Master-Thesis

ByteStorm

Faires Mehrzweck-Content-Distribution-Protokoll

Tilo Eckert

Version 1.0 31. Januar 2013

Autor: Tilo Eckert

Prüfer: Prof. Alejandro P. Buchmann
Betreuer: Christof Leng

Fachbereich Informatik
Fachgebiet Datenbanken und
Verteile Systeme (DVS)

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 31. Januar 2013

____

Tilo Eckert

Inhaltsverzeichnis

1 Einleitung **7**

2 P2P Content Distribution **8**

2.1 CDN vs. P2P? . . . 8
2.2 BitTorrent . . . 9
2.3 BitTyrant . . . 10
2.4 FairTorrent . . . 11
2.5 Flower-CDN . . . 12
2.6 Swarming-Defizite in bestehenden P2P Content Distribution Systemen . . . 13
2.7 OneSwarm . . . 14
2.8 Fazit . . . 15

3 Transfer Encoding **17**

3.1 Store-and-Forward / keine Kodierung . . . 17
3.2 Forward Error Correction . . . 18
3.3 Erasure Codes . . . 18
3.4 Reed Solomon Codes . . . 19
3.5 Tornado Codes . . . 19
3.6 Fountain Codes . . . 20
3.7 LT-Codes . . . 21
3.8 Raptor Codes . . . 22
3.9 Network Coding . . . 22
3.10 Zusammenfassung . . . 24

4 Belohnungssysteme **26**

4.1 Eine Klassifizierung . . . 26
4.2 Tit-for-tat in BitTorrent . . . 28
4.3 eMule Belohnungssystem . . . 29
4.4 BAR Gossip . . . 30
4.5 Dandelion . . . 31
4.6 Karma . . . 32
4.7 e-cash . . . 34

4.8 One Hop Reputation . . . 36
4.9 Simple Trust Exchange Protocol (STEP) . . . 37
4.10 FairSwarm.KOM . . . 38
4.11 Zusammenfassung . . . 39

5 Anonymität **41**
5.1 Was ist Anonymität? . . . 41
5.2 Anonymität vs. Glaubhafte Abstreitbarkeit . . . 42
5.3 Survey zur Anonymität in verschienen Systemen . . . 42
5.4 BitTorrent + Tor = FAIL . . . 45
5.5 Tarzan . . . 47
5.6 Hordes . . . 47
5.7 PKI-loses Onion Routing . . . 48
5.8 Anonymes Routing und Churn . . . 50
5.9 Zusammenfassung . . . 50

6 ByteStorm **52**
6.1 Vorarbeiten . . . 52
6.1.1 BubbleStorm . . . 52
6.1.2 CUSP . . . 54
6.1.3 Content-Suche via BubbleStorm . . . 56
6.1.4 BitTorrent-Tracking via BubbleStorm . . . 57
6.2 Tempest . . . 58
6.2.1 Squall . . . 59
6.2.2 Blöcke & Prüfsummen . . . 62
6.2.3 Hash Tree . . . 63
6.2.4 Tempest Tree . . . 65
6.2.5 Squall-Tempest-Bindung . . . 68
6.2.6 Tempest-Integrität . . . 70
6.2.7 Annoncierung fertiger Tempest-Teile . . . 71
6.3 Tempest-Suche . . . 74
6.4 Tracking . . . 75
6.4.1 Modifikationen des Protokolls . . . 75
6.5 Datenübertragung . . . 78
6.5.1 Segmentauswahl-Strategie . . . 78
6.5.2 Upload-Strategie . . . 80
6.6 Anonymität . . . 81
6.6.1 Tunneltypen . . . 82

6.6.2 Nachrichtenformat 83
6.6.3 Tunnelaufbau 85
6.6.4 Tunnelabbau 88
6.7 Abonnements / Feeds 88
6.8 HTTP-Gateways aka. ByteStorm CDN 89
6.9 Das Protokoll 92
6.9.1 Protokoll-Nachrichten 93

7 Kurzevaluierung & Fazit **103**
7.1 Kurzevaluierung 103
7.2 Zusammenfassung 104
7.3 Ausblick 105

8 Glossar **106**

9 Abbildungsverzeichnis **109**

10 Algorithmenverzeichnis **110**

11 Literaturverzeichnis **111**

Zusammenfassung

Bestehende Content Distribution Systeme sind meist auf eine bestimmte Aufgabe zugeschnitten. Unterschiedliche Anwendungsszenarien erfordern unterschiedliche Softwarelösungen, die jeweils separat betrieben werden müssen. ByteStorm ist ein von Grund auf entwickeltes Protokoll und Content Distribution System auf Peer-to-Peer-Basis, das es zum Ziel hat, vielseitig einsetzbar, für Endanwender leicht benutzbar und an die verfügbaren Ressourcen anpassbar zu sein. ByteStorm kann beispielsweise eingesetzt werden, um Software, Medieninhalte, signierte Software-Updates und wiederkehrende Inhalte beliebig skalierbar via P2P zu verteilen oder automatisiert Updates in größeren Infrastrukturen einzuspielen. Es ist auch möglich, ein klassisches Content Delivery Network (CDN) für HTTP-Downloads auf der Basis von ByteStorm aufzubauen, das die Last autonom auf die verfügbaren Knoten verteilt. CDN-Knoten können von beliebigen Entitäten bereitgestellt und jederzeit hinzugefügt oder entfernt werden. Eine Anonymisierungsfunktion erlaubt es Peers, ihre Identität zu verbergen. Eine integrierte dezentrale Suchfunktion macht das Finden von Inhalten einfach.

Abstract

Existing content distribution systems are usually fitted for one particular task. Different use cases require different software solutions, each being maintained separately. ByteStorm is a protocol and content distribution system developed from scratch based on peer-to-peer networks. It aims at versatile utility, being easy to use by end-users and adoptability to available resources. For example, ByteStorm can be used to distribute software, media content, signed software updates and recurring content scalable via P2P or to deploy updates in large environments in an automated fashion. It is also possible to build a classical Content Delivery Network (CDN) for downloads via HTTP based on ByteStorm, with autonomous load balancing across available nodes. Nodes can be provided by any entity and can be added or removed at any time. An anonymization feature allows peers to hide their identity. The integrated search functionality makes finding content easy.

1 Einleitung

Es gibt bereits eine ganze Reihe von Content Distribution Systemen. Viele davon arbeiten auf der Basis von Peer-to-Peer, also der direkten Kommunikation zwischen Teilnehmern. Jedes dieser Systeme befriedigt einen bestimmten Einsatzzweck. Dazu gehören unter anderem das Verteilen großer und kleiner Datenmengen, Software-Deployment, Software-Updates, regelmäßig wiederkehrende Inhalte wie Podcasts, Finden von Inhalten, Load Balancing, Hochverfügbarkeit von Daten, Anonymität und weitere. Für jeden dieser Einsatzzwecke gibt es Softwarelösungen, die das jeweilige Problem lösen. Es gibt jedoch keine auf Peer-to-Peer basierende Lösung, die für all diese Aufgaben gleichermaßen geeignet ist und darüber hinaus für Benutzer leicht zu bedienende Endprodukte ermöglicht. In diese Lücke tritt ByteStorm.

Bei ByteStorm handelt es sich um ein Kommunikationsprotokoll für den Austausch von Dateien, das dem von BitTorrent nicht gänzlich unähnlich ist. ByteStorm ist jedoch um einige zusätzliche Funktionen erweitert, deren Ziel es ist, geschickt kombiniert, alle oben genannten Einsatzszenarien bedienen zu können.

Die zu übertragenden Dateien werden durch den Einsatz von PKI und anderen kryptographischen Verfahren vor Manipulation geschützt. Optional kann die Authentizität der Daten über eine Vertrauenskette bis hin zu einzelnen Datenblöcken verifiziert werden, wodurch ByteStorm beispielsweise für das Verteilen signierter Software(-Updates) geeignet ist. Mehrere Dateien können zu einer Einheit, Tempest genannt, zusammengefasst und verteilt werden. Diese lassen sich über eine integrierte Suchfunktion leicht finden. Durch Abonnements können wiederkehrende Inhalte vollautomatisch heruntergeladen werden. Mittels eines integrierten HTTP-Gateways mit Load Balancing Funktion können via ByteStorm auf mehrere Knoten verteilte Inhalte für Downloads über das Web eingesetzt werden. So lässt sich ein dynamisch skalierbares Content Delivery Network (CDN) aufbauen, um HTTP-Downloads über eine verteilte Infrastruktur anzubieten. ByteStorm bietet außerdem die Möglichkeit, anonymisierte Downloads durchzuführen. Dazu werden bekannte Tunneltechniken eingesetzt, die eine Identifikation der Beteiligten erschweren.

