Universität des Saarlandes, Fachbereich Informatik

Failure Resilience

Fehlertoleranz in Peer-to-Peer-Systems am Beispiel des Projektes *OceanStore*

vorgelegt von:
Corinna Richter

Seminar Peer-to-Peer-Systems

Saarbrücken, den 29. November 2003

Inhaltsverzeichnis

1 Einleitung und Problemstellung

Der folgende Vortrag entstand im Rahmen des Seminars "Peer-to-Peer-Systeme" im Fachbereich Informatik an der Universität des Saarlandes im Wintersemester 2003/2004. Ausgangspunkt war der Text "Pond, the OceanStore Prototype" [08], der das Projekt *OceanStore*[1] vorstellt und die Ergebnisse des Prototypen Pond anhand verschiedener Benchmarks auswertet. Ausgehend von diesem Artikel soll im vorliegenden Referat der Begriff der Fehlertoleranz in Peer-to-Peer-Systemen näher erläutert werden. Dabei werden einige Lösungsansätze anhand der Implementierung von *OceanStore* vorgestellt und diskutiert.
Um die Zielsetzung dieses Vortrages zu verdeutlichen, ist zunächst eine klare Definition des Begriffes "Failure Resilience" notwendig. Im "Handbuch der Informatik" [05] findet sich folgende Definition:

> "Unter *Fehlertoleranz* versteht man die Fähigkeit eines Rechnersystems, sich trotz einer begrenzten Anzahl von Fehlern spezifikationsgerecht zu verhalten. Fehlertoleranz von Rechnersystemen zielt auf die Erhöhung der Zuverlässigkeit und Verfügbarkeit, also auf die Funktionsdauer und Funktionsbereitschaft des Systems. Sie kann auch zur Erhöhung der Sicherheit und Vertraulichkeit von Rechnersystemen beitragen."

Als zentral werden hier die Eigenschaften Zuverlässigkeit und Verfügbarkeit genannt. Intuitiv ist einleuchtend, dass die Zuverlässigkeit des Systems sowie die Verfügbarkeit von Daten und Ressourcen an Peer-to-Peer-Systeme deutlich höhrere Anforderungen stellen, als dies in traditionellen monolithischen Systemen der Fall ist. In monolithischen Architekturen ist die Art der auftretenden Fehler üblicherweise relativ gut vorhersehbar und lässt sich von außen kontrollieren, da die Konfiguration und die Eigenschaften des Systems hinlänglich bekannt sind. Peer-to-Peer-Netze dagegen sind offene, autonome Architekturen, d.h. die genaue Zusammensetzung des Systems ist variabel und die Reaktion der einzelnen Knoten unter Umständen nur begrenzt vorhersehbar. Peer-to-Peer-Systeme zeichnen sich daher durch folgende Charakteristika aus, die bezüglich der Fehlertoleranz besonders beachtet werden müssen:

- Ein Peer-to-Peer-Netzwerk ist üblicherweise einer großen Teilnehmerfluktuation unterworfen: Hosts können sich dem Netzwerk anschließen oder sich abmelden. Hier stellt sich die Frage nach einem fehlertoleranten Routing und nach einem Mechanismus, der Hosts auch bei häufigem Wechsel der Teilnehmer korrekt lokalisieren kann. Dies wird in *OceanStore* mit einem eigenen Routing- und Locationsystem "Tapestry" gelöst, das ich hier jedoch nicht detailliert vorstellen will, da es den Rahmen dieses Vortrages sprengen würde.

[1] *OceanStore* ist ein Peer-to-Peer-Systems, das an der Universität Berkeley in Zusammenarbeit mit IBM entstand siehe auch 2

- Aufgrund der offenen Struktur kann für die einzelnen Hosts keine Qualitätsgarantie übernommen werden: Rechner können nicht nur physisch ausfallen (sog. "Fail-Stop-Fehler"), sie können auch fehlerhaft reagieren (u.a. auch aufgrund äußerer Angriffe gegen das System) und so die Stabilität im gesamten Netzwerk gefährden. Hier muss also auch mit sogenannten "Byzantinischen Fehlern" gerechnet werden, die besondere Anforderungen an das System stellen (vgl. Absatz 3).

- In einem Peer-to-Peer-Netzwerk können Daten verteilt gespeichert werden. Da jedoch die Hosts, die die Datenspeicherung übernehmen, ausfallen oder sich abmelden können, muss das System Lösungen anbieten, die auch in diesem Fall die Verfügbarkeit der Daten gewährleisten (vgl. Absatz 5).

- Bezüglich der Authentizität der Daten ist eine größtmögliche Sicherheit gewünscht (Herkunft und Unverfälschtheit der Daten soll gewährleistet sein). Hierzu bieten sich kryptographische Modelle zum Versand der Daten und Nachrichten an, bei denen die beteiligten Teilnehmer über entsprechende Keys verfügen. Probleme entstehen hier unter Umständen, wenn einzelne Server ausgetauscht werden. Daher muss eine Lösung gefunden werden, die die Verständigung zwischen den momentanen Clients und den jeweiligen Servern nicht gefährdet, obwohl sich die Zusammensetzung des Systems häufig ändert.(vgl. Absatz 4).

