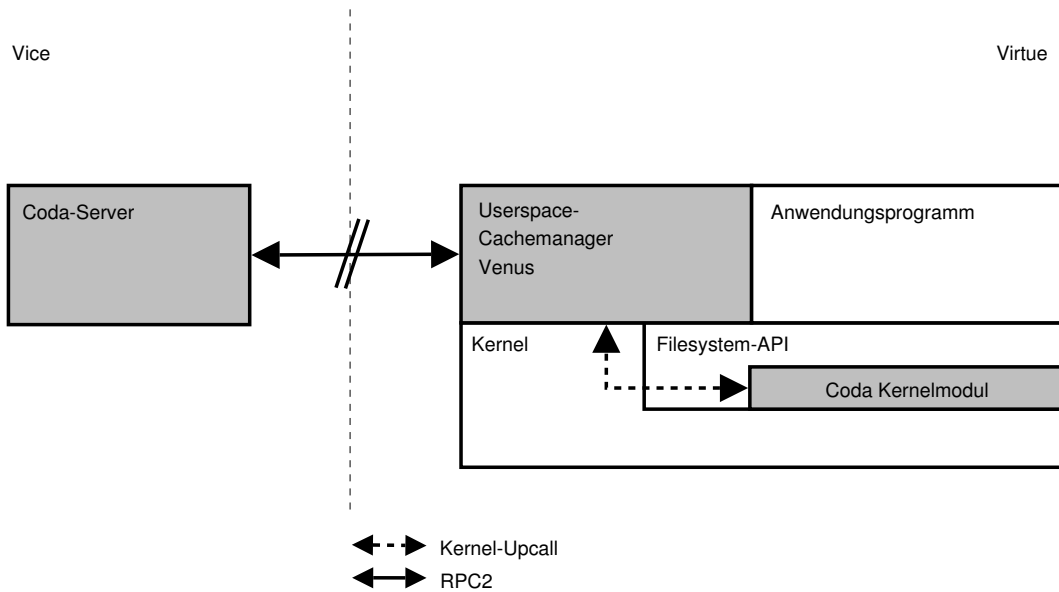


Abbildung 2: Struktur von Vice und Virtue. Der Coda Server im Vice ist ein reines Userspaceprogramm, wohingegen zur Emulation der UNIX-Dateisystemsemantik auf dem Client ein Modul im Betriebssystemkern notwendig ist. Aufgrund der teilweise recht komplexen Kommunikationsstruktur und Cacheverwaltung werden nahezu alle Teile von Coda in einem Cachemanager implementiert. Mit diesem kommuniziert das Kernelmodul über Upcalls. Somit vermeidet man übermäßige, schwer zu portierende Änderungen am Betriebssystemkern und lagert diese in den Userspace aus, wo sie im allgemeinen leichter portierbar sind.

keit größerer Codeteile über verschiedene Systeme hinweg schwierig bis unmöglich erscheint. Aus diesem Grund wurde der Clientcode in zwei Teile aufgeteilt, wie in Abbildung 2 erkennbar ist. Der im Kern verankerte Code übernimmt die Implementation der Dateischnittstelle und den Zugriff auf den lokale Cache, während der im Userspace laufende Cachemanager die weitaus komplexere Aufgabe der Kommunikation mit dem Server übernimmt. Die Trennung von Cachemanager und Kernelmodul erschien den Entwicklern notwendig um zu große, schwer portierbare Änderungen am Betriebssystemkern zu vermeiden.

Coda organisiert die verwalteten Daten in sogenannten *Volumes*. Ein Volume wird vom Cachemanager auf dem Client als kompletter Unterbaum des Coda-Namensraumes präsentiert. Es ist dabei in etwa vergleichbar mit einer Partition auf einer Festplatte: ein Speicher fester Größe, welcher transparent in den Verzeichnisbaum eingeblendet wird. Volumes können über mehrere Server verteilt vorliegen, welche dann eine sogenannte *Volume Storage Group (VSG)* bilden. Die für den Client aktuell sichtbaren Rechner aus einer VSG bezeichnet man als *accessible VSG*. Dieses Konzept ist notwendig, um dem Client eine Reaktion auf Fehler im Netz, die zur Nichterreichbarkeit einzelner Server führen, zu ermöglichen. Für den Nutzer herrscht beim Datenzugriff vollständige Ortstransparenz, da die Pfadangabe der im Volume vorliegenden Daten keinerlei Informationen über den tatsächlichen Speicherort enthält. So ist eine einfache Migration von Daten zwischen den einzelnen Server möglich ohne Änderungen an der Clientkonfiguration vorzunehmen.

Eine ungewöhnliche Eigenschaft in der Welt der verteilten Dateisysteme ist die Möglichkeit, ohne Netzverbindung zu arbeiten. Fällt die Anzahl der Server in der *accessible VSG* eines Volumes auf 0, so geht der Cachemanager in den sogenannten *Disconnected Operation* Modus, in dem der

Nutzer nur auf den bereits im lokalen Cache vorhandenen Dateien arbeiten kann. Der Cachemanager zeichnet sämtliche Änderungen auf diesen Daten auf und spielt sie bei Wiederaufnahme der Netzverbindung auf den oder die Server zurück. Für die Applikationen sind diese Vorgänge vollkommen transparent; spezielle Vorkehrungen sind nicht notwendig. Entstanden ist diese Fähigkeit aus der Erkenntnis, dass mit steigender Nutzerzahl und Datenmenge die Abhängigkeit vom einem verteilten Dateisystem wächst. Im Falle eines teilweisen oder kompletten Ausfalles des Systems sind so möglicherweise komplette Organisationen nicht arbeitsfähig. Der *Disconnected Operation* Modus ermöglicht eine sanfte Reaktion auf derartige Ausfälle und erlaubt so ein weiteres Wachstum des Systems.