Die Kapitel 2 bis 5 untersuchen anhand bestehender Systeme die Themen P2P Content Distribution, Transfer Encoding, Belohnungssysteme und Anonymität. Dies diente der Ideenfindung für das neue Protokoll. ByteStorm wird in Kapitel 6 ausführlich vorgestellt.

2 P2P Content Distribution

In diesem Kapitel wird die Abgrenzung zwischen P2P Content Distribution und herkömmlicher Content Distribution beschrieben sowie der aktuelle Stand von P2P Content Distribution Systemen untersucht. Es werden einige Systeme mit unterschiedlichen Zielsetzungen betrachtet, um einen Überblick über Umsetzungsmöglichkeiten und Ideen zu gewinnen, die bei der Entwicklung eines vielseitig einsetzbaren Content Distribution Systems wie ByteStorm hilfreich sein können.

2.1 CDN vs. P2P?

Klassische Content Delivery Netzwerke (CDN) bestehen aus meist weltweit verteilten Server-Farmen, die Benutzer mit populärem Content versorgen. Der Content stammt dabei in der Regel von einem Server in der Nähe des Benutzers, um geringe Latenzen und hohe Bandbreiten zu erreichen. Um die Überlastung einzelner Server des CDNs zu vermeiden, muss zusätzliches Load Balancing betrieben werden. Die dafür nötige, teure Infrastruktur wird oft von einem Provider betrieben, der sich auf Content Delivery spezialisiert hat und den Service zahlenden Kunden anbietet. Für populären Content können auf den Anbieter erhebliche Kosten zukommen. Kleinere CDNs haben bei so genannten Flash Crowds (sehr viele Benutzer versuchen plötzlich gleichzeitig denselben Content abzurufen) ein Skalierungsproblem, da ihre maximale Belastbarkeit beschränkt ist. Desweiteren kommt es immer wieder vor, dass CDNs zeitweise nicht erreichbar sind, sei es wegen Angriffen, Stromausfällen oder Hardwarefehlern.

Als Gegenpol zu den aufwendigen Infrastrukturen klassischer CDNs sind P2P Content Delivery Netzwerke entstanden. Sie nutzen die Bandbreite aller Benutzer, die am Content interessiert sind, um ihn zu verteilen. P2P CDNs skalieren mit der Anzahl ihrer Benutzer, da jeder, der etwas herunterlädt, die Daten gleichzeitig auch anderen anbietet. Der Anbieter kann weitgehend auf kostspielige Infrastrukturen verzichten, da er nur mindestens eine Kopie des Contents verteilen muss, um potenziell tausende von Benutzern zu versorgen.

P2P und klassische CDNs schließen sich nicht gegenseitig aus. Aufgrund der Tatsache, dass die Internetanbindungen der meisten Benutzer asymmetrisch ist, sie also schneller herunter- als hochladen können, erreichen reine P2P-Netze meist nicht die Performanz klassischer CDNs. Daher gibt es seit einigen Jahren einen Trend in Richtung hybrider Lösungen, bei denen beide

Techniken parallel eingesetzt werden (z.B. Akamai, Pando Networks). Wird die Downloadbandbreite durch P2P-Traffic nicht vollständig ausgenutzt, wird ein Teil der Daten zusätzlich von CDN-Servern heruntergeladen. Für diese Kosten/Nutzen-Optimierung müssen sich die Benutzer jedoch (genau wie bei reinen P2P CDNs) typischerweise ein Programm herunterladen, das den Downloadvorgang steuert. Für die verteilte Bereitstellung von statischen Inhalten auf Webseiten (Grafiken, CSS-Dateien, Skripte, Videos) eignet sich diese Lösung daher nicht.

2.2 BitTorrent

BitTorrent, das wohl meistbekannte P2P CDN, verteilt Dateien und ganze Verzeichnisse zwischen den Teilnehmern. Als Torrent wird sowohl eine Metadatei bezeichnet, die die Inhalte beschreibt, als auch die mit ihrer Hilfe per BitTorrent übertragenen Inhalte selbst. Zur besseren Unterscheidung wird die Metadatei in dieser Arbeit als Torrent-Datei bezeichnet. Die Torrent-Datei ist wenige Kilobytes groß und enthält Metainformationen über die zu verteilenden Dateien, Prüfsummen sowie Tracker-Adressen. Jedes Torrent ist durch einen so genannten Infohash eindeutig identifizierbar. Die Menge der aktiven Teilnehmer, die ein Torrent herunterladen oder verteilen, wird als Schwarm bezeichnet. Jeder Teilnehmer ist entweder ein Seeder oder ein Leecher. Seeder besitzen die vollständige Datei und wirken durch Hochladen von Daten an der Verteilung mit. Leecher versuchen das Torrent vollständig herunterzuladen, laden aber gleichzeitig bereits fertiggestellte Teile zu anderen Leechern hoch. Ein Tracker sorgt dafür, dass sich die Teilnehmer eines Schwarms gegenseitig kennenlernen. Der BitTorrent-Client holt sich, unter Verwendung des Infohashes, eine Liste von Peers vom Tracker und registriert sich damit gleichzeitig, um von anderen Peers als möglicher Tauschpartner gefunden zu werden. Mehrere Erweiterungen machen den zentralen Tracker jedoch überflüssig, indem Peers über eine verteilte Hash-Tabelle (DHT) oder durch den Austausch mit bereits bekannten Peers aus dem Schwarm eines Torrents gefunden werden.

Ein Problem, das der Funktionsweise des BitTorrent-Protokolls inhärent ist, ist das Free Riding, also das Herunterladen von Torrents ohne etwas für deren Verteilung beizusteuern. Zwar wird versucht, mit dem Tit-for-tat Ansatz einen Anreiz zu schaffen, an der Verteilung des Torrents durch Bereitstellen von Upload-Bandbreite mitzuwirken, gleichzeitig gibt es aber Optimistic Unchokes, welche ein standardkonformer Client alle 30 Sekunden durchführt, indem er zu einem Peer Daten hochlädt, ohne zuvor von ihm Daten erhalten haben zu müssen. Dies soll dazu dienen, neu hinzugekommenen Peers zu ihren ersten vollständigen Pieces (Dateiteilen) zu verhelfen, die sie dann verteilen können. In mehreren Arbeiten wird diese Protokollfunktion ausgenutzt, um Free Riding zu betreiben. In [FRC06] wird ein BitTorrent-Client vorgestellt, der niemals Daten hochlädt und sich den Unchoke-Mechanismus zunutze macht, indem er deutlich mehr und deutlich schneller Verbindungen zu anderen Peers aufbaut, als ein standardkonformer

Client. Mit der Anzahl der Verbindungen steigt auch die Wahrscheinlichkeit eines Optimistic Unchoke durch einen der verbundenen Peers und somit die Wahrscheinlichkeit Daten zu erhalten. Das Experiment ergab, dass der Free Riding Client das Torrent oft ähnlich schnell fertiggestellt hat, als Peers, die Daten hochladen. In einem ähnlichen Experiment, das in [FRB07] beschrieben wird, versucht der Free Riding Client, durch Vorspielen mehrerer Identitäten gegenüber dem Tracker, die Adressdaten von möglichst allen Peers im Schwarm des Torrents zu bekommen, baut eine Verbindung zu allen Peers auf und nutzt Optimistic Unchokes auf dieselbe Weise aus, um das Torrent zu komplettieren. Die eingesetzten konformen Clients verwenden eine erweiterte Version von Optimistic Unchokes. Sie berücksichtigen zusätzliche Kriterien wie die Zeit, die ein Peer sich bereits im Choke-Zustand befindet und wie viele Daten der Peer bereits an den Client gesendet hat. Trotzdem konnten Free Rider Torrents ähnlich schnell oder sogar früher fertigstellen als konforme Clients. Dies gilt insbesondere dann, wenn viele Seeder vorhanden sind, da diese Peers, die Daten von ihnen erhalten, nicht anhand der Übertragungsrate der zu ihnen gesendeten Daten auswählen (können), sondern danach, welche Peers vom Seeder am schnellsten herunterladen. Der durch Tit-for-tat beabsichtigte scheinbare Nachteil für Freerider lässt sich also relativ leicht umgehen.