Ein grundlegendes Ziel eines Peer-to-Peer-Systems ist es, die "Selbstheilungskraft" und die "Selbstorganisation" des Systems zu unterstützen. Dies gilt natürlich auch für den Aspekt der Fehlertoleranz: Das System selbst muss in der Lage sein, auf Fehler jeder Art sinnvoll zu reagieren und sich zu regenerieren. Fehlertoleranz in verteilten Systemen allgemein und Peer-to-Peer-Systemen im Besonderen beinhaltet somit auch die Fähigkeit des Gesamtsystems, den Fehler einer beteiligten Komponente zu maskieren und auszugleichen. Lösungsansätze für diese Fragestellungen sollen im Folgenden anhand des Projektes *OceanStore* genauer diskutiert werden.

2 *OceanStore* - ein kurzer Überblick

OceanStore entstand an der Universität Berkeley in Zusammenarbeit mit IBM. Ziel war die Entwicklung eines globalen, verteilten, beliebig skalierbaren, persistenten Datensystems. Die Grundidee war, die beteiligten Hosts aufzuteilen in hochpotente Hosts einerseits, die die Archivierung der Daten und die Serialisierung der verschiedenen Aktionen realisieren, und weniger starke Rechner andererseits, die Dienste in Anspruch nehmen können und als Gegenleistung Speicherplatz zur Verfügung stellen. Die grundlegende Speichereinheit des Modells, das "Datenobjekt", sollte so allgemein gehalten werden, dass sich darauf verschiedene verteilte Anwendungen aufsetzen lassen (verteiltes Dateisystem, E-Mail-Applikation, etc.). An das Datenobjekt wurden dabei folgende Anforderungen gestellt:

- universelle Erreichbarkeit von Daten, unabhängig von ihrer physichen Lokalisierung,
- Gewährleistung der nötigen Privacy bestimmter Daten trotz des angestrebten offenen Informationsaustausches,
- leicht verständliches Modell der Datenkonsistenz,
- Unverfälschtheit der Daten.

Dabei waren zwei Grundvoraussetzungen entscheidend für die Architektur von *OceanStore*: Die Infrastruktur ist nur in ihrer Gesamtheit zuverlässig, d.h. einzelne Hosts können unvorhersehbar ausfallen oder fehlerhaft reagieren. Jedoch ist sicherzustellen, dass jeweils nur ein bestimmter Teil der Hosts fehlerhaft ist, so dass das System als Ganzes aufgrund näher zu bestimmender Kompensationsmechanismen noch spezifikationsgerecht reagiert. Die zweite Annahme ist, dass die Infrastruktur permanenten Veränderungen unterworfen ist - Hosts melden sich ohne Vorwarnung beim Netzwerk an oder ab. Das System muss sich daraufhin selbst entsprechend neu organisieren können. Mit *OceanStore* sollte also ein System entworfen werden, das auf der Grundlage einer teilweise unzuverlässigen und fluktuierenden Infrastruktur ein Interface für eine zuverlässige verteilte Datenspeicherung und -bearbeitung realisiert. In den folgenden Abschnitten werden einige Besonderheiten der (komplexen) Architektur von *OceanStore* skizziert. Dabei sollen jedoch nur die Aspekte zur Sprache kommen, die für das Verständis der Lösungsstrategien bezüglich der Fehlertoleranz bei *OceanStore* essentiell notwendig sind. Für eine ausführlichere Darstellung sei vor allem auf die Artikel [08], [06] sowie auf die ausführliche Dokumentation im Netz unter http://oceanstore.cs.berkeley.edu verwiesen.

2.1 Gesamtarchitektur

OceanStore basiert auf dem Routing- und Locationsystem "Tapestry", das die Kommunikationswege in dem System reguliert. Clients können sich bei dem System anmelden und

ein bestimmtes Objekt anfragen, um eine Schreib- oder Leseaktion durchzuführen. Das Speichermodell für ein solches Datenmodell wird in Absatz 2.2 beschrieben. Um konkurrierende Updates für ein Objekt zu serialisieren und die Konsistenz der Daten zu sichern, wird jedem Objekt als kontrollierende Instanz eine primäre Replikation, eine Gruppe von Hosts, die das Objekt verwalten, zugeordnet. Die Aufgaben dieses inneren Rings sind in Absatz 2.3 beschrieben. Um die Effiezenz für den Zugriff auf einzelne Objekte zu erhöhen, können Clients ein Objekt nicht nur direkt bei der primären Replikation "anfragen", sondern auch bei einem anderen Client, der das Objekt in diesem Moment bearbeit. Dieser Mechanismus ist in Absatz 2.4 beschrieben.

2.2 Datenmodell

Ein Datenobjekt in *OceanStore* wird anhand einer eindeutigen AGUID (Active globally-unique identifier) im System identifiziert. Vergleichbar einer Datei im traditionellen Dateisystem kann ein Datenobjekt gelesen und/oder geschrieben werden. Da dies auch konkurrierend von mehreren Benutzern geschehen kann, ist ein Modell erforderlich, das die Konsistenz der Daten sichert (vgl. 2.3). Ein Datenobjekt kann im Laufe seiner Existenz verschiedene Änderungen durchlaufen - alle dabei entstehenden Versionen des Objektes werden persistent archiviert. Diese Versionen sind über eine VGUID (Version globally-unique identifier) eindeutig referenzierbar. Sie setzen sich aus verschiedenen Datenblöcken zusammen, wobei Datenblöcke, die sich in einzelnen Versionen nicht unterscheiden, auch nur einmal gespeichert und von beiden Versionen referenziert werden. Datenblöcke wiederum sind eindeutig über eine BGUID (Block globally-unique identifier) identifiziert.