3 Ansätze zur Steigerung der Skalierbarkeit

Im folgenden Abschnitt sollen verschiedene Ansätze zur Steigerung der Skalierbarkeit in verteilten Dateisystemen diskutiert werden. Dabei soll sowohl auf die Last- als auch auf die administrative Skalierbarkeit eingegangen werden, da Systeme mit mehreren tausend Nutzern ohne beide nicht möglich sind. Soweit vorhanden, werden Implementierungen in realen Systemen kurz vorgestellt, wobei hier hauptsächlich auf Coda eingegangen wird.

3.1 Lastskalierbarkeit

Die im folgenden vorgestellten Ansätze dienen hauptsächlich der Steigerung der Lastskalierbarkeit eines Systems. In verteilten Dateisystemen hauptsächlich auftretende Lastprobleme betreffen meist die Systemlast der Server, eher selten die des Clients. Aus diesem Grund formulierte der Autor von [Sat90b] als einen der Grundsätze zum Entwurf von Coda und AFS: “Workstations have cycles to burn”. Er stellt als Ergebnis seiner Arbeit an den beiden Systemen fest, dass grundsätzlich sämtliche rechenintensiven und nicht sicherheitsrelevanten Aufgaben an den Client übergeben werden sollten, um den Server weitestmöglich zu entlasten. Die im folgenden vorgestellten Ansätze konzentrieren sich daher auch auf die Entlastung des Servers und lassen die Clientlast größtenteils außer Acht.

3.1.1 Replikation

Ein effizienter Ansatz zur Steigerung der Skalierbarkeit und Verfügbarkeit eines Dienstes ist dessen Replikation. Darunter versteht man das mehrfache Vorhalten eines identischen Dienstangebots – im Fall verteilter Dateisysteme mehrfache identische Kopien der Daten – und die Verteilung von Anfragen auf die einzelnen Repliken. Als vorteilhaft erweist sich hier die nahezu lineare Skalierbarkeit des Gesamtsystems über die Anzahl der parallel vorhandenen Knoten. Effiziente Verteilbarkeit der Anfragen vorausgesetzt, lässt sich auf allen Knoten eine gleichmäßige Lastverteilung bis hin zur optimalen Ausnutzung der vorhandenen Ressourcen erreichen.

Da verteilte Dateisysteme ständigen Änderungen unterworfen sind, ist eine Strategie zur Synchronisation mehrerer Repliken notwendig. Ein gängiger Ansatz hierzu ist die Verwendung mehrerer Nur-Lese-Kopien und einer schreibfähigen Replik. Aufgrund der baumartigen Synchronisationsstrategie mit der schreibfähigen Masterkopie als Wurzel und den Nur-Lese-Kopien als Clients ist dieser Ansatz leicht zu implementieren und findet daher häufig Verwendung. Nachteilig ist dabei die Tatsache, dass die schreibfähige Kopie als Flaschenhals bleibt.

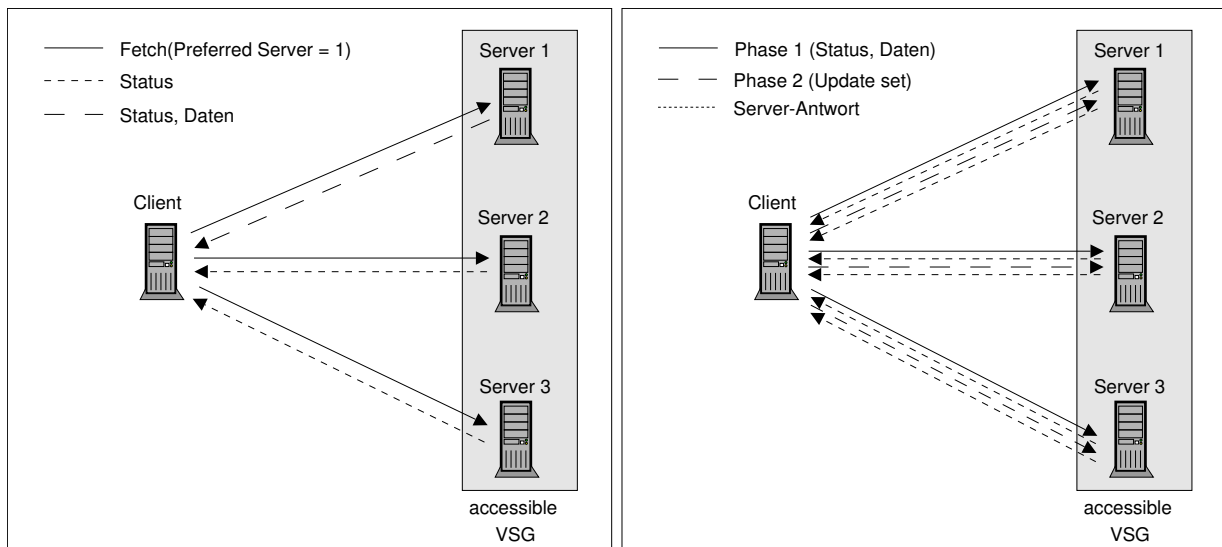


Abbildung 3: Les- und Schreibzugriff auf eine Datei in einem Coda-Volumen (Leszugriff mit cache miss). Im Falle eines Lesezugriffs erhält der Client auf Anfrage an alle Repliken von dem vorher durch ihn festgelegten primären Server eine Statusantwort und die Daten, von allen anderen nur einen Status. Mit Hilfe der im Status vorliegenden Versionsinformationen ist es möglich, die Repliken auf Konsistenz zu prüfen und gegebenenfalls eine Synchronisation anzustoßen. Bei einem Schreibzugriff schickt der Client die zu schreibenden Daten in Form eines update set an sämtliche Repliken. Sollten einige durch eine Spaltung des Netzes nicht erreichbar sein, so werden die daraus entstehenden Inkonsistenzen beim nächsten Lesezugriff entdeckt und behoben.