2.3 BitTyrant

Durch Optimistic Unchokes erhalten langsame Peers in BitTorrent trotz ihrer langsamen Uploadbandbreite im Mittel schneller Daten, als sie senden können. Dies führt dazu, dass Peers umso altruistischer handeln, also weniger Nutzen aus ihrem Beitrag am Schwarm ziehen, je größer ihre bereitgestellte Uploadbandbreite ist. In ihrer Arbeit [IBR07] gelangen die Autoren zu dem Schluss, dass BitTorrent, nach der Spezifikation implementiert, unfair arbeitet und dass die Mehrheit von Benutzern davon profitiert, da sie eine vergleichsweise langsame Internetverbindung besitzen. Um dem entgegenzusteuern, wird BitTyrant vorgestellt, ein modifizierter BitTorrent Client, der zum BitTorrent-Protokoll kompatibel ist, aber seine Uploadbandbreite strategisch einsetzt, um die eigene Downloadgeschwindigkeit zu maximieren. BitTyrant nutzt aus, dass Peers mit hinreichend kleiner Bandbreite nicht in der Lage sind, die erhaltene Bandbreite zu erwidern und dass Peers mit schneller Anbindung meist deutlich schneller senden können, als sie Daten empfangen. Dazu wird versucht, die von Peers auf eigene Upload-Vorgänge erwiderte Bandbreite zu maximieren und gleichzeitig die eigene Uploadbandbreite zu ihnen zu minimieren, sowie die Anzahl gleichzeitiger Downloads von Peers durch Anpassung der Anzahl gleichzeitiger Uploads zu maximieren. Mit anderen Worten sendet BitTyrant gerade genug Daten an andere Peers, sodass diese nicht aufhören, Daten in Erwiderung an den eigenen Client zu senden. Mit dieser Strategie können Peers mit hoher Bandbreite, ohne signifikante Einbußen der eigenen Performanz, ihren Beitrag zum Schwarm reduzieren bzw. optimieren. BitTyrant implementiert diese Strategie, indem für jeden Peer p eine Schätzung u_p der minimal

beizusteuernden Bandbreite gespeichert wird, die nötig ist, damit der Peer Bandbreite erwidert und die Geschwindigkeit d_p mit der dieser zuletzt gesendet hat. Das Verhältnis dieser beiden Werte bestimmt, in welcher Reihenfolge BitTyrant Peers für Unchokes auswählt, also Daten sendet. Unchokes passieren nur solange, bis die Summe der u_p die verfügbare Uploadbandbreite überschreitet. u_p wird bei jeder Choke/Unchoke Runde angepasst. Wenn ein Peer über mehrere Runden Bandbreite auf die eigene erwidert, wird die eigene Uploadbandbreite gesenkt. Hört der Peer auf, Bandbreite zu erwidern, wird u_p erhöht. d_p ist entweder die aktuellste Messung der Downloadrate vom Peer oder, falls noch kein Download von dem Peer stattfand, eine Schätzung anhand der Geschwindigkeit, in der der Peer die Fertigstellung von Pieces kundtut. Wenn BitTyrant nur von wenigen Peers im Schwarm eines Torrents genutzt wird, dann lässt sich die Downloadzeit laut Arbeit signifikant reduzieren, sodass Downloads meist deutlich schneller abgeschlossen werden, als mit nichtstrategischen Clients. Selbst wenn alle Peers BitTyrant nutzen, reduziert sich die durchschnittliche Downloadzeit, insbesondere weil Peers mit hoher verfügbarer Bandbreite diese früher voll ausnutzen und altruistisch Bandbreite beitragen können.

2.4 FairTorrent

Viele Ansätze zur Verbesserung der Fairness in BitTorrent (z. B. die zuvor beschriebenen) setzen auf eine Schätzung der zukünftigen Transferrate unter Verwendung der Transferrate in der aktuellen Runde und setzen somit auf Heuristiken, die auf Annahmen basieren. Dies lässt sich ausnutzen, um Freeriding auf Kosten von ehrlichen Benutzern zu betreiben, die Bandbreite bereitstellen oder um den eigenen Profit mit strategischen Clients zu maximieren. Das Paper [FTP09] stellt einen simplen und fairen Ansatz für die Auswahl der als nächstes zu sendenden Chunks an andere BitTorrent-Peers unter dem Namen FairTorrent vor. FairTorrent sendet den nächsten Chunk an Daten einfach an den Peer, dem er am meisten Daten schuldet. Dafür wird lediglich die Differenz aus hoch- und heruntergeladener Datenmenge für jeden Peer benötigt. Da der Empfänger des nächsten Chunks ausschließlich von den Defiziten gegenüber allen verbundenen Peers abhängt, wird der (Un)Choke-Mechanismus von BitTorrent nicht benötigt. Daher sendet FairTorrent jedem verbundenen Peer eine Unchoke-Nachricht, damit dieser Anfragen nach Chunks stellt. Das Verfahren ergibt natürlich nur Sinn, solange der Client als Leecher agiert, da der Client nach Fertigstellung des Torrents keine Blöcke mehr herunterlädt. Als Seeder sendet FairTorrent Blöcke per Round-Robin gleichmäßig an alle interessierten Peers, um die Wahrscheinlichkeit für jeden Leecher zu erhöhen, Pieces zu haben, die er tauschen kann. Dies ist auch die einzige Möglichkeit, um als neuer Peer, der noch keine Daten zum Verteilen hat, an erste Pieces zu kommen, wenn kein Leecher freie Kapazitäten hat. Die Autoren halten es für möglicherweise nützlich, eine zusätzliche Bootstrapping-Möglichkeit einzuführen (z.B. im Falle

eines sehr starken Seeder/Leecher-Missverhältnisses), hielten dies in ihren Experimenten aber nicht für notwendig.

Der Client arbeitet, so wird argumentiert, ständig der Unfairness entgegen, auf eine Konvergenz gegen eine faire Bandbreitenverteilung hin und ist resistent gegen strategische Peers. Ansätze, die die Herstellung von Fairness auf der Uploadrate anderer basieren, scheitern in der Praxis daran, dass die Uploadrate eines Peers stark variiert, ein Peer viele Choke/Unchoke-Runden mit nicht erwiderter Bandbreite vergeudet oder versucht, Peers mit ähnlicher Geschwindkeit zu finden.

Strategische Peers erlangen gegenüber FairTorrent zwar keine Vorteile, allerdings ist Whitewashing möglich, also das Verlassen des Netzwerks durch einen Freerider, nachdem er ein Defizit gegenüber anderen Peers aufgebaut hat, und anschließendem Beitritt unter einer neuen Identität. In Experimenten haben die Autoren eine schnelle Konvergenz zwischen der Uploadrate eines Leechers und der Downloadrate von anderen Leechern und kürzere minimale und maximale Downloadzeiten festgestellt, als mit dem originalen BitTorrent-Client oder BitTyrant. Im Gegensatz zu diesen hängt die Downloadrate bei FairTorrent proportional von der eigenen Uploadrate ab. Peers mit hoher Uploadrate werden durch FairTorrent also nicht stark benachteiligt, sondern profitieren von ihrer höheren bereitgestellten Bandbreite in angemessener Weise.

