Recherchebericht

swp11-4

18. April 2011

Begriffe

Folgende Begriffsdefinitionen beziehen sich auf (im abstrakten Sinne) verteilte Rechenprozesse und werden anderswo anders gebraucht. Die Anschauung und Definition sind an [1] angelehnt.

Byzantinischer Fehler Unbestimmter Fehler in einem verteilten System, bei dem ein oder mehrere Rechenprozesse das Kommunikationsprotokoll nicht verfolgen. Die Verletzung des Kommunikationsprotokoll kann dabei auch absichtlich herbeigeführt werden und muss nicht auf einen Fehler des Rechenprozesses zurückführbar sein.

Förderiertes System Logische Netzwerktopologie in der gleichberechtigte Kommunikation zwischen Servern und Kommunikation gemäß Client-Server-Paradigma zwischen Clients und den jeweiligen Servern stattfindet.

Integrität Einhaltung vorgegebener Eigenschaften innerhalb eines Rechenprozesses.

Konflikt Globaler Zustand, in dem kein Konsens zwischen mehreren Rechenprozessen erreicht werden konnte.

Konsens Kommunikationsbasierte Einigung zwischen mehreren Rechenprozessen mit anfänglich möglicherweise unterschiedlichen Ausgang Zuständen auf einen gemeinsamen Endzustand.

Persistenz Eigenschaft, dass die Daten eines Rechenprozesses dessen Ausführungen überdauern.

Replikation von Rechenprozessen Verteilung eines andernfalls zentralisierten Rechenvorgangs auf mehrere unabhängige Prozesse durch Kopieren von Daten und unabhängige Ausführung des selben Rechenvorgangs.

Single-Point-of-Failure Teil eines Systems, der bei Ausfall zum Ausfall des gesamten Systems führt, da er für die funktionieren des Systems notwendig ist und nicht mehrfach vorhanden ist.

Synchronisation Methode, um einen Konsens von Rechenprozessen zu erreichen.

Verteiltes System Verbund mehrerer unabhängiger Rechenprozesse, um ein gemeinsames Ziel zu erreichen.

Konzepte

Byzantinische Einigung Eine byzantinische Einigung beschreibt einen Konsens zwischen Rechenprozessen in einem verteiltem System unter der Annahme, dass byzantinische Fehler auftreten können. In einem verteilten System mit f Rechenprozessen, bei denen ein byzantinischer Fehler aufgetreten ist, und insgesamt $n + f$ Prozessen, muss $n > 2 \cdot f$ sein, damit eine byzantinische Einigung erreicht werden kann [4].

CAP-Theorem Das CAP-Theorem besagt, dass ein verteiltes System höchstens zwei der Eigenschaften Konsistenz (*consistency*), Verfügbarkeit (*availability*) und Toleranz gegenüber Netzwerkpartionierungen (*tolerance to network partitions*) vereinigen kann [2]. Daraus ergeben sich insgesamt drei Paarungen.

In einem verteilten System das Verfügbarkeit und Konsistenz garantiert, muss die zuverlässige Kommunikation zwischen den Rechenprozessen des Systems stets gewährleistet sein, Nachrichten dürfen nicht verloren gehen. Die synchrone Master-Master-Replication relationaler Datenbank ist ein bekanntes Beispiel für dieses Eigenschaftspaar: Im Falle der Trennung der Verbindung zwischen den Datenbankserver ist ein Schreibzugriff auf die einzelnen Server mit ACID-Eigenschaften nicht mehr möglich.

Ein verfügbares und gegenüber Netzwerkpartionierungen tolerantes System kann nicht konsistent sein. Im Falle einer Netzwerkpartionierung und gleichzeitigem Fortlauf der Rechenprozesse kann während der Partitionierung und ohne nachherige (nicht allgemein lösbare) Synchronisation kein Konsens und konsistenter Zustand hergestellt werden. Ein alltägliches Beispiel ist die Synchronisation von Dateien zwischen einem Laptop und Desktop-Computer, die nicht ständig verbunden sind, aber auf denen die gleichen Ausgangsdateien im Laufe eines Arbeitstags modifiziert wurden und danach synchronisiert werden müssen.