Coda implementiert Replikationsalgorithmen in Form mehrerer schreibfähiger Kopien. Die Replikation geschieht dabei auf Volumeebene. Wie in Abbildung 3 gezeigt liegt sowohl die Synchronisation der Repliken als auch die Entdeckung von Inkonsistenzen beim Client. Er schickt bei einem Lesezugriff unter Angabe eines bevorzugten Servers eine Anfrage an sämtliche Server seiner accessible VSG. Diese antworten mit einer Statusantwort, welche unter anderem Versionsinformationen zum angefragten Objekt enthält. Zusätzlich antwortet der bevorzugte Server mit den eigentlich gewünschten Daten. Entdeckt der Client eine Inkonsistenz in den empfangenen Versionsinformationen, so fordert er die Server auf, ihre Kopien wieder in Einklang zu bringen. Im Falle eines Schreibzugriffs sendet der Client die zu schreibenden Daten an alle in der accessible VSG vorhandenen Repliken. Unter den Servern findet im Normalfall keinerlei Kommunikation zu Synchronisation statt. Entstehen durch temporäre Nichterreichbarkeit einzelner Server Inkonsistenzen, so werden diese beim nächsten Lesezugriff auf das betreffende Objekt entdeckt und korrigiert. Diese auf den ersten Blick ungewöhnliche Synchronisationsstrategie folgt der in [Sat92] aufgestellten These, der Client habe im Gegensatz zum Server Rechenzeit übrig, welche er für aufwändige, nicht sicherheitsrelevante Aufgaben einsetzen kann.

3.1.2 Caching

Ein Cache ist ein lokaler, transparenter und schneller Zwischenspeicher, welcher eingesetzt wird um wiederkehrende Zugriffe auf Objekte stark zu beschleunigen. Seinen Vorteil zieht er aus der Nähe zum Verarbeitungsort der Daten, was eine Anbindung mit hoher Bandbreite und geringen

Latenzen ermöglicht. Aufgrund der hohen Anforderungen an die Zugriffsgeschwindigkeit ist ein Cache im allgemeinen relativ teuer und klein im Verhältnis zu anderen Speichern. Daher wird er nur als temporärer Speicher für oft benötigte Daten verwendet.

Problematische Effekte ergeben sich durch mögliche Inkonsistenzen zwischen Daten im Cache und im persistenten Speicher. So ist es beispielsweise möglich, dass ein paralleler Schreibzugriff eines Nutzers die Daten im persistenten Speicher geändert werden und ein darauf folgender Lesezugriff eines anderen Nutzers, welcher aus dessen Cache bedient wird, veraltete Daten liefert. Aus diesem Grund sind komplexe Konsistenzhaltungsalgorithmen für den effektiven Einsatz eines Caches notwendig. Hierbei ist im Kontext der verteilten Dateisysteme eine Abwägung zu treffen zwischen der durch häufige Konsistenzprüfung erzeugten Netzlast und den durch seltene Prüfung verursachten Inkonsistenzen zwischen Cache und persistentem Speicher.

Caching spielt für die Skalierbarkeit von Coda eine zentrale Rolle. Im Gegensatz zu vielen anderen Implementierungen findet hier ein persistenter Cache in Form der lokalen Festplatte des Clients Verwendung. Das Caching wird auf Dateiebene betrieben, das heißt beim Öffnen einer Datei wird diese komplett in den Cache übertragen, sämtliche Änderungen werden auf dieser lokalen Kopie ausgeführt und beim Schließen werden diese auf den Server geschrieben. Hier wird deutlich, dass Coda von der normalen Semantik der UNIX-Dateisysteme abweicht. Statt *as soon as possible*, das heißt dem sofortigen Sichtbarwerden von Änderungen für andere Nutzer beim Schreiben, findet hier eine Session-Semantik Verwendung. Als Session kann dabei der Zeitraum vom Öffnen bis zum Schließen einer Datei betrachtet werden. Alle Schreibzugriffe, welche dazwischen stattfinden, sind für andere Nutzer nicht von Bedeutung und werden erst beim Schließen sichtbar. Diese Vorgehensweise führt je nach Nutzungsmuster zu einer erheblich sinkenden Server- und Netzlast, sowie geringeren Antwortzeiten und höheren Schreibraten durch weniger Kommunikation zwischen Server und Client. Als negativ ist die Begrenzung der bearbeitbaren Dateigröße auf die Größe des Caches aufgrund des Caching auf Dateiebene zu werten. Der typische Einsatzfall von Coda als Arbeitsverzeichnis für einzelne Nutzer oder Lagerplatz der Systemprogramme lässt empfohlene Cachegrößen im Bereich bis etwa 300 MB als ausreichend erscheinen. Spezialfälle wie die Bearbeitung von Multimediadaten mit typischerweise großen Datenmengen werden so allerdings unmöglich gemacht. Ein weiteres Problem, welches durch die Aufweichung der UNIX-Dateisystemsemantik entsteht, ist die fehlende Möglichkeit, Änderungen bei geöffneten Dateien persistent und systemweit sichtbar zu machen. Diese Fähigkeit ist für die Transaktionssemantik der meisten Datenbanksysteme zwingend notwendig. Laut [HKM⁺88] Abschnitt 5 wurde diese Fähigkeit allerdings bewußt ausgeschlossen, da nach Ansicht des Autors eine effiziente Implementierung des Systems damit nicht möglich wäre.