2.5 Flower-CDN

Flower-CDN, beschrieben in [FLO09] und [FLO11], ist ein Peer-to-Peer CDN für die Verteilung von Content von Websites. Beim Caching der Inhalte wird die Lokalität der Peers innerhalb des Netzwerks berücksichtigt. Die eigene Lokalität wird durch Messung der Latenz zu bekannten Knoten bestimmt und der Peer, basierend auf dem Ergebnis, in eine von k Lokalitäten einsortiert. Peers mit derselben Lokalität und einem Interesse an denselben Content organisieren sich in unstrukturierte Cluster namens Petals. Petals tauschen Informationen per Gossiping über Content und Kontakte untereinander aus. Ein Peer in jedem Petal übernimmt die Rolle eines Directory Peers. Er kennt alle Peers in seinem Petal und ihren gespeicherten Content. Der Directory Peer ist auch Teilnehmer einer D-ring genannten DHT. Will ein Client den Content einer Website abrufen, dann sucht er beim ersten Zugriff in D-ring nach einem Directory Peer, der für die Website zuständig ist und seiner Lokalität möglichst nahe ist. Der gefundene Directory Peer wird kontaktiert, welcher die Anfrage an einen Peer in seinem Petal weiterleitet, der die Anfrage befriedigen kann. Wenn der Client willens ist, Speicherplatz für die Website bereitzustellen, speichert er den erhaltenen Content und tritt dem Petal bei, aus dem er den Content heruntergeladen hat. Gibt es keinen Directory Peer für das Website-Lokalität-Tupel, wird der Peer selbst zum Directory Peer und tritt dem D-ring DHT bei. D-ring ist kein komplett eigenständiges DHT, sondern setzt auf strukturierte Overlays wie Chord oder Pastry auf. Es wird lediglich das Mapping von Websites

und Lokalitäten auf Hashes beschrieben. Letztere bestehen einfach aus einem Hash der Website ID (z.B. des Hostnamens) konkateniert mit der ID der Lokalität. Dadurch sind Petals mit unterschiedlichen Lokalitäten, die für eine Website zuständig sind, Nachbarn im ID-Raum.

Flower-CDN ist für den Einsatz im Web konzipiert, wo Dateien eine relativ geringe Größe haben. Dies spiegelt sich darin wieder, dass Dateien erst nach ihrem vollständigen Download bereitgestellt werden. Mit dem Directory Peer gibt es zudem einen Super-Peer pro Petal, der alle Anfragen an dieses Petal angegen nimmt. Da es keine Auswahlstrategie für diesen gibt, sondern einfach der erste Knoten im Petal zum Directory Peers wird, ist er gleichzeitig der Flaschenhals für dieses Petal, der Anfragen möglicherweise nicht schnell genug weiterleiten kann. Auf der anderen Seite führt das geographische Gruppieren von Knoten zu kürzeren Übertragungswegen und damit kürzeren Latenzen sowie potenziell schnelleren Downloads.

2.6 Swarming-Defizite in bestehenden P2P Content Distribution Systemen

Wie bereits erwähnt, bezeichnet der Begriff Schwarm (engl. swarm) die Gesamtheit der Knoten in einem P2P-System, die zusammenarbeiten, um einen bestimmten Content (z.B. eine Datei) untereinander zu verteilen. Jeder Teilnehmer eines Schwarms stellt den anderen Teilnehmern einen Teil seiner Ressourcen (Speicherplatz, Bandbreite) zur Verfügung, damit alle den gewünschten Content möglichst schnell vollständig herunterladen können. Bei den meisten bestehenden Systemen werden die Ressourcen nur innerhalb eines Schwarms geteilt. Wechselwirkungen zwischen Schwärmen gibt es nur dann, wenn der Benutzer sich für den Content mehrerer Schwärme interessiert, wodurch die lokal verfügbare Bandbreite zwischen diesen aufgeteilt wird. Dies führt beispielweise bei BitTorrent-Netzwerken oft dazu, dass Peers, die wenig nachgefragte Torrents bereitstellen, kaum bis gar nicht ausgelastet sind, während Peers in stark nachgefragten Torrents überlastet sind.

Wenn für populären Content genügend Bandbreite vorhanden ist, ist die Uploadbandbreite von Peers in der Regel nicht voll ausgelastet. In diesem Fall kann der Peer zur Maximierung der Performanz des Gesamtsystems beitragen, indem er Bandbreite für anderen Content bereitstellt, an dem er eigentlich nicht interessiert ist. Es kann jedoch passieren, dass ein Peer, der Content herunterlädt, nur um ihn weiterzuverteilen, die Performanz einzelner Schwärme – durch den Verbrauch von Uploadbandbreite anderer Peers – senkt, wenn die Strategie für den Einstieg in eigentlich uninteressante Schwärme falsch gewählt wird. In [SOC12] werden einige Strategien aus der Literatur zusammengefasst, die entweder unpopulären Content besser verfügbar machen oder die Downloadzeit reduzieren sollen:

Um die Verfügbarkeit unpopulären Contents zu verbessern, wird am Beispiel BitTorrent vorgeschlagen, mehrere Dateien verschiedener Schwärme zu bündeln und diese Bündel zu verteilen.

Die Idee dabei ist, dass durch das Bündeln jeder Peer auch Dateien herunterlädt, an denen er eigentlich nicht interessiert ist, und so die Wahrscheinlichkeit der Nichtverfügbarkeit unpopulärer Dateien verringert.

Ein Ansatz zur Reduktion der durchschnittlichen Downloadzeit basiert auf dem eben erwähnten Prinzip. Dabei lädt jeder Peer nur die Chunks des Bundles, die zu den Dateien gehören, die ihn interessieren, falls deren Verfügbarkeit schlecht ist oder ihm keine Dateien bekannt sind, die schlecht verfügbar sind. Wenn uninteressante Dateien in anderen Schwärmen schlecht verfügbar sind und interessante Dateien eine ausreichende Verbreitung haben, dann wechselt der Peer in einen sozialeren Modus. Er lädt dann Chunks anderer Schwärme herunter und stellt sie bereit.

Eine weitere vorgestellte Lösung nutzt den BitTorrent Tracker zur Lastverteilung. Jeder Peer sendet dem Tracker bei jeder Anfrage nach neuen Peers seine eigene Downloadstatistik. Aus den gesammelten Daten aller Peers kann der Tracker ermitteln, ob der Schwarm selbst genügend Kapazitäten aufbringen kann, um alle Downloadanfragen zu bewältigen. Wenn nicht, weist der Tracker Peers anderer Schwärme, die noch freie Uploadkapazitäten haben, diesem Schwarm zu, um ihn zu unterstützen.

2.7 OneSwarm

Peer-to-Peer Content Distribution Protokolle bieten in der Regel entweder hohe Performanz bei geringer Anonymität oder hohe Anonymität bei deutlich reduzierter Performanz (TOR, Freenet, etc). OneSwarm, beschrieben in [PPP10], soll einen Kompromiss zwischen diesen beiden Polen liefern, der also sowohl einen gewissen Grad an Privatsphäre, als auch akzeptable Downloadgeschwindigkeiten liefert. Der Benutzer kann dabei den Grad der Privatsphäre selbst bestimmen, indem er Vertrauensbeziehungen eingeht oder pauschal allen Peers vertraut und definiert, wer auf welchen Content Zugriff hat. Peers in OneSwarm sind gruppiert in solche, denen der Benutzer vertraut und alle anderen, den unvertrauten Peers. Jeder Knoten wird durch einen Public Key identifiziert. Dieser wird zum Eingehen von Vertrauensbeziehungen und zum Herstellen verschlüsselter Verbindungen benötigt. Um einen Benutzer als vertrauten Kontakt aufzunehmen, muss der öffentliche Schlüssel bekannt sein. Für den Schlüsselaustausch kennt OneSwarm vier Methoden:

1. über das lokale Netzwerk (LAN)
2. über existierende soziale Netzwerke
3. per Email
4. über Community Server

Community Server sind subskriptionsbasierte Schlüsselserver. Es gibt sowohl private als auch öffentliche Community Server für den Schlüsselaustausch. Private Community Server werden zur Vereinfachung des Schlüsselaustauschs innerhalb einer Gruppe von Benutzern verwendet, die sich untereinander vertrauen. Neue Benutzer melden ihren Public Key beim Schlüsselserver an, von wo ihn andere Benutzer der Gruppe abfragen können. Öffentliche Community Server dienen als Bootstrap für das Finden von unvertrauten Knoten, die als Zwischenknoten indirekter Verbindungen dienen können.

Benutzer werden anhand von Identitäten verwaltet. Jedoch kann die IP-Adresse des Benutzers sich jederzeit ändern. Das Finden von IP/Port zu einer Identität (Public Key) erfolgt via DHT, wobei diese Information und ein symmetrischer Schlüssel für jeden vertrauten Benutzer mit dessen Public Key separat verschlüsselt wird, damit nicht jeder, der den Public Key eines Benutzers kennt, ständig die aktuelle IP abfragen und den Benutzer überwachen kann.