Gleichenfalls kann ein konsistentes und gegenüber Netzwerkpartionierungen tolerantes verteiltes System nicht stets verfügbar sein, denn abhängig vom Fehlermodell kann eine ausreichend große Menge an Prozessen getrennt werden, so dass kein Konsens und damit kein konsistenter Zustand erreicht werden kann.

Capability-based Security Capability-based Security ist ein Zugriffskontrollsystem, in dem nicht ein Satz dynamischer (z.B. Regel- und Rollensysteme) oder statischer Zugriffsregeln (z.B. Zugriffskontrolllisten) sondern der Besitz Capabilities, einer Art Schlüssel, den Zugriff von auf Ressourcen ermöglichen. Neben dynamischerer Vergabe und Weitergabemethoden ermöglichen Capabilities z.B. die Entkopplung der Rechte von Personen und Rechenprozessen [3].

Peer-to-Peer-Architektur Eine Peer-to-Peer-Architektur ist eine Form der Organisation eines Netzwerks und Auslegung von Netzwerksoftware, in der alle Netzwerkteilnehmer gleichberechtigt Senden und Empfangen können und gegenseitig erreichbar sind. Im Gegensatz zur wohl bekannten Client-Server-Architektur gibt es nur eine logische Softwarekomponente die beide Aufgaben erfüllt.

Eine stark- oder vollvermaschte physische Netzwerktopologie scheint dabei am besten geeignet für das Lastprofil der eines Peer-to-Peer Netzwerks.

Beschreibung der zu studierenden Aspekte

Mercurial

Mercurial ist dezentrales Versionskontrollsystem. In einem Versionskontrollsystem können Änderungen an in ihm verwalteten Daten aufgezeichnet, so dass später auf einzelne durch die jeweiligen Änderungen entstandene Versionen zugegriffen werden kann. Bei einem dezentralen Versionskontrollsystem gibt es zusätzlich keinen zentralen Speicherort der Daten, die komplette Versionsgeschichte der Daten kann problemlos auf andere Computer repliziert werden, die so entstandenen Replikate können unabhängig voneinander bearbeitet werden und verschiedene Änderungen können im Nachhinein unabhängig von einer zentralen Instanz zusammengeführt werden (*merging*).

Mercurial verwaltet in Dateien und Ordnern strukturierte Daten in Repositorien (*repositories*), voneinander im gewissen Sinne unabhängige Datenbestände. Mercurial erfasst jedoch nur Änderungen an Dateien, Verzeichnisse existieren innerhalb des Datenmodells von Mercurial nur als Teile der Pfade der von Mercurial verwalteten Dateien und werden aus diesen nach Bedarf rekonstruiert. Bei Replikation von Repositorien können so insbesondere leere Verzeichnisse nicht repliziert werden. Da die Daten in einem Repositorium einen abgeschlossenen Datenbestand bilden, ist die teilweise Replikation von Teilen eines Repositories (*partial checkout*) nur möglich, wenn die Teile aus vorher eingerichteten und unabhängigen Teilrepositorien (*subrepositories*), die in einem Hauptrepositorium (*main repository*) als Teilreposositorien verzeichnet sind, bestehen. Änderungen beziehen sich aus dem gleichen Grund auch auf das gesamte Repositorium werden in Änderungsmengen (*changesets*) zusammengefasst.