Ein Spezialfall des Caching findet bei Coda in Form des *Disconnected Operation* Modus Anwendung. Fällt die Anzahl der Server in der accessible VSG eines Clients auf 0, so beginnt der Cachemanager, Anfragen nur noch aus dem lokalen Cache zu bedienen und sämtliche Änderungen in einem Logfile aufzuzeichnen. Ist der Zustand der Netztrennung beendet, so werden die aufgezeichneten Vorgänge auf die Servern übertragen. Diese Fähigkeit erlaubt dem Nutzer eine erhöhte Mobilität und schafft Reaktionsmöglichkeiten auf Ausfälle. Laut [Sat92] war die gesteigerte Zahl an Ausfällen in einem großen verteilten Dateisystem ein Grund für die Entwicklung des Disconnected Operation Modus. So sollte die Arbeitsfähigkeit des Systems erhalten werden, auch wenn es zu Netztrennungen oder Serverausfällen kommt.

Einen anderen Ansatz des Caching implementiert das Sprite Network File System [NWO88]. Hier ist der Cache im Arbeitsspeicher des Clients abgelegt. Außerdem arbeitet das System auf Blockebene. Es werden nur einzelne Blöcke einer Datei im Cache vorgehalten, was prinzipiell die

Verarbeitung von Dateien größer als der Cache erlaubt. Sprite emuliert die UNIX-Dateisystemsemantik, indem es für Dateien, welche zum Schreiben geöffnet werden, das Caching verbietet und so immer die neuesten Daten zur Verfügung stehen. Um die Festplattenbenutzung auf dem Server zu reduzieren werden dabei ankommene Schreibaufträge um 30 bis 60 Sekunde verzögert und mit folgenden Aufrufen gebündelt. So würden sich beispielsweise ein Schreiben auf eine Datei und deren anschließendes Löschen aufheben und keinerlei Zugriff auf die Platte erfolgen. Des Weiteren bietet Sprite die Möglichkeit zum sequentiellen Schreiben, wobei immer nur ein Client eine Datei zum Schreibzugriff öffnen kann. Zusammengefasst bietet das Sprite Network File System eine wesentlich engere Emulation der UNIX-Dateisystemsemantik, bezahlt dies jedoch mit einem teilweise gesteigerten Leistungsbedarf.

3.1.3 Callbacks

Ein Mechanismus zur Wahrung der Konsistenz zweier Kopien eines Datums sind sogenannte *Callbacks*. Ein Callback ist das Versprechen eines Partners über die Unveränderlichkeit von herausgegebenen Daten. Wird das Original geändert, so wird dieses Versprechen gebrochen und es erfolgt eine Mitteilung an den empfangenden Partner. Dieser kann dann geeignet auf diese Information reagieren. Vorteil eines Callbacks ist die durch ihn stark sinkende Netz- und Serverlast. Prinzipiell ist während der gesamten Gültigkeitsdauer eines Callbacks keinerlei Kommunikation zwischen Server und Client notwendig, da der Client die Daten solange als unverändert annehmen kann, bis er über das Gegenteil informiert wird. Nachteilig wirkt sich der gesteigerte Speicherbedarf auf dem Server aus. Dieser muss nun für jedes von einem Client geöffnete Objekt einen Status über alle ausgegebenen Callbacks halten.

Coda verwendet das Callbackkonzept ebenso wie sein Vorgänger AFS. Ursprünglich waren bei AFS statuslose Server vorgesehen. Wie jedoch in [SHN⁺85] ausgeführt, erwies sich dieser Ansatz in der Prototypenphase als problematisch. Das eingesetzte Protokoll zur Wahrung der Cachekonsistenz sah vor, dass sich die Clients in regelmäßigen Abständen beim Server über den Status ihrer Daten informieren. Dadurch konnte der Server statuslos gehalten und sämtliche Arbeit auf den Client ausgelagert werden. Bei einer höheren Anzahl an Clients erwies sich die Anzahl an Statusanfragen für den Server als Problem. Dieser verbrachte einen Großteil der Zeit damit Statusanfragen zu beantworten und geriet somit an seine CPU-Leistungsgrenzen. Aus der Beobachtung, dass viele dieser Statusanfragen unnötig waren – speziell die in AFS gelagerten Systemprogramme änderten sich so gut wie niemals – leiteten die Autoren die Verwendung eines Callbackmechanismus ab. In AFS Version 2 erstmals implementiert erwies sich dieser als Kernpunkt für die Skalierbarkeit des Systems und wurde von Coda übernommen. Der Coda-Server führt hierbei über jede geöffnete Datei jedes Clients einen Status und informiert alle relevanten Clients vor dem Ändern einer Datei auf dem Server. Diese können dann geeignete Maßnahmen ergreifen um ihre Cachekonsistenz sicherzustellen.

3.1.4 Hints

Eine weitere Möglichkeit zur Senkung der Kommunikationslast zwischen Client und Server ist die Verwendung von *Hints*. Als Hint kann eine Information betrachtet werden, welche bei Korrektheit eine starke Beschleunigung eines Prozesses bewirkt, bei Fehlerhaftigkeit aber keine falschen Ergebnisse produziert. Prinzipiell muss diese Information also selbstvalidierend bei Verwendung sein. Dies ist der größte Nachteil von Hints: im Allgemeinen sind nur sehr wenige der im System