Direkte Verbindungen und Datenübertragungen finden nur zwischen vertrauten Knoten statt. Alle anderen Verbindungen werden durch beliebig lange Pfade getunnelt, welche sich aus dem Suchmechanismus ergeben. Die Suche nach Content erfolgt per Flooding durch das Netzwerk entlang der Kanten von Vertrauensbeziehungen, wobei für jeden verbundenen Peer probabilistisch entschieden wird, ob die Nachricht an diesen weitergeleitet wird. Datenübertragungen nehmen den umgekehrten Pfad zum Initiator der Suche. Dadurch bleiben Sender und Empfänger anonym. Um Timing-Angriffe zu erschweren, werden Nachrichten zwischen unvertrauten Knoten künstlich verzögert. Die eigentliche Datenübertragung erfolgt per BitTorrent, jedoch über die bei der Suche aufgespannten Pfade anstatt per Direktverbindung. Um überladene und langsam angebundene Knoten zu umgeben, werden mehrere Verbindungen über unterschiedliche Pfade zum Ziel aufgebaut.

OneSwarm ist resistent gegen Timing Attacken und gegen Angreifer, die nur einige wenige Knoten kontrollieren. Ist ein Angreifer in der Lage, sich mittels mehrerer kollaborierenden Knoten mit einem anzugreifenden Knoten zu verbinden, so ist es möglich, den Knoten als Datenquelle zu identifizieren. Dem kann sich der Benutzer aber entziehen, indem er keine öffentlichen Community Server benutzt und keine Nachrichten an nicht vertraute Kontakte weiterleitet. Die Performanz von Downloads zwischen direkt verbundenen Knoten ist ähnlich der von BitTorrent und bei indirekten Verbindungen deutlich schneller als Tor (3,4-fach) oder Freenet (6,9-fach).

2.8 Fazit

Die untersuchten Systeme erfüllen jeweils eine ganz spezifische Aufgabe. BitTorrent verteilt große Datenmengen ohne zentrale Instanz in kurzer Zeit zwischen allen Interessenten. Dabei treten in der Praxis jedoch Probleme wie Freeriding, unfaires Verhalten und, mangels vollstän-

diger Quellen (Seeder), nicht verfügbare Downloads auf. Flower CDN ist ein CDN für die Bereitstellung von Webseiten, also insbesondere kleinen Dateien, wobei Peers, die eine Webseite anfragen, automatisch Teil des CDNs für diese Webseite werden. Dies setzt eine entsprechende Software auf der Client-Seite voraus. In OneSwarm besitzt jeder Peer eine Identität und kann Vertrauensbeziehungen eingehen. Nur mit vertrauten Peers wird direkt kommuniziert. OneSwarm setzt dabei auf PKI, wobei der nötige Schlüsselaustausch umständlich manuell durchgeführt oder über einen zentralen Server abgewickelt werden muss, welcher verfügbar sein muss. Jede dieser Lösungen ist für ihr vorgesehenes Einsatzszenario gut geeignet. Mit Ausnahme von Flower CDN ist für die Nutzung die Installation einer entsprechenden Client-Software erforderlich. Keines der Systeme liefert jedoch eine integrierte und benutzerfreundliche Lösung für die Verteilung authentischer, großer und kleiner Daten über mehrere Verbreitungskanäle, Unterstützung für wiederkehrende Inhalte, eine faire Bandbreitenzuteilung und einen vom Benutzer bestimmten Grad an Anonymität.

3 Transfer Encoding

In BitTorrent ist die Verfügbarkeit von Torrents oft ein Problem. In der Regel gilt, dass die Abwesenheit von Seedern zur Folge hat, dass niemand dieses Torrent mehr fertigstellen kann. Es gibt gelegentlich Situationen, in denen die Mehrheit der Blöcke eines Torrents noch im Schwarm vorhanden ist. Fehlt ein Block, kann der Torrent dennoch nicht vervollständigt werden. Es gibt jedoch Algorithmen auf der Grundlage fehlerkorrigierender Übertragungsverfahren, mit denen sich fehlende Datenblöcke widerherstellen lassen, sofern ausreichend Redundanzinformationen vorhanden sind. Derartige Verfahren werden in diesem Kapitel auf ihre Eignung für den Einsatz in einem Content Distibution System wie ByteStorm untersucht.

3.1 Store-and-Forward / keine Kodierung

Der Begriff Store-and-Forward bezeichnet eine Datenübertragungsart aus der Netzwerktechnik. Pakete werden von Zwischenstation zu Zwischenstation weitergeleitet. Dies geschieht aber mit einer zeitlichen Verzögerung, da jede Zwischenstation immer solange wartet, bis sie das komplette Paket empfangen hat. Dann prüft sie die Integrität des Pakets anhand einer Prüfsumme. Bei Misserfolg wird das Paket verworfen, bei Erfolg weitergeleitet. Zwischenstationen modifizieren das Paket nicht, sondern senden es unverändert weiter (abgesehen vom TTL-Zähler). Analog dazu verhalten sich die meisten P2P Content Distribution Systeme. Lediglich die Pakete sind größer, entweder ganze Dateien oder Chunks davon. Knoten erhalten diese, prüfen die Integrität und leiten sie an andere Knoten weiter. Im Unterschied zur Netzwerktechnik geschieht die Weiterleitung allerdings in der Regel auf Anfrage. Die Dateidaten werden für die Übertragung nicht verändert. Es findet also keine spezielle Kodierung statt, die die Übertragung effizienter machen könnte. Insbesondere wird von den meisten Systemen nichts unternommen, um im Falle fehlerhafter Daten die erneut zu übertragende Datenmenge zu minimieren. Ergibt beispielsweise in BitTorrent die Integritätsprüfung, dass ein Chunk fehlerhaft ist, so muss der komplette Chunk erneut übertragen werden, auch wenn nur wenige Bytes fehlerhaft sind. Für Benutzer mit geringer Bandbreite oder begrenztem Datenvolumen ist dies von Nachteil. Andererseits wird für eine erneute Übertragung kaum Rechenzeit benötigt, wenn keine Kodierung stattfindet, was aufgrund begrenzter Rechen- und Akkuleistung insbesondere für mobile Geräte ein Vorteil ist.

3.2 Forward Error Correction

Forward Error Correction (FEC) ist eine Kodierungstechnik zur Erkennung und Korrektur einer begrenzten Anzahl Fehler in einem Datenstrom bzw. in Datenblöcken. Dazu wird den Nutzdaten mittels eines Codes Redundanz hinzugefügt. Bestehen die zu übertragenden Nutzdaten aus k Blöcken und werden per FEC weitere n Blöcke Redundanz hinzugefügt, dann können, je nach Code, bis zu n Übertragungsfehler korrigiert werden. Abhängig vom verwendeten Code kann es ausreichen, k der $k + n$ Blöcke korrekt zu empfangen, um die Nutzdaten wiederherstellen zu können. Vom gewählten Code ist auch abhängig, ob die unmodifizierten Nutzdaten Teil des Ausgabestroms sind oder nicht. FEC wird eingesetzt, wenn eine erneute Übertragung zu teuer oder unmöglich ist, z. B. bei Übertragungen per Multicast, und zur Fehlerkorrektur bei Speichermedien. In P2P Content Distribution kann FEC benutzt werden, um die Robustheit des Systems, die Datenverfügbarkeit und die Datenzugriffszeit zu reduzieren [FEC02]. Es wurde allerdings lediglich der Fall untersucht, in dem ein Dateiblock auf einmal heruntergeladen wird und nicht der Fall von parallelen Downloads, wie in aktuellen P2P Systemen üblich. Die zahlreichen Arbeiten zu FEC im P2P-Sektor beschränken sich im Wesentlichen auf Media Streaming.