Die wesentliche Datenstruktur von Mercurial ist das Revisionsprotokoll (*revlog*). Für jedes Repositorium und jede von Mercurial in dem Repositorium verwaltete Datei existiert ein eigenes Revisionsprotokoll. Ein Revisionsprotokoll einer Datei heißt Dateiprotokol. Ein Revisionsprotokoll besteht aus einem separat gespeichertem Index und den eigentlichen Daten des Revisionsprotokolls. In einem Dateiprotokoll werden Revisionen von Daten entweder als Differenz (*delta*) zwischen zwei Revisionen oder als ganzes gespeichert. Innerhalb des Datenbereichs des Revisionsprotokolls (*revlog data*) sind die Revisionen nacheinander angeordnet. Im Revisionsindex (*revlog index*) sind fortlaufend Metadaten der jeweiligen Revisionen gespeichert, die unter anderem den Versatz und die Länge des Differenz- oder Volldaten (*hunks*), zwei Bezugsrevision der Revisionen (*parent revisions*), wobei die zweite Bezugsrevision nur beim Zusammenführen von Änderungen angegeben wird, eine Referenz auf die Revision der Änderungsgeschichte (*changelog*), der die Revision angehört, und ein Hashwert, der sich aus den Daten der Revison und den Bezugsrevisionen errechnet, umfassen. Es können weitere Metadaten als Schlüsselwertpaare am Anfang der Daten einer Revision stehen, was bei Umbenennungen genutzt wird, da ein Dateiprotokoll hier nicht umbenannt wird, sondern nur das Dateiprotokoll der neuen Datei referenziert wird. Neben der Speicherung von Revisionsunterschieden wird der Datenbereich einer Revision im Falle der Änderungsgeschichte auch zur Speicherung der Änderungsbeschreibung (*commit message*) und Metadaten der Änderungsmenge und im Falle des Manifests des Repositoriums, das den Zustand eines Repositoriums nach Anwendung einer gegebenen Änderungsmenge verzeichnet, zur Speicherung von Dateiname, Hash-Wert und Dateiberechtigungs Tupeln verwendet. Bis auf die Traversierung des vollständigen Revisionsgraphen, beispielsweise zur zeilenweisen Revisionsannotation von Dateien, die vermutlich in $O(n^2)$ liegt, liegen sämtliche Operationen auf einem Revisionsprotokoll in $O(n)$, viele sogar in $O(1)$.

Grundsätzlich sind alle Mercurial Repositorien unabhängig. Änderungen können so unabhängig von einander durchgeführt und später zusammengeführt werden, da mit Mercurial mehrere Versionen einer Datei (*heads*), also Revisionen einer Datei die nicht als Bezugsrevisionen referenziert werden, gleichzeitig verwaltet werden können. Mercurial hat im Sinne des von verteilten Datenbanken bekannten CAP-Theorems also die Eigenschaften Verfügbarkeit und Partitionstoleranz, da stets auf einem lokalen Repositorium gearbeitet wird und Änderungen auch ohne Synchronisation mit anderen Repositorien durchgeführt werden können. Konsistenz kann hierbei als Folge des CAP-Theorems nicht erreicht werden und so ist die manuelle Zusammenführung von Änderungen, die bei nicht kollidierenden Änderungen teilautomatisiert werden kann, notwendig. Gerade bei bei nicht zeilenorientieren Textdateien oder Binärdateien kann die Zusammenführung von Änderungen durchaus schwierig sein und scheint allgemein nicht entscheidbar.

Die Replikation von Änderungen besteht daher bei Mercurial im Wesentlichen daraus, anhand der Hash-Werte der Revisionen (die für lokale Operationen oft verwendete fortlaufende Revisionsnummer identifiziert eine Revision nicht eindeutig, da sie nur den Versatz im Revisionsindex angibt), die Revisionen zu finden, die in den Revisionsprotokollen des Zielrepositoriums der Replikation nicht vorhanden sind und diese an die entsprechenden Revisionsprotokolle anzufügen. Abhängig von dem für die Replikation verwendetem Übertragungsprotokoll kann die Replikation verschieden effizient sein, abhängig davon, ob die Revisionen einzeln übertragen werden können oder Teile von Dateien angefordert werden können. Nach der Replikation der Änderungen müssen dann, wenn mehrere Dateiversionen existieren und vollständige Konsistenz der Daten angestrebt wird, die Änderungen durch den Benutzer zusammengeführt werden.

Abhängig von der frei wählbaren Transportschicht kann die Replikation in verschiedenen Organisationsmodellen durchgeführt werden. Bei HTTP etwa ist dies das Client-Server-Modell. Durch die Unterscheidung zwischen dem Herunterladen (*pull*) und Veröffentlichen (*push*) von Änderungen ist bei Mercurial die Replikation in zwei Rollen, den Empfänger und Sender der Änderungen, die aber nicht notwendigerweise ein Client-Server-Modell implizieren, aufgeteilt, was aber nur die Replikation in oder von entfernten Repositorien betrifft. Die Replikation von oder in mehrere Repositorien kann automatisiert werden.