vorhandenen Informationen als Hints geeignet. In einem verteilten Dateisystem sind beispielsweise die Inhalte der Dateien nicht als Hints verwendbar, da diese nicht automatisch auf Korrektheit geprüft werden können. In Coda werden die Einträge aus der *Volume Location Database* als Hints betrachtet. Diese Datenbank enthält die tatsächlichen Speicherorte zu den einzelnen Dateien im Coda-Namensraum. Die Zuordnung zwischen dem Pfad im Namensraum und dem tatsächlichen Speicherort kann als Hint benutzt werden, da bei Richtigkeit der Zugriff auf die Daten ohne Rückfrage bei der Volume Location Database erfolgen kann, bei Fehlerhaftigkeit aber die Datei einfach nicht gefunden wird, was dann zur erneuten Auflösung des Speicherortes führt. Somit ist auch im Fehlerfall mit einer leichten Verzögerung ein korrektes Ergebnis gesichert. Wichtig ist hierbei das überwiegend statische Verhalten eines Volumes. Die tatsächlichen Speicherorte der Dateien ändern sich äußerst selten, so dass die Links in den meisten Fällen korrekt sind, was eine starke Senkung der Anfragelast auf die Volume Location Database zur Folge hat. Eine weitere Information, welche von Coda im weitesten Sinne als Hint betrachtet wird, ist die Annahme der korrekten Synchronisation der einzelnen Repliken einer Datei. Diese Annahme wird als gültig betrachtet, solange die betreffenden Server nicht durch einen Client über Inkonsistenzen in den Versionsinformationen informiert werden. Damit entfällt ein aufwändiger regelmäßiger Abgleich der Datenbestände der einzelnen Server, was bei großen Datenmengen sowohl Netz- als auch Serverlast erheblich senkt.

3.2 Administrative Skalierbarkeit

Wie bereits erwähnt ist die Sicherheit eines verteilten Dateisystems von elementarer Bedeutung für Verwendung als zentraler Datenspeicher einer Organisation. Voraussetzung für die Sicherheit eines Systems ist eine effektive Administration des Systems. Speziell im Bereich Nutzer- und Rechteverwaltung ergeben sich hier bei verteilten Dateisystemen mit ihren typischerweise großen Subjekt- und Objektzahlen Probleme, welche sich nur durch weitreichende Unterstützung durch das System lösen lassen. In den folgenden Abschnitten werden daher einige Ansätze zur Steigerung der administrativen Skalierbarkeit vorgestellt.

3.2.1 Zentrale Nutzerverwaltung

Ein wichtiger Aspekt zur Sicherung eines Systemes ist eine effiziente Nutzerverwaltung. Ein Administrator muss jederzeit in der Lage sein, zu überblicken, welche Personen wie auf ein System zugreifen können, um sicherzustellen, dass alle Zugriffe der aktuell gültigen Sicherheitspolitik folgen. In einem Dateisystem, welches sich über viele Server verteilt, ist es unmöglich, eine dezentralisierte Nutzerverwaltung einzurichten. Die normalerweise gewünschte Dezentralisierung eines für die Arbeit des Systems wichtigen Dienstes führt hier schnell zu administrativen Problemen, welche eine einfache Administration unmöglich machen. Ein Beispiel hierfür wären Inkonsistenzen in verschiedenen Nutzerdatenbanken eines Systems, welche eine Nutzung bestimmter Systemteile für einzelne Personen unmöglich machen. Aus diesem Grund ist eine zentrale Nutzerverwaltung prinzipiell vorzuziehen. Zu beachten ist dabei jedoch, dass der Begriff "zentrale Organisation" hier eine einheitliche Schnittstelle für Administratoren bezeichnet. Eine Verteilung der Administrationsarbeit auf verschiedene Untereinheiten einer Organisation – welche sich beispielsweise an der Struktur einer Organisation orientiert – trägt zur Skalierbarkeit der gesamten Organisationsstruktur außerhalb des Systems bei und ist daher auch als notwendig zu betrachten. Jedoch darf diese Aufteilung nicht zu gesteigertem Aufwand für die Erledigung einzelner Administrationsaufgaben führen, da dies im Allgemeinen eine höhere Fehlerrate zur Folge hat.

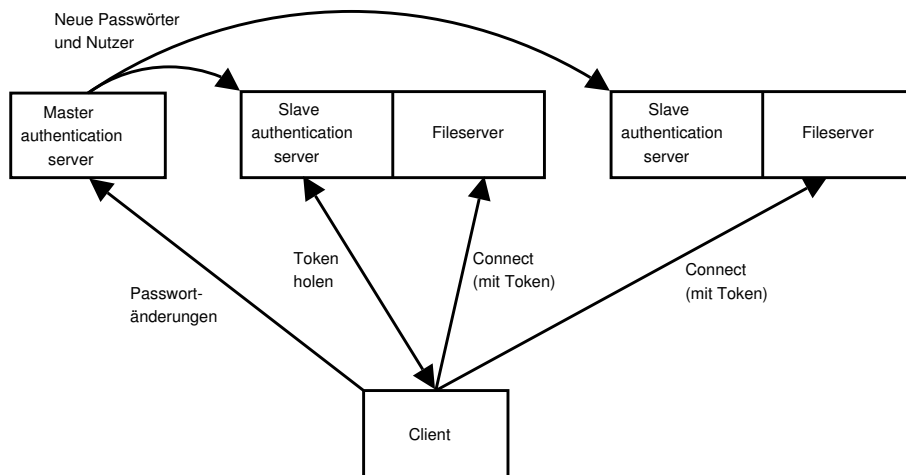


Abbildung 4: Struktur der Coda-Nutzerverwaltung. Die zentrale Verwaltung aller Nutzer auf einem Master Authentication Server ermöglicht einem Systemadministrator einen schnellen Überblick über die im System vorhandenen Nutzer und ihre Eigenschaften. Um diesen zentralen Dienst nicht zum Flaschenhals für große Installationen zu machen, wird die Nutzerdatenbank auf die Slave Authentication Server repliziert. Jeweils die Kombination aus Slave Authentication Server und Fileserver bildet einen Knoten im Vice. Ein Nutzer meldet sich an einem beliebigen Slave Authentication Server an und bekommt von diesem einen zeitlich begrenzt gültigen Token ausgestellt, mit dem er sich auf allen Servern des entsprechenden Volumes frei bewegen kann, ohne erneut eine Authentisierung durchzuführen. Diese dezentrale Struktur ermöglicht eine Arbeit des Systems auch dann, wenn der Master Authentication Server nicht erreichbar ist.