2.3 Innerer Ring

Der innere Ring (auch die *primäre Replikation* genannt) ist genau einem Objekt zugeordnet und verwaltet deren momentane Instanzen im System (im Cache der Clients) und die archivierten Versionen. Um hier einen "single point of failure" zu vermeiden, besteht der innere Ring aus mehreren Servern. Zudem sind die Aufgaben des inneren Rings vergleichsweise rechenintensiv, so dass ein Lastenverteilung auf mehrere Rechner sinnvoll erscheint. Zu diesen Aufgaben zählt u.a. die Serialisierung der Änderungen, die an einem Objekt vorgenommen werden, die Prüfung, ob eine Änderung korrekt ist und die Propagierung der Änderungen an alle momentanen Instanzen des Objektes im System. Die Zusammensetzung des inneren Rings kann sich im Laufe der Zeit ändern (vgl. auch 4). Der innere Ring verständigt sich durch Nachrichten über Entscheidungen, die gemeinsam zu treffen sind. Er folgt dabei dem Byzantine Fault Protocol, das in Absatz 3 näher vorgestellt wird. Der innere Ring "kennt" auch die aktuellste Version eines Objektes und kennzeichnet diese durch einen sog. "Heartbeat". Sobald eine Änderung der Daten erfolgt ist, macht die

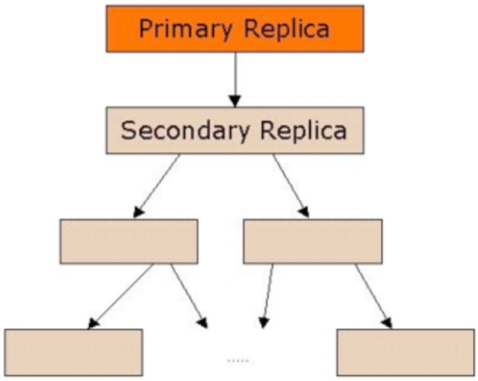

Abbildung 1: Dissemination Tree

primäre Replikation diese Änderung im Dissemination Tree bekannt (vgl. 2.4). Der innere Ring ist auch zuständig für die Archivierung von Daten, wie sie in Absatz 5 beschrieben wird.

2.4 Dissemination Tree

Ein Client, der ein bestimmtes Objekt sucht, lässt sich über Tapestry zu dem nächstgelegenen Host verbinden, der dieses Objekt aktuell in seinem Cache hat. Alle derartigen Instanzen eines Objektes werden als sekundäre Replikation bezeichnet. Sie sind in einer Baumstruktur angeordnet, an deren Wurzel die primäre Replikation steht. Die Struktur dieses Baumes ist selbstorganisierend, d.h. *OceanStore* nutzt Mechanismen der Introspektion, um Effizienz für die Traversierung des Baumes zu gewährleisten [06]. Ein Client, der ein bestimmtes Objekt bearbeiten möchte, wird von dem Routingsystem Tapestry zu dem nächstgelegenen Knoten geleitet, der dieses Objekt momentan bearbeitet. Diesen Knoten nutzt der Client als Elternknoten und fügt sich selbst in den Baum ein. Ein Client, der ein Objekt schreiben möchte, sendet sein Update an die primäre Replikation. Diese serialisiert die verschiedenen Änderungen und propagiert sie an alle anderen sekundären Replikationen. Der Vorteil dieser Struktur besteht u.a. darin, dass der aufwendige Dekodierungsprozess aus der Archivspeicherung, der in Absatz 5 beschrieben wird, nur dann durchgeführt werden muss, wenn keine Instanz des Objektes im Cache zu finden ist.

3 Byzantinsche Fehler vs. Fail-Stop-Prozesse

Bei einem Peer-to-Peer-System handelt es sich im Allgemeinen um ein sog. asynchrones System [04]. Die einzelnen Komponenten verständigen sich dabei in Form von Nachrichten, sog. Messages. Diese können zu einem beliebigen Zeitpunkt abgesetzt und evtl. auch mit einer starken Verzögerung ausgeliefert werden. Ein spezieller Aspekt dabei ist die gemeinsame Konsensfindung im System: Wie in Absatz 2.3 gezeigt wurde, verwalten *mehrere* Server die primäre Replikation eines Objektes. Bei Zugriffen auf dieses Objekt müssen sich die Server des inneren Rings somit über ihre Entscheidungen abstimmen, um zu einem gemeinsamen Konsens zu gelangen. Zum Beispiel muss für die durchzuführenden Updates eine eindeutige Reihenfolge der Ausführung festgelegt werden. Die Konsensfindung innerhalb der primären Replikation erfolgt ebenfalls über den Austausch von Nachreichten. Wünschenswert ist nun, dass die gemeinsame Entscheidung trotz einzelner fehlerhafter Hosts, evtl. sogar trotz äußerer Manipulationen spezifikationsgerecht bleibt. Dabei ist sowohl zu fragen, wieviele fehlerhafte Prozesse ein System maximal kompensieren kann, als auch, wie ein Protokoll aussehen kann, das die Konsensfindung unterstützt. Zunächst soll jedoch betrachtet werden, welche unterschiedlichen Arten von Fehlern in einem Peer-to-Peer-System von Bedeutung sind.[2]

3.1 Fail-Stop-Prozesse

Fail-Stop-Prozesse sind Prozesse, bei denen die fehlerhafte Komponente sich genau in einem von zwei Zuständen befinden kann: Entweder arbeitet sie korrekt und verhält sich gemäß dem vereinbarten Protokoll, oder sie arbeitet gar nicht und sendet entsprechend auch keine Nachrichten mehr. Dies kann z.B. der (Hardware)-Ausfall einer bestimmten Komponente sein. Fail-Stop-Fehler treten ohne Vorwarnung auf, d.h. sie lassen sich nur aufgrund der fehlenden Reaktion erkennen. Fehler dieser Art findet man in allen Softwaresystemen und sie lassen sich aufgrund des ihnen innewohnenden Determinismus verhältnismäßig leicht feststellen. Bracha und Toueg [02] haben bewiesen, dass die korrekte Funktionsweise des Systems nicht beeinträchtigt ist, wenn nur Fail-Stop-Fehler auftreten und mehr als die Hälfte der Rechner korrekt arbeitet. In einem offenen, autonomen Peer-to-Peer-System muss jedoch leider nicht nur mit Fail-Stop-Fehlern gerechnet werden, sondern auch mit 'Byzantinischen Fehlern'.