Mercurial allein umfasst neben den vom Dateisystem bereitgestellten Mechanismen keine Zugriffskontrolle. Für entfernte Repostorien kann der Zugriff aber auf Transportebene, beispielsweise über SSH-, SSL-Authentifizierung oder *HTTP Basic and Digest Access Authentication*, für das gesamte Repositorium, aber nicht einzelne Dateien, beschränkt werden. So lassen sich auch vom Benutzer entwickelte Authentifikationsmechanismen anbinden, so dass Mercurial sehr einfach in bestehende Authentifikationssysteme eingebunden werden kann. Ein Beispiel hierfür ist der mit Mercurial ausgelieferte Repositorienbetrachter hgweb, der die Authentifizierung optional über HTTP anhand von Zugriffssteuerungslisten (*access control lists*) durchführt und auch ein auf Mercurial ausgelegtes HTTP-basiertes effizientes Replikationsprotokoll implementiert. Aber auch eine Zweifaktorauthentifizierung mit Hilfe von Smartcards oder anhand biometrischer Merkmale wäre denkbar. Eine Zugriffskontrolle für einzelne Dateien ließe sich aber möglicherweise auch mit Hilfe von Erweiterungsfunktionen (*hooks*), Funktionen die nach bestimmten Ereignissen aufgerufen werden und die Möglichkeit bieten Änderungsmengen abzulehnen, in dem Schreibtransaktion für die jeweilige Änderungsmenge unter gewissen Umständen abgebrochen wird, umsetzen.

Eine Identitäts- und Integritätskontrolle ist durch kryptographisches Signieren von Änderungsbeschreibungen und deren Verfikation, die beispielsweise automatisiert durch entspre-

chende Hooks durchgeführt werden kann, möglich. Standardmäßig wird dafür eine GPG-Erweiterung mit Mercurial ausgeliefert, es gibt aber auch eine Erweiterung, die X.509 nutzt. Allgemein ist aber jede textbasierte Signatur denkbar.

Diaspora

Diaspora ist ein ähnlich zu E-Mail organisiertes föderalistisch-verteiltes soziales Netzwerk. Das zugrunde liegende Protokoll ist jedoch nicht dokumentiert. Im Weiteren beziehen sich Ausführungen deshalb auf den Hauptentwicklungszweig der Referenzimplementierung.

Diaspora ist als Netzwerk von Server, so genannten Pods, angelegt, auf denen Benutzer Konten haben. Ein Benutzerkonto und die damit verbundenen Daten werden Seed genannt. Ähnlich dem sozialen Netzwerk Facebook beinhaltet Diaspora für jeden Seed die Basisfunktionalitäten Profile, Statusmitteilungen und Nachrichtensystem. Anfängliche Versionen beinhalteten auch die Möglichkeit Fotoalben anzulegen, die derzeitige Version erlaubt nur den Anhang von Fotos an Nachrichten.

Zur Kommunikation zwischen Pods setzt Diaspora ausschließlich HTTP ein: Kontakte werden mit WebFinger und XRDS gesucht, öffentliche Statusmeldung mit OStatus ausgeliefert, die Benutzeroberfläche erhält Benachrichtigungen über Aktivitäten und Nachrichten in Echtzeit mittels WebSockets und die Kommunikation zwischen Pods erfolgt über das Salmon Protokoll mittels WebHooks.

Die Kommunikation zwischen den Pods ist ereignisbasiert, Nachrichten werden unverzüglich an die Pods aller Seeds, die diese empfangen sollen ausgeliefert. Schlägt mehrmals die Auslieferung fehl, wird die Nachricht für den entsprechenden Pod verworfen. Eine verzögerte Auslieferung findet nicht statt, lediglich öffentliche Nachrichten, die über OStatus ausgeliefert werden, können zeitversetzt abgerufen werden, da sie nicht ereignisbasiert ausgeliefert werden. Die Nachricht wird aus diesem Grund im Seed des Empfängers gespeichert. An die Nachricht angehängte Fotos verbleiben beim Sender und werden beim Betrachten von dort geladen.