Coda versucht die Vorteile von zentraler und dezentraler Nutzerverwaltung zu vereinigen. Die Verbindung aus der einfachen Administration einer zentralen Datenbank und der Lastverteilung und Ausfallsicherheit eines dezentralen Authentisierungsdienstes ermöglicht dem System eine effiziente Verwaltung auch größter Nutzerzahlen. Abbildung 4 zeigt die Struktur des von Coda verwendeten Systems. Einerseits geschieht die zentrale Nutzerverwaltung auf einem sogenannten *Master Authentication Server (MAS)*, welcher einen kompletten Überblick über die im System vorhandenen Nutzer bietet. Andererseits wird mit Hilfe der Replikation der Nutzerdatenbank auf die *Slave Authentication Server (SAS)* ein Ausfall des Gesamtsystems bei Ausfall des MAS vermieden. Tatsächlich ist der MAS für die Arbeit des System nicht notwendig, da jeder Codaserver über einen eigenen SAS verfügt, welcher die tatsächliche Authentisierung eines Nutzers übernimmt. Dieser meldet sich einmal an einem beliebigen SAS an und bekommt einen zeitlich begrenzten Token ausgestellt, mit welchem er sich an allen Fileservern dieses Volumes anmelden kann ohne erneut Nutzernamen und Passwort zu übermitteln. Hier kommt die Eigenschaft von Coda zum Tragen, nach der die Server im Vice per Definition vertrauenswürdig sind: Eine erneute Anmeldung an jedem Fileserver ist unnötig, da der ursprüngliche SAS mittels des Tokens die Echtheit des Clients bestätigt.

Die Fähigkeit des MAS Änderungen an der Nutzerdatenbank vorzunehmen macht ihn zu einem zentralen Ziel für Angriffe. Ist der Schutz des MAS gebrochen, so ist das gesamte System unter der Kontrolle des Angreifers. Da allerdings eine beliebige Kommunikation mit dem MAS nicht notwendig ist, sondern dieser immer nur mit bestimmten Servern (den SAS) kommuniziert und diese bekannt sind, sind besondere Schutzmaßnahmen, wie Paketfilter problemlos implementierbar.

Ein Verbindungsaufbau von außerhalb des Vice ist unnötig und kann so abgewiesen werden. Da die Kommunikation zwischen MAS und SAS verschlüsselt erfolgt, kann auch hier von der Sicherheit der Nutzerinformationen ausgegangen werden.

Insgesamt ergibt sich so ein sehr (ausfall-)sicheres Authentisierungssystem, welches trotzdem die Vorteile einer zentralen Nutzerverwaltung für die Administratoren bietet. Um eine Verteilung der Administration auf verschiedene Organisationseinheiten zu ermöglichen, ist die gesamte Authentisierungsstruktur jeweils zu einem Volume gehörig. Somit ist durch Anlegen eines neuen Volumes innerhalb des Systems und damit die Verfügbarkeit eines eigenen MAS eine einfache Möglichkeit zur Administration einer eigenen Nutzermenge für Unterstrukturen einer Organisation geschaffen.

3.2.2 Gruppen/Rechtevererbung

Ein typisches Problem, welches verteilte Dateisysteme betrifft, ist ihre große Nutzerzahl. Trotz ausgefeilter Techniken zur zentralen Nutzerverwaltung wird es ab einer bestimmten Anzahl an Nutzern schwierig, den globalen Überblick über die Berechtigungen einzelner Nutzer zu behalten. Ein vergleichbares Problem führt in großen Firmen zur Einführung einer festen, stark hierarchisch gegliederten Organisationsstruktur mit Abteilungen, Unterabteilungen, Arbeitsgruppen etc. Diese Gliederung ermöglicht die globale Betrachtung einzelner Teilbereiche eines Unternehmens ohne durch zuviele Details unüberschaubar zu werden.

Inspiziert von diesen Strukturierungsprinzipien führen viele Dateisysteme den Begriff der *Gruppe* ein. Eine Gruppe ist eine Zusammenfassung verschiedener Nutzer mit denselben Rechten. Ihr können als ganzes Rechte erteilt oder entzogen werden ohne ihre einzelnen Mitglieder zu modifizieren. Um mit Hilfe von Gruppen die hierarchische Struktur einer Organisation abbilden zu können, bedarf es der Möglichkeit, sie wiederum zu größeren Gruppen zusammenzufassen (analog der Zusammenfassung von mehreren Unterabteilungen zu einer Abteilung). Grundlage für diese Fähigkeit bildet das Prinzip der *Rechtevererbung*. Hierbei erhält das Mitglied einer Gruppe – sei es eine weitere Gruppe oder ein Nutzer – sämtliche Rechte zugeteilt, die dieser Gruppe selbst zustehen. Hat also eine Gruppe das Leserecht auf ein bestimmtes Objekt, so haben auch alle ihre Mitglieder dieses Recht. Innerhalb der Bäume, die durch verschachtelte Gruppen aufgespannt werden akkumulieren sich die vergebenen Rechte in den Blättern (\cong Nutzern). Ein Nutzer erhält so alle Rechte aller Gruppen, in denen er direkt oder indirekt (über die Mitgliedschaft in weiteren Gruppen) Mitglied ist. Dieses Prinzip ermöglicht auf einfache Art und Weise die Festlegung von Berechtigungen für große Nutzerzahlen. Folgendes Beispiel soll zur Verdeutlichung dienen: In den Systemen einer Universität existiert eine Gruppe *Mitarbeiter*, welche alle Angestellten enthält. Zusätzlich existiert eine Gruppe *Fachgebietsleiter*, welche aus den Leitern sämtlicher Fachgebiete an der Universität besteht. Diese Gruppe ist ihrerseits Mitglied in der Gruppe *Mitarbeiter*. Im System existiert eine zentrale Übersicht über sämtliche Diplomarbeitsthemen, welche aktuell an der Universität vergeben werden können. Mitglieder der Gruppe *Mitarbeiter* haben Lesezugriff auf dieses Objekt, Mitglieder der Gruppe *Fachgebietsleiter* Schreibzugriff. Möchte Nutzer K, welcher Mitglied in der Gruppe *Fachgebietsleiter* ist, auf dieses Objekt zugreifen, so wird er einerseits das Leserecht (über die Vererbungskette $Mitarbeiter \leftarrow Fachgebietsleiter \leftarrow K$) und andererseits das Schreibrecht (über die Kette $Fachgebietsleiter \leftarrow K$) erhalten. Auf diese Weise lässt sich die Organisationsstruktur einer Einrichtung im System relativ einfach nachbauen. Dieses Gruppenkonzept wird von Coda ebenso wie von vielen anderen Dateisystemen implementiert. Lokale Dateisysteme implementieren dabei eine Vereinfachung, welche die Mitgliedschaft von Gruppen in anderen Gruppen nicht ermöglicht.