[2] In der Literatur finden sich verschiedene, teilweise noch feingranularere Klassifizierungen. Ich übernehme hier die Klassifizierung, die Bracha und Toueg [02]verwenden.

3.2 Byzantinische Fehler

Byzantinische Fehler sind Fehler, bei denen die fehlerfafte Komponente (evtl. auch bewusst) *verfälschte* Nachrichen senden kann, d.h. die Reaktion einer Komponente garantiert noch keine korrekte Arbeitsweise. Peer-to-Peer-Systeme bieten aufgrund ihrer offenen Struktur eine große Angriffsfläche für Fehler dieser Art.
Schon in den 80er-Jahren entstanden Untersuchungen dazu, wieviele fehlerhafte Komponenten ein Netzwerk maximal tolerieren kann, bzw. wie ein Protokoll für die Konsensfindung in einem offenen, asynchronen System aussehen muss, wenn sowohl Fail-Stop als auch Byzantinische Fehler auftreten. Das Problem wurde in der Literatur als "Byzantine Generals Problem" diskutiert (vgl. Absatz 3.2.1). Dabei wurde in verschiedenen Untersuchungen[3] gezeigt, dass eine korrekte Funktionsweise des Systems nur gewährleistet sein kann, wenn zu jedem Zeitpunkt weniger als ein Drittel der beteiligten Prozesse fehlerhaft ist. Die grundlegende Funktionsweise des Byzantine Fault Protocols, das ein Protokoll zur Konsensfindung beschreibt, wird in Absatz 3.2.2 geschildert, während Absatz 3.2.3 die Umsetzung des Byzantine Fault Protokolls bei *OceanStore* beschreibt.

3.2.1 Das Problem der Byzantinischen Generäle

Das "Byzantine Generals Problem" geht von folgender anschaulicher Situation aus: Eine Stadt wird belagert durch die byzantinische Armee, die aus mehreren Divisionen besteht. Jede Division wird von einem General angeführt. Die Armee hat nur dann eine Chance, die Stadt einzunehmen, wenn genügend Divisionen *gleichzeitig* angreifen. Einer der Generäle fungiert als Oberbefehlshaber und stößt den Vorgang an. Er sendet all seinen Generälen einen bestimmten Befehl, in Form einer Nachricht. Dies kann entweder der Befehl zum Angriff ("attack") oder aber zum Rückzug ("retreat") sein. Er verbreitet seine Entscheidung an alle Generäle, wobei jedoch keiner der Empfänger weiß, welche Botschaft die *anderen* Generäle empfangen haben. Daher tauschen alle Generäle in darauf folgenden "Runden" den jeweils empfangenen Befehl aus. Dies geschieht beispielsweise in der ersten Runde in der Form "Der Befehlshaber hat X befohlen". Unglücklicherweise gibt es jedoch Verräter unter den Generälen, die Nachrichten verfälschen oder bewusst verzögern können. Zudem kann auch die Integrität des Oberbefehlshaber nicht garantiert werden: Er könnte *unterschiedliche* Befehle an die einzelnen Generäle gesendet haben, so dass ein überzeugender Angriff unmöglich ist.
Gesucht ist eine obere Grenze für illoyale Generäle, so dass Folgendes garantiert ist: Sei n die Anzahl der Generäle, wovon einer der Oberbefehlshaber ist und es v_i die Entscheidung von General i. Dann soll gelten:

1. Alle loyalen Generäle i , j kommen zu derselben Entscheidung: $v_i = v_j$.

[3] (u.a. [07], [03] und [02])