Die Nachrichten werden also beim Versenden repliziert und dort verbleiben beim Empfänger, der sie dann lokal abruft. Fotos werden hingegen nicht repliziert und verbleiben beim Sender. Da Nachrichten nicht verändert werden können, entstehen auch keine Konsistenzprobleme. Eine Anwendung des CAP-Theorems erscheint wenig sinnvoll, da die Nachrichten nur zur Betrachtung auf andere Pods repliziert werden. Streng genommen arbeitet Diaspora auch nicht auf einem verteilten Datenbestand und ist bis auf dem Verbleib von Fotos von der Dezentralität mit E-Mails vergleichbar.

Diaspora bietet durch die Einteilung von Kontakten in Gruppen (*aspects*), die Möglichkeit Statusmitteilungen nur an eine Untermenge von Kontakten zu senden. Vorangelegt sind die Gruppen Work und Family. Durch das Nachrichtensystem, das mit einer sehr einfachen Version von E-Mail vergleichbar ist, können auch Nachrichten an einzelne Kontakt jenseits der Gruppeneinteilung versandt werden.

Jeder Seed hat einen persönlichen und öffentlichen RSA-Schlüssel, die auf dem zugehörigem Pod erzeugt und gespeichert werden und zur Signatur und Verschlüsselung sämtlicher ausgetauschter Nachrichten genutzt werden können. Durch die automatische Verifikation der Signaturen aller Nachrichten wird eine Identitätskontrolle durchgeführt.

Tahoe-LAFS

Tahoe-LAFS ist eine Netzwerk-Architektur zum sicheren, verteilten Speichern von Daten. Tahoe-LAFS verwendet ein Rechteverwaltungssystem, kryptografische Methoden und Erasure-Coding, um Vertraulichkeit, Datenintegrität und Ausfalltoleranz bei der Verwendung des dieses Netzwerks zu gewährleisten.

Ein zentrales Ziel bei der Konzipierung von Tahoe-LAFS war das Umgehen von Sicherheits- und Verfügbarkeitsproblematiken, die den meisten verbreiteten zentralen (Cloud-)Storage-Providern eigen sind. Zumeist bedeutet die Nutzung von konventionellen, zentralen Storage-Providern, dass der Nutzer Daten und Informationen eine zentralen Betreiber eines Server-Netzwerkes überlassen muss. Dementsprechend muss der Nutzer dem Anbieter vertrauen, dass dieser dafür Sorge trägt, dass zu autorisierte Personen Zugriff auf die Nutzerdaten erhalten und dass er diese Daten zuverlässig verfügbar hält. Ein großer, zentraler Storage-Provider kann aber nicht nur ein attraktives Ziel für Hacking-Angriffe darstellen. Darüber hinaus können aber auch Sicherheitslücken innerhalb des zentralen Storage Providers, etwa durch Fehlkonfigurationen oder durch verantwortungsloses oder absichtlich schädliches Handeln von Mitarbeitern [5]. Um die Abhängigkeit der Sicherheit von Nutzerdaten von dem Storage-Provider zu umgehen, wurde das Prinzip eines Least Authorization Filesystems entwickelt, welches sich seinerseits vom Principle of Least Authority (POLA) ableitet. Gemäß dieses Prinzips wird darauf abgezielt, dass kein Teilnehmer und Prozess innerhalb von Tahoe-LAFS mehr Privilegien oder Möglichkeiten hat als unmittelbar notwendig.