3.3 Erasure Codes

Erasure Codes sind eine Klasse von Forward Error Correction Codes. Sie erhalten eine Nachricht bestehend aus k Symbolen als Eingabe und transformieren sie in eine Ausgabe aus m Symbolen, wobei $m > k$. Die Originalnachricht lässt sich aus einer Untermenge der m Symbole rekonstruieren. Man spricht von einem optimalen Erasure Code, wenn jede beliebige Kombination von k der m kodierten Symbole ausreicht, um die ursprüngliche Nachricht wiederherzustellen. Für große m sind optimale Codes kostspielig, da sie meistens eine quadratische Komplexität beim Kodieren bzw. Dekodieren aufweisen. Ein simples Beispiel für optimale Codes ist die Berechnung von Parität, bei der aus k Eingabesymbolen durch aufsummieren ein Paritätssymbol berechnet wird, also $m = k + 1$. Geht eines der k Eingabesymbole verloren, kann es durch aufsummieren aller übrigen Symbole (einschließlich der Parität) rekonstruiert werden. Da es unerheblich ist, welches der m Symbole verloren geht, heißt der Code optimal. Beinahe optimale Erasure Codes erfordern $k + \varepsilon$ Symbole für die Wiederherstellung, wobei $\varepsilon > 0$ ist. Es gilt die Faustregel, dass ein beinahe optimaler Code umso rechenintensiver ist, je kleiner ε sein soll. [1]

[1] http://en.wikipedia.org/w/index.php?title=Erasure_code&oldid=470434981 (abgerufen am 07.06.2012)

3.4 Reed Solomon Codes

Reed Solomon Codes sind fehlerkorrigierende Codes, die insbesondere gegen Burstfehler gut geeignet sind. Burstfehler sind Fehler, die mehrere aufeinander folgende Bits in einem Block von Daten betreffen und häufig in physische Übertragungsmedien auftreten. Reed Solomon Codes arbeiten auf Symbolen, welche üblicherweise 8 Bit lang sind. Durch Hinzufügen von n redundanten Symbolen können bis zu n verlorene oder fehlerhafte Symbole wiederhergestellt werden. Die Symbole aus den Eingabedaten werden von Reed Solomon Codes als Koeffizienten eines Polynoms $p(x)$ über einem endlichen Körper $GF(q)$ betrachtet. Redundante, kodierte Symbole werden in heutigen Verfahren berechnet, indem das Polynom $p(x)$ mit einem zyklischen Generatorpolynom $g(x)$ multipliziert wird. Die Codesymbole ergeben sich aus den Koeffizienten des resultierenden Polynomes $s(x) = p(x)g(x)$. Die Korrektur von Fehlern erfolgt durch Lösen eines Gleichungssystems aus dem Restpolynom, welches sich bei einem fehlerhaft erhaltenen $s'(x)$ per Polynomdivision durch $g(x)$ ergibt.[2]

Der größte Nachteil von Reed Solomon Codes ist die hohe Rechenkomplexität von $O(knx)$ pro Datenblock zum Kodieren und $O(m^3)$ zum Dekodieren, wobei k die Anzahl Eingabesymbole und x die Anzahl Symbole pro Block sind. [AEC04] Mit Erhöhen jedes der Parameter steigt also der Rechenaufwand proportional an. Vor diesem Hintergrund lohnt sich der Einsatz im P2P CDN höchstens, um das erneute Übertragen einzelner fehlerhafter Chunks zu vermeiden, da bei mehreren Gigabytes großen Dateien der Rechenaufwand für das Erzeugen von Redundanz erheblich wäre. Da ein Chunk typischerweise nur wenige hundert Kilobytes groß ist, könnte ein Peer mit korrektem Chunk on-the-fly einige Blöcke mit Redundanz erzeugen und senden, mit denen sich der fehlerhafte Chunk retten lässt.

3.5 Tornado Codes

Tornado Codes gehören zur Klasse der beinahe optimalen Erasure Codes. Sie erlauben die Fehlerkorrektur, wenn mindestens k – im Normalfall aber etwas mehr – Blöcke übertragen wurden. Sie sind damit zwar weniger dateneffizient als Reed Solomon Codes, sind dafür aber 100 bis 10.000 mal effizienter zu berechnen als diese. Der Grund liegt darin, dass Tornado Codes für die Kodierung und Dekodierung ausschließlich effiziente XOR-Operationen ausführen müssen. Die Eingabedaten werden in gleich große Blöcke zerteilt, deren Größe gleichzeitig auch der Größe der Recoveryblöcke entspricht. Um einen Recoveryblock zu erzeugen, werden zwei oder mehr Blöcke per XOR verknüpft. Als Beispiel sei angenommen es gebe vier Eingabeblöcke A bis D. Ein Recoveryblock bestehe aus der XOR-Verknüpfung von je drei zufällig ausgewählten Blöcken, z. B. $R_1 = (A \oplus B \oplus D)$ und $R_2 = (B \oplus C \oplus D)$. Geht bei der Übertragung Block B ver-

[2] `http://en.wikipedia.org/wiki/Reed-Solomon_error_correction` (abgerufen am 05.06.2012)

loren oder ist fehlerhaft, kann er mithilfe des Recovery Blocks R_1 oder R_2 berechnet werden: $B = (A \oplus D \oplus R_1)$. Mit diesen beiden Blöcken können auch zwei fehlerhafte Blöcke wiederhergestellt werden, außer es handelt sich dabei um B und D. In diesem Fall wären mehr Informationen in Form eines weiteren Blocks nötig. Recovery Blöcke können wegen der Kommutativität und Assoziativität von XOR auch kaskadiert erzeugt werden, sprich aus anderen Recovery Blöcken. Es kann also $R_3 = (R_1 \oplus D) = (A \oplus B \oplus D) \oplus (D) = (A \oplus B \oplus D \oplus D) = (A \oplus B)$ berechnet werden. Damit können B und D wiederhergestellt werden, indem nämlich zuerst $B = (A \oplus R_3)$ und dann $D = (A \oplus B \oplus R_1)$ berechnet wird. Der Overhead zusätzlich zu übertragender Blöcke variiert, da nicht jeder Recovery Block Informationen zu jedem Eingabeblock kodiert. Es hängt also von der Reihenfolge der erhaltenen Blöcke ab, wann eine vollständige Rekonstruktion möglich ist. Bei effizienten Implementierungen ist ein durchschnittlicher Mehraufwand von ca. 5,4 Prozent erreichbar. Dabei werden also $1,054 \times k$ Blöcke übertragen. [DFA98]

Tornado Codes sind dann vorteilhaft, wenn nur in eine Richtung kommuniziert werden kann, eine erneute Übertragung schwierig oder unmöglich ist, aber trotzdem eine zuverlässige Übertragung sichergestellt werden soll. Aufgrund des Overheads von mindestens 5 Prozent und der Tatsache, dass in P2P-Netzen bidirektionale Übertragungen kein Problem darstellen, lässt sich der Mehraufwand nur schwer rechtfertigen. Mit blockbasierten Hash-Verfahren ist die Eingrenzung von fehlerhaften Datenblöcken leicht möglich. Die Übertragung der dazu nötigen Hash-Werte unterschreitet den Overhead von 5 Prozent deutlich.

3.6 Fountain Codes

Zu den Erasure Codes gehört auch die Unterklasse der Fountain Codes. Sie haben die Eigenschaft, dass sie aus einer Menge von Eingabesymbolen eine unbegrenzte Anzahl Codesymbole erzeugen können. Im Falle eines optimalen Fountain Codes genügen bei k Eingabesymbolen k Codesymbole, um den Originalzustand der Eingabe wiederherzustellen. Fountain Codes sind dann gut geeignet, wenn es darum geht, große Mengen von Daten effizient zu Kodieren und zu Dekodieren. Auch ohne Rückkanal kann eine Datei zuverlässig an eine Menge von Empfängern verteilt werden, indem für eine längere Zeit stetig neu erzeugte Codesymbole gesendet werden. Andere Erasure Codes erzwingen, aufgrund ihrer Codeeigenschaften oder der Berechnungskomplexität, dass eine Datei in Blöcke eingeteilt und jeder Block unabhängig kodiert wird. Das führt dazu, dass ein Empfänger für jeden einzelnen Block eine ausreichende Menge Codesymbole empfangen haben muss, um die ganze Datei zu rekonstruieren. Bei Fountain Codes ist dies nicht notwendig. [FOU05]