3.2.3 Negative Rechte

Durch das in Abschnitt 3.2.2 vorgestellte Gruppenkonzept haben zwangsläufig größere Personengruppen gleiche Rechte in einem System. In bestimmten Situationen kann dieses zu Problemen führen, wenn sich ein Mitglied einer Gruppe als nicht vertrauenswürdig erweist. In [Sat89] Abschnitt 6.1 beschreibt der Autor die auf diesen Fall zugeschnittenen *negativen Rechte*. Diese erlauben den schnellen und selektiven Entzug von Berechtigungen eines Subjektes auf ein Objekt. Hierbei werden die als negativ aufgeführten Rechte von der Summe der durch Gruppenmitgliedschaft ererbten Rechte abgezogen und danach der Zugriff auf ein Objekt geprüft. Da sich sowohl positive, als auch negative Rechte in den *Access Control Lists* eines Objektes befinden, ist eine selektive Behandlung des Zugriffs einzelner Nutzer auf einzelne Objekte mit den Mitteln der Rechteverwaltung einfach möglich. So ist es im Falle von Sicherheitsproblemen durch die mangelnde Vertrauenswürdigkeit eines Nutzers möglich, diesem Rechte auf ein bestimmtes Objekt kurzfristig zu entziehen und danach eine umfassende Analyse der Gruppenmitgliedschaften dieses Nutzers durchzuführen, um eine Korrektur der entsprechenden Rechte vorzunehmen. Prinzipiell sollte diese Analyse zwar nicht besonders aufwändig sein, da durch die zentrale Nutzerverwaltung dem Administrator die notwendigen Hilfsmittel an die Hand gegeben werden, jedoch zeigt die Realität in großen Systemen, dass eine gewisse Komplexität nicht vermeidbar ist, welche im Falle von Sicherheitsproblemen zu verzögerten Reaktionen führen kann. Keinesfalls sollten negative Rechte dauerhaft zum Einsatz kommen, um einzelnen Personen oder Gruppen Rechte zu entziehen, welche sie durch die Mitgliedschaft in anderen Gruppen erhalten haben. Wenn dies notwendig ist, so liegt meist ein Fehler in der Strukturierung der Gruppen vor.

4 Zusammenfassung

Ziel dieser Arbeit war eine Einführung in das weite Feld der Skalierbarkeit in verteilten Dateisystemen. Dazu wurde zuerst der Begriff "Skalierbarkeit" im aktuellen Kontext definiert. Wichtig war dabei die Unterscheidung zwischen *Lastskalierbarkeit*, also der Skalierbarkeit des Systems mit wechselnden Nutzerzahlen und damit Lastanforderungen, und *administrativer Skalierbarkeit*, also die Möglichkeit zur effizienten Verwaltung großer Subjekt- und Objektmengen in einem System. Nach Ansicht des Autors können verteilte Dateisysteme nur erfolgreich sein, wenn sie beide Probleme adressieren. Ein System, welches in der Lage ist, tausende von Nutzern zu bedienen, dabei aber eine Verwaltung durch Dezentralisierung extrem komplex macht, wird früher oder später aufgrund von Fehlern in Nutzerdatenbank oder Zugriffslisten zum Sicherheitsrisiko und damit nicht tragbar. Speziell große verteilte Dateisysteme als zentraler Speicher für die Daten ganzer Organisationen müssen hohe Anforderungen an die Sicherheit der ihnen anvertrauten Daten erfüllen. Bereits die Verletzung eines der drei Schutzziele *Vertraulichkeit*, *Integrität* und *Verfügbarkeit* kann für eine Organisation große Verluste bis hin zum finanziellen Ruin bedeuten. Daher muss ein Überblick über die Nutzer und die einfache Verwaltung der sie betreffenden Berechtigungen jederzeit durch das System ermöglicht werden. Anschließend wurden einige Problemstellungen genannt, welche sich bei verteilten Dateisystemen besonders stark auswirken. Diese sind die Nutzerzahl, die verwaltete Datenmenge, das Zugriffsverhalten der Nutzer auf einzelne Objekte und die verfügbare Netzbandbreite und -latenz. Ein skalierbares System muss diese Problemstellungen in Betracht ziehen um erfolgreich auf sie reagieren zu können.