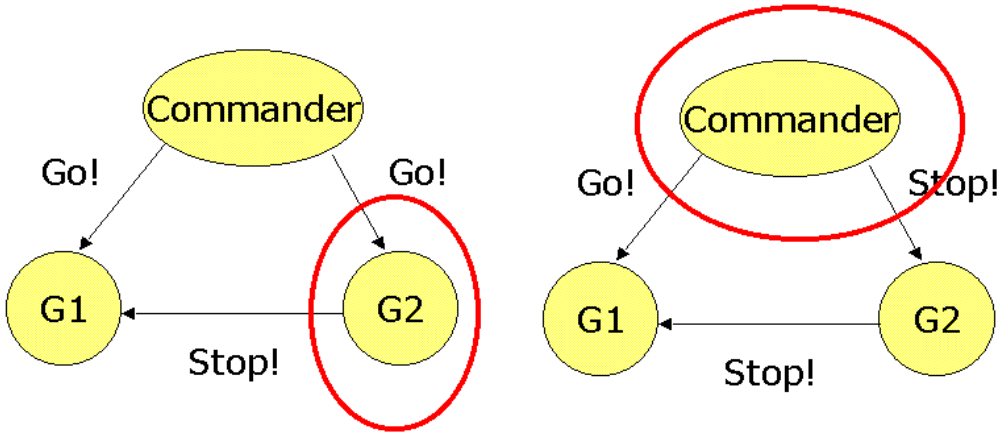

Abbildung 2: Das Problem der Byzantinischen Generäle für drei Generäle

2. Wenn der Oberbefehlshaber kein Verräter ist, teilen alle loyalen Generäle seine Entscheidung.

Ein Protokoll, das korrekt arbeitet, muss die obengenannten Bedingungen erfüllen. Um das Problem zu verdeutlichen, ist in Abbildung 2 die Situation für n=3 Generäle dargestellt: In der linken Abbildung ist der Oberbefehlshaber loyal und sendet beiden Generälen den Befehl zum Angriff. Der General G2 jedoch ist ein Verräter und verfälscht die Nachricht in seiner Mitteilung an G1. In der rechten Abbildung dagegen ist der Commander nicht loyal und sendet widersprüchliche Befehle an G1 und G2. Diesmal gibt G2 den Wert, den er vom Commander erhalten hat, korrekt an G1 weiter. In beiden Fällen steht G1 vor der gleichen Situation, denn für ihn ist nicht zu unterscheiden, ob der Commander oder aber G2 der Verräter ist. Ist der Commander ein Verräter, wie in der Abbildung rechts, müsste er gemeinsam mit G2 eine Default-Aktion befolgen. Hat jedoch G2 die Nachricht verfälscht, so muss er den Befehl des Commanders befolgen. Diese vereinfachte Situation zeigt die unlösbare Situation für 3 Generäle, von denen einer ein Verräter ist. Die Situation lässt sich nun auch auf Mengen von Generälen übertragen. Dabei haben umfangreiche Beweise folgenden Satz bestätigt:[4]

Das Byzantine Generals Problem ist nur dann lösbar, wenn weniger als ein Drittel der beteiligten Prozesse fehlerhaft ist.

[4]Ein intuitiver, gut verständlicher Beweis für die Korrektheit dieser Aussage findet sich u.a. bei Bracha und Toueg [02].

3.2.2 Das Byzantine Fault Protocol

Dieses Besipiel lässt sich leicht übertragen auf Konsensprobleme in einem verteilten, asynchronen Netzwerk, in unserem Fall also z.B. darauf, dass der innere Ring korrekte und einheitliche Entscheidungen treffen muss. Die Server des inneren Rings entsprechen hier den beteiligten Generälen. Einer von ihnen stößt eine Aktion an, und selbst, wenn dieser Initiator oder andere Hosts des inneren Ring fehlerhaft sind, soll am Ende eine korrekte, mehrheitsgetragene Entscheidung stehen. Bei der Lösung des Byzantine General Problems geht es v.a. um eine obere Grenze für die Anzahl der fehlerhaften Prozesse und somit um die Frage, wieviele "Störungen" ein System maximal kompensieren kann. Da die einzelnen Knoten des inneren Ring nicht "wissen", welchen Befehl die anderen Hosts erhalten haben, müssen sie sich über ihre Entscheidung austauschen. Das Byzantine Fault Protocol fügt zu diesem Zweck eine schwache Synchronisierung in "Runden" ein. In der ersten Runde multicastet der anstoßende Prozess seine Entscheidung an die anderen n-1 Prozesse. In der folgenden Runde sendet jeder der n-1 Prozesse den Wert, den er erhalten hat, an die n-2 Prozesse, von denen er noch keine Nachricht bekommen hat. Dies Verfahren wird f+1 Runden fortgesetzt, wobei f die maximale Anzahl fehlerhafter Prozesse im System ist. Jeder Prozess nimmt als zu versendenen Wert in Runde i eine Auswertung der Nachrichten, die er in Runde i-1 erhalten hat. Diese Auswertung findet nach dem Majoritätsprinzip statt.
Ein Beispielablauf für vier Prozesse ist in Abbildung 3 dargestellt. Angenommen sei folgende Situation: Für ein Update X ist ein Zeitpunkt der Ausführung festzulegen, d.h. eine Ordnungsnummer in der Reihenfolge der serialisierten Schreibaktionen. P1 stößt diese Aktion an und sendet seine Entscheidung ("i") an die anderen drei Prozesse. Anschließend tauschen sich P2, P3 und P4 in dem genannten Verfahren über den erhaltenen Wert aus, d.h. P2 teilt P3 und P4 seinen Wert mit, etc. Betrachtet man die Entscheidungsgrundlage der korrekten Prozesse P2 und P3 in der linken Darstellung, so verfügen beide trotz der verfälschten Nachricht von P4 über folgenden Entscheidungsvektor: (i,i,k). Somit würden sie die korrekte Entscheidung von P1 teilen. Hat P1 dagegen widersprüchliche Werte gesandt, wie dies in der rechten Abbildung der Fall ist, so verfügen am Ende dennoch alle korrekten Prozesse über *denselben* Entscheidungsvektor (i,j,k), so dass unabhängig von der Gleichheit einiger Werte in jedem Fall alle korrekten Prozesse dieselbe Entscheidung treffen werden.