Die zu speichernden Datenbestände werden vom Nutzer mittels eines Tahoe-LAFS-Clients einem Tahoe-LAFS-Gateway übergeben. Dabei können entweder Gateway und Client auf demselben System ausgeführt werden oder die Daten werden mit einem Tahoe-LAFS-spezifischem Protokoll (Tahoe-LAFS-WAPI) vom Client zum Gateway übertragen. Im Gateway werden die Dateien und Verzeichnisse verschlüsselt, wobei die aus den verwendeten Schlüsseln Capabilities abgeleitet werden, die die Dateien und Verzeichnisse innerhalb des Dateisystems eindeutig identifizieren und zugleich verschiedene Rechte im Umgang mit der Datei repräsentieren. Nach dem Verschlüsseln, werden die Dateien in kleine Teile normierter Größe, sogenannte Chunks zerlegt, um die Verarbeitungsgeschwindigkeit, Speicherausnutzung und die Zugriffsgeschwindigkeit beim Abrufen zu verbessern. Anschließend wird jedes Share durch einen Erasure-Code auf in N Teile repliziert (mit der Voreinstellung $N = 10$). Vor der Replikation wird ebenfalls festgelegt, welche Anzahl K dieser N Teile nötig sind, um den ursprünglichen Share wiederherzustellen (Voreinstellung: $K = 3$). Anschließend werden für jeden Share die N Teile möglichst gleichmäßig auf verschiedene Storage-Server im Storage-Grid verteilt. Falls dabei im Idealfall jeder Teil eines Shares auf einem anderen Storage-Server liegt, genügt es wenn $\frac{K}{N}$ dieser Server später verfügbar und nicht kompromittiert sind, um einen Share wiederherzustellen. Ein zentraler Aspekt für die Sicherheit von Tahoe-LAFS ist der Umstand, dass das Bestandteile des Storage Grids selbst keine Möglichkeit haben, auf den Klartext von Nutzerdaten zuzugreifen, ein Nutzer von Tahoe-LAFS macht sich also nicht von Zusicherungen über die Integrität und Zuverlässigkeit einzelner Storage-Server abhängig.

Die Zugriffskontrolle für Dateien und Verzeichnisse innerhalb von Tahoe-LAFS wird über das Bekanntgeben von sogenannten Capabilities gesteuert. Capabilities sind kurze Bitketten, die eine Datei eindeutig identifizieren und zugleich kryptographischen Informationen zur Prüfung auf Integrität (*verify-cap*), zum Lesen des Klartextes der Datei (*read-cap*) und ggf. zum Ändern einer veränderlichen Datei (*write-cap*) darstellen. In der Standardeinstellung werden Dateien als unveränderlichen Dateien in das Dateisystem übernommen, es können aber auch

veränderliche Dateien erzeugt werden. Verzeichnisse werden als veränderliche Dateien realisiert, die Dateibezeichner und zugehörige Capabilities in einem speziellen Format beinhalten. Für die Prüfung auf Datenintegrität wird neben einer entsprechenden Capability zu jedem Share auch ein Merkle-Tree gespeichert. Ein Merkle-Tree ist eine binäre Baumhierarchie von partiellen Hashcodes einer gespeicherten Datei, die es ermöglicht, bereits bei weniger als K Teilen der mit Erasure-Code verschlüsselten Datei korrumpierte Teile zu erkennen. Dadurch kann ausgeschlossen werden dass etwa durch Einschleusen eines gefälschten Erasure-Code-Teils eine andere Datei wiederhergestellt wird, als Tahoe-LAFS übergeben wurde. Um Zugriffsrechte auf eine Datei weiterzugeben, werden die entsprechenden Capabilities an den Rechte-Empfänger übergeben. Darüber hinaus findet durch die Verzeichnishierarchie im Dateisystem eine implizite Weitergabe von Zugriffsrechten statt. Wenn etwa Nutzer A eine veränderliche Datei F in ein Verzeichnis ablegt, für das Nutzer B Leserechte (also die *read-cap*) besitzt, erhält er dadurch auch Lesezugriff auf die Datei F.

Das Tahoe-LAFS wurde als technische Grundlage für einen kommerziellen Storage-Provider entwickelt, mittlerweile wurden aber sowohl die Architektur als auch der Quellcode der Implementierung dieses Systems offengelegt. Tahoe-LAFS kann dadurch im derzeitigen Zustand oder nach Modifikation als Komponente für sicheres, verteiltes und ausfalltolerantes Speichernetzwerk für andere Software-Vorhaben genutzt werden.

Vergleich

Ein allgemeiner Vergleich der vorgestellten Software scheint jenseits der Einordnung innerhalb des CAP-Theorems und allgemeinem Vergleich des Zugriffskontrollmechanismus scheint schwierig, da die Software verschiedene Zwecke erfüllt, auf verschiedene Anwendergruppen zugeschnitten ist und Daten verschiedener Sensitivität und Verfügbarkeit speichert.