3.7 LT-Codes

LT-Codes sind beinahe optimale Fountain Codes, die sehr einfach zu Kodieren und Dekodieren sind und damit eine hohe Performanz aufweisen. Der Algorithmus basiert im Wesentlichen auf XOR-Operationen. Die zu kodierende Nachricht wird zunächst in n gleich große Blöcke eingeteilt. Für jedes zu sendende Paket passiert folgendes: Der Grad des Pakets d mit $1 \leq d \leq n$ wird zufällig aus einer Wahrscheinlichkeitsverteilung $p(d)$ gewählt. Genau d Blöcke der Originalnachricht werden zufällig ausgewählt und miteinander per XOR verknüpft. Dem generierten Block wird ein Header vorangestellt, der n, d, die Indizes der d genutzten Eingabeblöcke und eine Prüfsumme enthält. Der Empfänger prüft zunächst, ob der erhaltene Codeblock fehlerhaft oder ein Duplikat ist. Wenn $d > 1$ ist, wird versucht, den Grad des Codeblocks zu reduzieren. Dazu werden die bereits erfolgreich wiederhergestellten Originalblöcke, die im Codeblock ebenfalls kodiert sind, mit dem Codeblock geXORt. Gilt immer noch $d > 1$, wird der Codeblock zurückgestellt. Reduziert sich der Grad d eines Codeblocks zu 1, wurde ein Originalblock erfolgreich dekodiert. In diesem Fall werden alle zurückgestellten Codeblöcke, die den neuen Originalblock kodiert enthalten, mit diesem geXORt und somit deren Grad weiter reduziert. Die Wahrscheinlichkeitsverteilung $p(d)$ ist der entscheidende Parameter. Es werden viele Blöcke mit kleinem Grad d benötigt, um das Dekodieren starten zu können und es am Laufen zu halten, sowie einige Blöcke mit hohem Grad, damit ausgeschlossen ist, dass es Eingabeblöcke gibt, die nicht im Code enthalten sind. Es gibt einige Arbeiten, die sich mit dem Finden einer optimalen Verteilung beschäftigen. [LTC02, FOU05]

Ein praktisches Problem für P2P Content Distribution ist, dass regelmäßig frische Codeblöcke generiert werden müssen, die aus einem großen Teil der Originaldaten berechnet werden, nämlich wenn d nahe n liegt. Eine zu verteilende Datei kann jedoch mehrere Gigabytes groß sein. Das Lesen der Eingabeblöcke von der Festplatte zur Generierung eines Codeblocks würde damit Minuten beanspruchen, was offensichtlich inakzeptabel ist. Möglicherweise lässt sich der beschriebene Kodieralgorithmus aber so modifizieren, dass es gar nicht nötig ist, solch komplexe Codeblöcke zu erzeugen. In BitTorrent weiß jeder Peer von allen anderen verbundenen Peers stets, welche Blöcke diese bereits fertig gestellt haben. Mit diesem Wissen können gezielt Codeblöcke erzeugt werden, die für den Empfänger nützlich sind. Dies sind nämlich solche, die sowohl aus ihm bekannten, als auch unbekannten Eingabeblöcken generiert sind. Da die Menge der dem Empfänger unbekannten Eingabeblöcke während des Downloads stetig schrumpft, ist es ausgeschlossen, dass es einen Eingabeblock gibt, der nie in einen Codeblock einfließt.

3.8 Raptor Codes

Raptor Codes stellen eine praktische Verbesserung gegenüber LT Codes dar. Symbole können eine beliebige Länge haben. Die Kosten zum Kodieren und Dekodieren sind konstant. Bei der aktuellsten Variante von Raptor Codes (RaptorQ) liegt die Chance, die Originalnachricht erfolgeich zu dekodieren, bei 99%, wenn k Symbole empfangen wurden. Bei $k+2$ Symbolen liegt sie bei mindestens 99,9999%.[3] Raptor Codes verweben zwei verschiedene Codes miteinander. Es gibt einen äußeren und einen inneren Code. Der äußere Code ist ein beliebiger Erasure Code, der die Originaldaten als Eingabe erhält und eine fixe Menge an Redundanz erzeugt. Der innere Code ist ein LT-Code, der den äußeren Code als Eingabe erhält. Der Vorteil gegenüber reinen LT-Codes ist, dass dessen Eingabe (der äußere Code) nicht vollständig wiederherstellbar sein muss. Es muss also insbesondere nicht sichergestellt sein, dass jedes Eingabesymbol des LT-Codes in mindestens einem Codesymbol kodiert ist. Daher können alle Codesymbole des LT-Codes eine kleine Dimension d besitzen. Der äußere Code kann, abhängig vom gewähltem Code, entweder vorberechnet werden, was entsprechend zusätzlichen Speicherplatz benötigt, oder on-the-fly berechnet werden, was ggf. mehr Rechenzeit beansprucht. [RAP06]

Raptor Codes nutzen die starken Seiten von Erasure Codes mit festgelegter Redundanz und LT-Codes aus. Erstere lassen sich meist effizient berechnen, wenn die Menge zu erzeugender Redundanz klein ist. Letztere lassen sich sehr effizient berechnen, wenn ein Codeblock aus einer realistisch geringen Zahl Eingabeblöcken berechnet wird, sodass die Performanz des Speichermediums nicht zum Flaschenhals wird. Wenn unzuverlässige Transportprotokolle zum Übertragen von Content zum Einsatz kommen, sind Raptor Codes somit eine in Betracht zu ziehende Übertragungskodierung.

3.9 Network Coding

Network Coding hat das Ziel, den maximal möglichen Informationsfluss in einem Netzwerk zu erreichen. Es gibt einen Sender und mehrere Zwischenknoten, die gleichzeitig Empfänger sein können – eine typische Multicast-Situation. In Multicast-Netzwerken ist die Topologie meist bekannt. In einem dezentralen CDN hat jedoch jeder Knoten in der Regel nur lokale Informationen über die Netzwerktopologie in der unmittelbaren Nachbarschaft. Knoten wissen von ihren Nachbarknoten insbesondere nicht, welche Informationen diese bereits von anderen Nachbarn erhalten haben. Ohne Kodierung und Wissen über das gesamte Netzwerk ist die obere Schranke für den Informationsfluss gemäß des Max-Flow-Min-Cut-Theorems praktisch kaum erreichbar. Network Coding kann den Informationsfluss maximieren, indem jeder Zwischenknoten aus seinen bisher erhaltenen Paketen durch Linearkombinationen über einem endlichen

[3] http://www.ietf.org/proceedings/77/slides/rmt-1.pdf

Körper $GF(2^x)$ neue Pakete erzeugt, die er an seine Nachbarn sendet. Nach dem Erhalt ausreichend vieler linear unabhängiger Pakete kann der Empfänger die Originalpakete mittels des gaußschen Eliminationsverfahrens wiederherstellen. Als Beispiel für den Nutzen für den Informationsfluss sei folgendes Beispiel skizziert: Es gibt einen Sender S und drei Knoten A, B, C, die dieselbe Datei herunterladen wollen. Knoten A hat Paket 1 und 2 von S geladen. Knoten B ist mit A verbunden und kann mit gleicher Wahrscheinlich entweder Paket 1 oder 2 herunterladen. Knoten C entscheidet sich unabhängig, Paket 1 herunterzuladen. Ohne Network Coding kann sich Knoten B ebenfalls für das Herunterladen von Paket 1 entscheiden. Dann haben B und C aber nichts, das sie tauschen könnten. Lädt B aber eine Linearkombination der Pakete 1 und 2 von A herunter, haben B und C für den jeweils anderen Knoten relevante Informationen. Es ist jedoch etwas problematisch, nur aufgrund lokaler Informationen die richtigen Pakete zu erzeugen, also solche, die für den Empfänger nützlich sind.

In [NWC05] wird eine Möglichkeit vorgestellt, Content mittels Network Coding nur mit lokalen Informationen zu propagieren. In heterogenen Netzwerken haben die Autoren per Simulationen eine Verbesserung der Downloadrate um 30% gemessen und festgestellt, dass Knoten selbst dann ihren Download fertigstellen können, wenn der originale Seeder nach dem Hochladen genau einer Kopie das System verlässt und jeder Knoten nach der Fertigstellung das System ebenfalls verlässt. Um ein Paket zu erzeugen wird für jeden bereits bekannten Block ein zufälliger Koeffizient generiert, jeder Block mit seinem jeweiligen Koeffizienten multipliziert und die Produkte aufsummiert. Das Ergebnis dieser Rechnung und der Koeffizientenvektor werden als Paket gesendet. Für Dateien mit einer Größe, die sich nicht im Hauptspeicher ablegen lassen, ist der vorgeschlagene Ansatz extrem ineffizient, da für die Erzeugung jedes Blocks alle k Blöcke der kompletten Datei in die Linearkombination einfließen. Folglich müsste für jeden erzeugten Block die komplette Datei vom Datenträger erneut eingelesen werden. Bei sehr großen Dateien fällt zudem die Übertragung der Koeffizienten zusätzlich ins Gewicht, da für jeden Dateiblock ein Koeffizient benötigt wird. Bei zehntausenden Blöcken müsste also pro übertragenem Block zusätzlich ein Vektor im zwei- bis dreistelligen Kilobyte-Bereich übertragen werden.