3.2.3 Die Implementierung des Byzantine Fault Protocol bei *OceanStore*

Laut Byzantine Fault Protocol muss garantiert sein, dass *zu jedem Zeitpunkt* weniger als ein Drittel der beteiligten Server fehlerhaft ist. Bei maximal f fehlerhaften Servern muss demnach die Gesamtanzahl der Hosts mindestens 3f+1 sein. Dies ist eine problematische

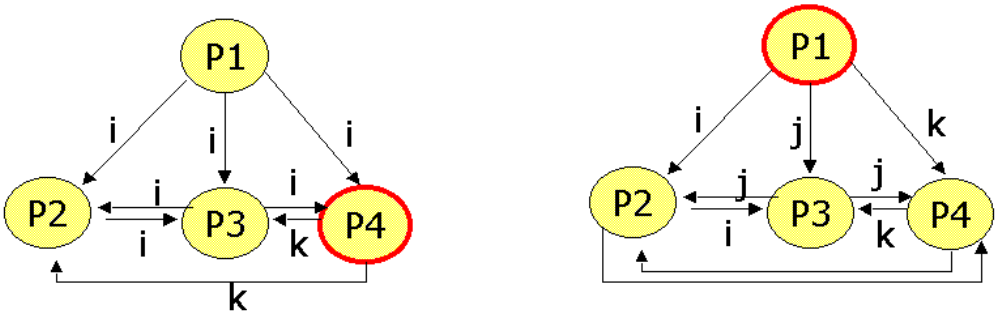

Abbildung 3: Byzantine Fault Protocol für vier Prozesse

Anforderung für Systeme mit einem langen Lebenszyklus. In früheren Implementierungen des Byzantine Fault Protocols wurde dieses Problem durch einen Reboot der Hosts von einer sicheren Partition aus in regelmäßigen Abständen gelöst. Dies setzt jedoch voraus, dass die Hosts des inneren Rings fest sind und jegliche kryptographischen Informationen fest gespeichert sind. Die Entwickler von *OceanStore* wünschten sich hier eine flexiblere Zusammensetzung des inneren Rings und gingen daher einen anderen Weg: Sie tauschen die Server des inneren Ring in regelmäßigen Abständen aus. Die Verantwortung für diesen Prozess liegt bei einer "Responsible Party". In einem System können mehrere Responsible Parties mit verteilten Aufgaben existieren. Diese Instanz ermittelt durch Analyse und Messungen Knotenmengen des Netzwerkes, die über einen längeren Zeitraum fehlerfrei waren und publiziert diese in Tapestry. Um einen neuen inneren Ring zu bilden, wird aus jeweils 3f+1 dieser unabhängigen Knotenmengen je ein Knoten gewählt. Auf diese Weise kann das System garantieren, dass der innere Ring eines Objektes zu jedem Zeitpunkt zu weniger als einem Drittel aus fehlerhaften Hosts besteht.
Ein Client, der eine Anfrage an den inneren Ring stellt, kann also mit einer korrekten Bearbeitung rechnen. Ein weiterer Aspekt, der dabei betrachtet werden muss, ist die Frage, wie Nachrichten im System übertragen werden. Das Byzantine Fault Protocol wurde in verschiedenen Formen implementiert. Ursprünglich wurden hierbei v.a. symmetrische Verschlüsselungsverfahren angewandt (sog. MACs, "Message Authentication Codes"), bei denen sich Sender und Empfänger "kennen" mussten. Während dies für die Kommunikation innerhalb der primären Replikation möglich ist (begrenzte Anzahl von Teilnehmern), ist dies für die Kommunikation mit den sekundären Replikationen und Clients nicht wünschenswert. Aus diesem Grunde kombiniert *OceanStore* symmetrische Verschlüsselung (innerhalb des inneren Rings) und Public-Key-Verfahren (zur Kommunikation mit "dem Rest der Welt"). Die spezifischen Probleme, die dabei entstehen, schildert der folgende Abschnitt.

4 Proactive Threshold Signatures

Symmetrische Verschlüsselungsverfahren haben üblicherweise einen großen Vorteil gegenüber Public-Key-Verfahren: Sie sind wesentlich effizienter zu berechnen. Dennoch hätte ein rein symmetrisches Verfahren einen entscheidenden Nachteil: Es wäre dann nicht möglich, dass Clients die Daten selbstständig verifizieren - für jede Replikation wäre die direkte Kommunikation mit dem inneren Ring erforderlich. Dadurch müsste die Anzahl der sekundären Replikationen zwangsweise begrenzt werden, um das Nachrichtenaufkommen für die innere Replikation handhabbar zu halten.
Bei *OceanStore* wurde die symmetrische Verschlüsselung nur für den Nachrichtenaustausch *innerhalb des inneren Ring* angewandt. Nach außen ist die primäre Replikation über *einen* Public Key ansprechbar, der allen beteiligten Servern des inneren Ring gemeinsam ist. Wie oben beschrieben, werden die Server des inneren Rings in regelmäßigen Abständen ausgetauscht. Damit dies nicht zum Erliegen oder Verfälschen des Datenaustauschs mit den sekundären Replikationen führt, arbeitet der Protoyp Pond von *OceanStore* mit sog. "Proactive Threshold Signatures".