Ergebnis

Bezüglich der Zugriffskontrolle lässt sich ohne Anwendungsfall nicht sagen, ob der gewählte Mechanismus skaliert, für die Anwender angemessen ist oder sicher genug ist, denn gemäß Zookos Dreieck ist ein Zugriffskontrollmechanismus immer eine Abwägung und ein Kompromiss (im Sinne der Verzerrung des Dreiecks) zwischen Denzentralistät, Sicherheit und Benutzbarkeit. Für eine kleine Anwendergruppe mit über die Zeit geringem Änderungsbedarf kann eine zentral verwaltete Zugriffskontrollmatrix mit $n \times n$ Einträgen besser funktionieren als auf Kryptographie basierendes Capability-System, auch wenn es flexibler, dezentraler und sicherer ist und global skaliert, da die Anwender vielleicht gar nicht in der Lage sind es bedienen.

Vor diesem Hintergrund scheint es auch nicht verwunderlich, dass Diaspora, das als Ersatz für bestehende soziale Netzwerke, wie Facebook oder Twitter gedacht ist, einer eher einfachen an Sozialstrukturen orientieren Gruppenansatz, der zur Delegation von Zugriffsrechte überhaupt nicht geeignet ist und bei einer größeren Menge von Personen zu hohen Verwaltungsaufwand führt, gewählt hat, und Tahoe-LAFS als verteiltes Dateisystem, das eher auf versierte Anwender und Programmierer ausgerichtet ist, einen allgemeineren Ansatz wählt.

Für die zu entwickelnde Software bedeutet dies, dass sehr genau zu ermitteln ist, welche Anforderungen bestehen und wer die Benutzer des Systems sind, und anhand dieser Angaben dann passende Modelle ausgewählt und möglicherweise mit realen Benutzern getestet werden müssen.

Ein ähnlicher Ansatz muss für die Auswahl der Eigenschaften aus dem CAP-Theorem gewählt werden, denn diese bestimmen sich auch aus den Anforderungen an das Projekt bezüglich Verfügbarkeit, Konsistenz und Benutzbarkeit. Ein unerfahrener Benutzer könnte zwar seine Dokumente aus einem Textverarbeitungsprogramm in einem Mercurial-Repositorium speichern und Projektpartnern zur Kollaboration zugänglich machen, doch es ist nicht gesagt, dass der Benutzer im Falle eines Änderungskonflikts technisch (jenseits des Vergleich Zeile für Zeile) in der Lage wäre, die Änderungen allgemein zusammenzuführen, und es wäre in dem Fall vielleicht besser, wenn die Projektpartner auf einem gemeinsamen Dateisystem arbeiten würden und die Dokumente zur Bearbeitung der gesperrt würden, solange sie geändert werden, so dass Änderungskonflikte gar nicht erst entstehen können.

Zum bisherigen Zeitpunkt erschien uns jedoch keines der vorgestellten oder sonstig recherchierten System passend, was zu erwarten war, da die nach bisheriger Kenntnis zu entwickelnde Software so noch nicht existiert und die möglicherweise zu benutzenden Algorithmen für verteilte Systeme noch nicht massentauglich oder wiederverwendbar implementiert sind und verwendet werden.

Während der Recherche wurden aber wichtige Erkenntnisse jenseits des hier Dargestelltem und als einsetzbare Software Verfügbarem gewonnen. Daraus konnte ein Vorschlag zum Aufbau der Software entwickelt werden, der dem Kunden vorgestellt werden wird.

Literatur

[1] Christian Cachin, Rachid Guerraoui und Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* 2. Aufl. Springer, Feb. 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3.

[2] Seth Gilbert und Nancy Lynch. „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33 (2 2002), S. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[3] Ka-Ping Yee Mark Miller und Jonathan S. Shapiro. *Capability Myths Demolished.* Techn. Ber. Johns Hopkins University, 2003. URL: http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf.

[4] M. Pease, R. Shostak und L. Lamport. „Reaching Agreement in the Presence of Faults". In: *J. ACM* 27 (2 1980), S. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: http://doi.acm.org/10.1145/322186.322188.

[5] Christopher Soghoian. „Caught in the Cloud: Privacy, Encryption, and Government Back Doors in the Web 2.0 Era". English. In: *8 J. on Telecomm. and High Tech. L. 359* (2009). URL: http://ssrn.com/paper=1421553.