Da es sehr viele verschiedene Implementierungen verteilter Dateisysteme gibt, wurde als zentrales Beispiel für diese Arbeit das Coda-Dateisystem vorgestellt. Dieses System, welches an der Car-

negie Mellon University als Nachfolger des Andrew File System entwickelt wurde, hat ebenso wie sein Vorgänger bereits in großen Installationen mit mehreren tausend Nutzern seine Nutzbarkeit bewiesen und kann daher als erfolgreiches Beispiel für die Anwendung der vorgestellten Ansätze zur Steigerung der Skalierbarkeit gelten. Die Vorstellung dieser Ansätze nimmt den größten Teil der Arbeit ein. Dabei wurde aus genannten Gründen Wert auf eine gleichwertige Betrachtung von Last- und administrativer Skalierbarkeit gelegt. Unter dem Oberbegriff "Lastskalierbarkeit" lassen sich dabei Maßnahmen wie Replikation, Caching, Callbacks und Hints zusammenfassen. Während Replikation und Caching sich eher auf die Verteilung der durch den Datenzugriff entstehende Netz- und Serverlast konzentrieren, lösen Callbacks und Hints das Problem der Konsistenzhaltung mehrfacher Kopien einzelner Daten. Ein speziell durch Caching entstehendes Problem ist der Zugriff auf veraltete Daten im Cache eines Clients. Um deren Konsistenz mit den Originalen sicherzustellen, kommen verschiedene Konsistenzprotokolle zum Einsatz. Nachdem der ursprüngliche Ansatz eines statuslosen Servers bei AFS Version 1 und die daraus resultierende erhöhte Netzlast durch Statusabfragen der Clients sich als problematisch erwiesen hatten, verwendete AFS Version 2 (und später als Nachfolger auch Coda) den Callback-Mechanismus, bei welchem der Server sich einen Status für jeden Client merkt und diesen über Änderungen an den angeforderten Daten informiert. Ein anderer Ansatz zur Performanzsteigerung sind die bereits genannten Hints. Diese repräsentieren Daten, die bei der Verwendung selbstvalidierend sind. Sie bewirken im Falle ihrer Korrektheit eine Beschleunigung eines Prozesses (beispielsweise durch Einsparung verschiedener Anfragen), haben aber im Falle der Fehlerhaftigkeit keine falschen Ergebnisse zur Folge.

Auf Seiten der administrativen Skalierbarkeit wurden die zentrale Nutzerverwaltung, Gruppen/Rechtevererbung und negative Rechte vorgestellt. Speziell die zentrale Nutzerverwaltung ist eine Grundlage für die effiziente Verwaltung weit verteilter Systeme. Es wurde kurz der von Coda verwendete Ansatz skizziert, welcher die Vorteile einer zentralen Nutzerverwaltung mit den Vorteilen eines ausfallsicheren dezentralisierten Systems verbindet. Dabei wurde auch auf die Fähigkeit zur Delegation von Administrationsaufgaben innerhalb einer Organisation eingegangen, welche bei Coda durch die Verwendung von Volume-lokalen Master Authentication Servern realisiert wird. Als Ansatz zur Strukturierung großer Nutzermengen wurde das Konzept der Gruppen und die damit verbundene Rechtevererbung vorgestellt. Diese beiden Konzepte ermöglichen eine konsistente Handhabung größerer Gruppen von Nutzern mit gleichen Rechten – ein Fall, welcher in der Struktur größerer Organisationen häufig auftritt. Als letzter Ansatz wurde das Prinzip der negativen Rechte vorgestellt. Dieses ermöglicht im Fall von Sicherheitsproblemen durch mangelnde Vertrauenswürdigkeit einzelner Nutzer einen gezielten Entzug von Rechten auf bestimmte Objekte. In komplexen Organisationsstrukturen kann eine Analyse der Rechtestruktur eines Subjekts im Falle der Entdeckung von Problemen zu lange Zeit in Anspruch nehmen, so dass dieses Mittel zum kurzfristigen Rechteentzug hilft, ein System zu sichern und so Zeit für diese Analyse zu gewinnen.

Zusammenfassend bleibt festzustellen, dass die genannten Ansätze ihre Tauglichkeit bereits in Systemen wie Coda und AFS bewiesen haben, welche mit Nutzerzahlen über 5000 und Verteilungen über teilweise mehr als 40 Standorte als Paradebeispiele für erfolgreiche verteilte Dateisysteme gelten.

Literatur

- [HKM⁺88] HOWARD, JOHN H., MICHAEL L. KAZAR, SHERRI G. MENEES, DAVID A. NICHOLS, MAHADEV SATYANARAYANAN, ROBERT N. SIDEBOTHAM und MICHAEL J. WEST: *Scale and Performance in a Distributed File System*. ACM Transactions on Computer Systems, 6(1):51–81, Februar 1988.
- [NWO88] NELSON, MICHAEL N., BRENT B. WELCH und JOHN K. OUSTERHOUT: *Caching in the Sprite Network File System*. ACM Transactions on Computer Systems, 6(1):134–154, 1988.
- [Sat89] SATYANARAYANAN, MAHADEV: *Integrating Security in a Large Distributed System*. ACM Transactions on Computer Systems, 7(3):247–280, August 1989.
- [Sat90a] SATYANARAYANAN, MAHADEV: *Coda: A Highly Available File System for a Distributed Workstation Environment*. IEEE Transactions on Computers, 39(4), April 1990.
- [Sat90b] SATYANARAYANAN, MAHADEV: *Scalable, Secure and Highly Available Distributed File Access*. IEEE Computers, 23(5), Mai 1990.
- [Sat92] SATYANARAYANAN, MAHADEV: *The Influence of Scale on Distributed File System Design*. IEEE Transactions on Software Engineering, 16(1), Januar 1992.
- [SHN⁺85] SATYANARAYANAN, MAHADEV, JOHN H. HOWARD, DAVID A. NICHOLS, ROBERT N. SIDEBOTHAM, ALFRED Z. SPECTOR und MICHAEL J. WEST: *The ITC Distributed File System: Principles and Design*. In: *Proceedings of the 10th ACM Symposium on Operating System Principles*, Seiten 35–50, Orcas Island, Washington, Dezember 1985.