4.1 Funktionsweise

Threshold Signatures arbeiten mit einem Public key, zu dem in einem verteilten Algorithmus l (hier l = 3f+1 = Anzahl der Hosts des inneren Rings) "Private Key Shares" erzeugt werden. Jeder der l Hosts nutzt seinen Private Key Share, um ein "Signature Share" zu generieren. Beliebige f+1 dieser Signature Shares können (gleichlautende Nachrichten vorausgesetzt) zu einer vollwertigen Signatur kombiniert werden. Da wenigstens einer dieser f+1 Server laut vorausgesetzter Byzantinischer Vereinbarung fehlerfrei ist, alle fehlerfreien Server aber deterministisch arbeiten, ist die Korrektheit der Entscheidung gewährleistet. Folgendes Beispielszenario soll die Richtigkeit dieser Aussage veranschaulichen (vgl. [03]): Ein Client sendet einen Request zur Ausführung eines Updates an die primäre Replikation. Diese einigt sich intern über einen Ausführungszeitpunkt, wie im Abschnitt 3.2.2 beschrieben. Nach der Ausführung des Updates erhält der Client eine Antwort (Reply) von verschiedenen Hosts des inneren Rings, die u.a. einen Zeitstempel t und ein Ergebnis r der Operation beinhalten und mit dem Signature Share eines Hosts der primären Replikation signiert wurden. Der Client wartet nun, bis er f+1 dieser Reply-Nachrichten mit demselben Zeitstempel t und demselben Ergebnis r erhalten hat. Ein Zertifikat aus f+1 solcher korrekt signierten Replies zeigt an, dass das Ergebnis gültig ist, da wenigstens einer dieser Hosts korrekt arbeitet, alle f+1 dasselbe Resultat aufweisen und alle korrekten deterministisch agieren.
Erhält der Client innerhalb eines bestimmten Zeitraumes nicht genügend Reply-Nachrichten, so gibt er dies an alle Replikationen bekannt, so dass ein eventuelles Fehlverhalten des in-

neren Rings schnell entdeckt werden kann.

4.2 Austausch der Server des inneren Rings

Es stellt sich nun die Frage, wie der Austausch der Hosts des inneren Rings vollzogen werden kann, ohne die Kommunikation mit den Clients zu behindern. Bei einem Austauschverfahren wird in einem verteilten Algorithmus eine neue Menge von 3f+1 Private Key Shares erzeugt. Ein Schlüssel dieser Menge kann nicht mit den vorherigen kombiniert werden. Die alten Schlüssel werden daraufhin vernichtet. Alle korrekt arbeitenden Hosts des inneren Rings werden auch die Vernichtung ihres alten Schlüssels korrekt durchführen. Da höchstens f Hosts fehlerhaft sind, bleiben demnach höchstens f ungültige Schlüssel im System, womit keine gültige Signatur des alten Sets mehr erzeugt werden kann.
Der Public Key der primären Replikation hat sich jedoch bei diesem Verfahren, trotz des Austausches der einzelnen Hosts nicht geändert. Somit können Clients und sekundäre Replikationen unverändert die Daten über diesen Public Key verifizieren.

5 Replikation und Erasure Coding

Um die Verfügbarkeit der Daten in verteiltgen Systemen zu garantieren, kann mit redundanten Daten gearbeitet werden. Bei der naiven Replikation werden die Daten lediglich repliziert und verteilt gespeichert. Nachteil dieses Verfahrens ist, dass der Speicherplatzbedarf in einem relativ schlechten Verhältnis zur dazu gewonnenen Datenverfügbarkeit steht, d.h. um die Verfügbarkeit ausreichend zu erhöhen, wird viel Speicherplatz benötigt. Komplexere Formen der redundanten Speicherung findet man u.a. bei den sog. RAID-Systemen (Redundant Array of Inexpensive Disks): RAID-Systeme ab Level 2 arbeiten mit der Aufspaltung der Daten und der Verteilung der einzelnen Fragmente. Einen ähnlichen Ansatz bietet das Erasure Coding, das *OceanStore* verwendet. Hierbei wird ein Datenblock in m Fragmente unterteilt, die wiederum in n Fragmente kodiert werden. Dabei gilt n >m. Der Wert $\frac{m}{n}$ bezeichnet hierbei die Kodierungsrate (Rate of Encoding). Der zusätzliche Speicherplatzbedarf ergibt sich aus $\frac{n}{m}$. Die verschlüsselten Fragmente werden auf verteilten Servern abgespeichert. Der große Vorteil bei diesem Prinzip liegt in der Form der Kodierung, die es erlaubt, den Datenblock mit *beliebigen m der n Fragmente* zu rekonstruieren. Bei Pond wird hierfür der Cauchy Reed Solomon Code verwendet, ein XOR-basiertes Kodierungsschema [01].

5.1 Datenverfügbarkeit

Folgende Situation diene hier als Beispiel: Ein Datenblock wird in 16 Fragmente unterteilt, die wiederum in insgesamt 32 Fragmente verschlüsselt werden. Diese verschlüsselten Fragmente werden verteilt gespeichert. Die Kodierungsrate ist hier $\frac{1}{2}$, der Speicherplatzbedarf erhöht sich auf das Doppelte. Für die Rekonstruktion eines Datenblockes sind nun jedoch beliebige 16 (=m) Fragmente ausreichend, d.h. man kann die Fragmente auf wesentlich mehr Arten rekonstruieren, als das bei der naiven Replikation der Fall ist. Dies ist unmittelbar einleuchtend, wenn man die Anzahl der möglichen Permutationen der Fragmente betrachtet. H. Weatherspoon und J.Kubiatowicz zeigten in einem quantitativen Vergleich [09] die unterschiedliche Datensicherheit in beiden Verfahren. Sie untersuchten dabei sowohl die Verfügbarkeit der Daten, die benötigte Bandbreite der Übertragung und die "Mean time of Failure" (MTTF, nach welcher Zeit tritt durchschnittlich ein Fehler auf). Im Falle von n=32 und m=16 zeigten die Autoren eine um den Faktor 4000 gesteigerte Datenverfügbarkeit. Auch in Bezug auf die notwendige Übertragungsleistung und die MTTF zeigten Systeme mit Erasure Coding ein deutlich günstigeres Verhalten (vgl. [09]).

5.2 Effizienz

Der Nachteil des Erasure Coding besteht in der aufwendigen Kodierung und (De-)kodierung der Fragmente. Diesen Prozess führt die Primäre Replikation durch. Um den Rechenauf-

wand gering zu halten, wird die Dekodierung nur dann vorgenommen, wenn keine Instanzen des Objektes im Cache sind (vgl. Absatz 2.4). Wird ein bestimmtes Objekt angefordert, prüft Tapestry zunächst, ob es schon im Cache eines anderen Client verfügbar ist. Ist dies der Fall, so muss keine Dekodierung stattfinden und das Objekt kann direkt aus dem Cache verwendet werden. Nur falls keine aktuelle Instanz des Objektes im System zu finden ist, leitet Tapestry die Anfrage weiter an die primäre Replikation, die dann die Rekonstruktion der Daten aus den archivierten Fragmenten anstößt. Diese Kombination aus Erasure Coding und einem größtmöglichen Ausnutzen von Caching-Prinzipien gewährleistet somit eine große Datenverfügbarkeit bei effizientem Datenzugriff.

6 Zusammenfassung und Ausblick

Unter Fehlertoleranz versteht man die Eigenschaft eines Systems, auch bei Auftreten einer begrenzten Zahl von Fehlern noch spezifikationsgerecht zu reagieren. Zentrale Begriffe sind dabei insbesondere die der Verfügbarkeit und Zuverlässigeit. Aufgrund der hohen Fluktuation, der autonomen Funktionsweise sowie der Anfälligekeit für sog. Byzantinische Fehler stellt Fehlertoleranz an Peer-to-Peer-Systeme besonders hohe Anforderungen. Spezifische Probleme und mögliche Lösungsansätze sollte dieser Vortrag anhand des Projektes *OceanStore* (Universität Berkeley in Zusammenarbeit mit IBM) diskutieren.
Ziel von *OceanStore* war die Entwicklung eines globalen, beliebig skalierbaren, persistenten Datenspeichersystems, das auch trotz einer unzuverlässigen Infrastruktur noch spezifikationsgerecht reagiert. Hier ist insbesondere an den Ausfall und die fehlerhafte Reaktion der beteiligten Hosts zu denken. Auf eine Betrachtung des von OceanStore benutzten Routingsystems Tapestry wurde hier verzichtet, da es den Rahmen des Vortrages gesprengt hätte. Bezüglich der Zuverlässigkeit der Ressourcen standen vor allem das Byzantine Fault Protocol und seine Implementierung bei *OceanStore*, sowie Proactive Threshold Signatures im Mittelpunkt der Untersuchung. Beide Prinzipien zielen auf ein korrektes Verhalten des Systems trotz einer begrenzten Anzahl von Fehlern. Anhand verschiedener Untersuchungen konnte gezeigt werden, dass für ein System, in dem byzantinische Fehler auftreten können, nur dann ein korrektes Verhalten garantiert werden kann, wenn weniger als ein Drittel der beteiligten Hosts fehlerhaft sind. In *OceanStore* wird dies durch einen regelmäßigen Austausch der Server des inneren Rings garantiert.
Bezüglich der Datenverfügbarkeit wurde das Prinzip des Erasure Codings erläutert, eine fragmentierte Form der Speicherung, die eine hohe Datenverfügbarkeit bei effizienter Ausnutzung von Speicherplatz gewährleistet.
Der Prototyp von *OceanStore* wurde mittels verschiedener Benchmarks getestet [08]. Dabei hat sich gezeigt, dass die Berechnung der Threshold Signatures der größte Schwachpunkt des Systems ist, wenn man die Effizienz betrachtet. Hier sollen weitere wissenschaftliche Arbeiten an der Universität Berkeley mögliche Alternativen aufzeigen.

Literatur

[01] J.Blömer, M.Kalfane, R.Karp, M.Karpinsiki, M.Luby, D.Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report, International Computer Science Institute, Berkeley, California, 1995.

[02] G.Bracha, S.Toueg. Asynchron Consensus and Broadcast Protocols. In: Journal of the Association for Computing Machinery, October 1995.

[03] M.Castro und B.Liskov. Proactive Recovery in a byzantine-fault-tolerant system. In Proc.of Sifmetrics, Juni 2000.

[04] F.C.Gärtner. Fundamentals of Fault-Tolerand Distributed Computing in Asynchronous Environments. In: ACM Computing Surveys, Vol. 31, No.1, March 1999.

[05] Handbuch der Informatik (hrsg. von Peter Rechenberg und Gustav Pomberger).-2.aktualisierte und erweiterte Auflage. München / Wien, 1999.

[06] J. Kubiatowicz, D. Bindel, Yan Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November, 2000.

[07] L.Lamport, R.Shostak, M.Pease. The Byzantine Generals Problem. ACM Trans. Prog.Lang.Syst.4,3,(July 1982, 228-234).

[08] S.Rhea, P.Eaton, D.Geels, H.Weatherspoon, B.Zhao, J.Kubiatowicz. Pond: The OceanStore Prototype. In Proc. of USENIX File and Storage Technologies FAST, 2003.

[09] H.Weatherspoon, John D. Kubiatowicz. Erasure Coding vs. Replication: A quantitative Comparison. In Proc. of IPTPS, March 2002.

[10] Ben Y.Zhao, John Kubiatowicz, Anthony D.Joseph.Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing.Report No.UCB/CSD-01-1141, Computer Science Division (EECS) University of California. April 2001.