In [CVL06] werden diese Probleme angegangen, indem Linearkombinationen nicht aus der kompletten Datei, sondern aus Segmenten aus 50 bis 100 Blöcken berechnet werden, wobei eine 4 GiB Datei in 1000 bis 2000 Blöcke zerlegt wird. Der Overhead für die Übertragung der Koeffizienten wird jedoch nicht thematisiert, der bei 2000 Blöcken und dem verwendeten $GF(2^{16})$ ca. 5 KiB pro übertragenem Block beträgt. Ein Block ist somit ca. 2-4 MiB groß. Dieser muss vollständig übertragen werden, um von Nutzen zu sein, im Gegensatz zur Übertragung in BitTorrent, wo eine Übertragungseinheit nur 16KiB groß ist. Zudem könnte ein Angreifer gefälschte Blöcke senden und damit das komplette Segment für den Empfänger unbrauchbar machen. Dagegen wird ein Checksummen-Algorithmus angewendet, bei dem ein Seeder für

jeden Client einen eigenen Satz Secure Random Checksums (SRCs) genannter Prüfsummen berechnet und über einen sicheren Kanal übermittelt, da diese geheim bleiben müssen, weil andernfalls ein Angreifer leicht gefälschte Blöcke berechnen kann, die nicht erkannt werden. Was in dem Fall geschieht, wenn es keine Seeder gibt, wird von den Autoren nicht behandelt. Ein anderes Verfahren zur Integritätsprüfung bei anonymem P2P File Sharing und Einsatz von Network Coding wird in [INC10] vorgestellt. Es basiert aus homomorphen Hashes, wobei mehrere empfangene Blöcke auf einmal verifiziert werden. Der Algorithmus ist jedoch sehr langsam und daher für eine on-the-fly Nutzung uninteressant.

Swifter [SWI08] und iSwifter [ISW12] kombinieren segmentbasiertes Network Coding, auch Chunked Network Coding genannt, mit dem aus BitTorrent bekannten Local-Rarest-First Scheduling-Algorithmus. Per Pull-Prinzip wird ein anderer Knoten gebeten, einen per Network Coding erzeugten Block aus dem gewählten Segment zu senden. Ein Block ist 256KB groß, wobei jedes Segment aus 16-64 Blöcken besteht. Sind alle Blöcke eines Segments empfangen, kann letzteres dekodiert werden. Peers tauschen in regelmäßigem Abstand eine Liste mit der Anzahl der für jedes Segment gepufferten Blöcke aus. iSwifter reduziert im Vergleich zu Swifter lediglich den Overhead von Block-Requests und erübrigt das Übertragen der Koeffizientenvektoren, indem stattdessen nur der Seed-Wert des Pseudozufallszahlengenerators übermittelt wird, der die Koeffizienten erzeugt. Die vorgenommenen Performance-Messungen sind jedoch nur wenig aussagekräftig, da nur drei Varianten einer eigenen Implementierung von Swifter getestet wurden, wobei jeder Peer auf unrealistische 4 gleichzeitige Uploads und Downloads beschränkt ist und jeder Peer regelmäßig den Schwarm verlässt, um Churn zu simulieren. Die Autoren ziehen aus ihren Messungen den Schluss, dass größere Segmentgrößen die Downloadzeit verkürzen, weil es für den einzelnen Peer leichter ist, nützliche Blöcke zu finden. Dies ist jedoch nur unter der Annahme haltbar, dass kein Peer falsche Blöcke sendet, da deren Integrität nicht vor dem Weiterleiten überprüft wurde. Es werden also auch Blöcke aus Segmenten weitergeleitet, die noch nicht vollständig heruntergeladen und einem Integritätstest unterzogen wurden. Knoten könnten also durch Zuspielen gefälschter Blöcke so manipuliert werden, dass sie die korrupten Daten an ihre Nachbarn weiterleiten, womit im Netzwerk erheblicher zusätzlicher Traffic entsteht, wenn viele Knoten ganze Segmente neu herunterladen müssen.

3.10 Zusammenfassung

Die meisten der untersuchten Kodierverfahren benötigen, auf große Datenmengen angewendet, sehr viel Rechenzeit zum Erzeugen von Redundanzinformationen und/oder Nutzung dieser zur Wiederherstellung von fehlenden Daten. Zudem wird zusätzlicher Speicherplatz benötigt, um die Redundanzinformationen vorzuhalten. Diese Kodierverfahren sind also eher zur Korrektur von Übertragungsfehlern oder Fehler auf Speichermedien geeignet als zur Rekonstruktion

in großen Datenmengen. Das einzige vielversprechende unter den untersuchten Verfahren ist Network Coding, da es sich effizient berechnen lässt. Es erfordert jedoch, dass die kompletten zu kodierenden Daten im Speicher gehalten werden müssen, da jeder gesendete Block per Linearkombination der kompletten Datenmenge gebildet wird. Daher eignet es sich in der Praxis nur für Datenmengen bis einige hundert Megabyte. Dieses Problem lässt sich durch Segmentierung der zu übertragenden Daten teilweise lösen, wie das Projekt iSwifter beweist. Dadurch müssen die aktuell benötigten Segmente im Speicher vorgehalten werden. Dies erkauft man sich aber wiederum dadurch, dass nur Codeblöcke zur Wiederherstellung eines Segments geeignet sind, die aus demselben Segment generiert wurden. Kann man nicht für jedes Segment genügend viele Codeblöcke herunterladen, kann nicht jedes Segment wiederhergestellt und der Download damit nicht abgeschlossen werden. Folglich stellt sich wieder dasselbe Problem ein, für das wir eigentlich eine Lösung suchten. Da sich keine Lösung für das eingangs beschriebene Verfügbarkeitsproblem abzeichnet, die nicht mit erheblichen Nachteilen verbunden ist, verbleibt also nur der Verzicht auf eine spezielle Übertragungskodierung in ByteStorm.

4 Belohnungssysteme

P2P-Systeme leben von den Beiträgen ihrer Benutzer. Ein Schlüsselelement zum Erfolg oder Misserfolg eines P2P-Systems für Content Distribution ist daher, möglichst viele (am besten alle) Benutzer dazu zu bewegen, Ressourcen – in Form von Speicherplatz und Upload-Bandbreite – bereitzustellen. Viele Benutzer möchten vom System in maximaler Qualität profitieren, aber gleichzeitig den eigenen Beitrag möglichst gering halten. Die Extremform dieses Phänomens, bei der Benutzer das System nutzen, ganz ohne jeglichen Beitrag zu leisten, wird als Freeriding bezeichnet. Dieses Verhalten ist jedoch ein Widerspruch, da die Summe der genutzten Ressourcen die Summe der von anderen Benutzern bereitgestellten Ressourcen nicht übersteigen kann. Um dieses Dilemma zu lösen, gibt es zahlreiche Ansätze, den Benutzern Anreize zu geben, Ressourcen bereitzustellen, um die eigene Performanz zu steigern und Freeriding unattraktiv zu machen. Diese Ansätze belohnen entweder Benutzer, die Ressourcen freigeben und sich fair verhalten, sanktionieren solche, die dies nicht tun, oder wenden eine Kombination dieser Maßnahmen an.

Dieses Kapitel betrachtet existierende Belohnungssysteme, die für den Einsatz in Content Distribution Systemen geeignet sind. Ziel ist die Ideenfindung für ein einfaches, betrugsresistentes, aber effektives System für einen Anreiz zur Bereitstellung von Upload-Bandbreite.

4.1 Eine Klassifizierung

Belohnungssysteme für P2P lassen sich nach [PAP11] in die folgenden drei Klassen einteilen:

- Direkte Erwiderung: Jeder Knoten besitzt nur Informationen, die er aus der Interaktion mit anderen Knoten heraus lokal beobachten kann. Er entscheidet anhand dieser direkten Beobachtungen, mit wem er wann Daten austauscht. Für nützliche Daten, die ein Knoten von direkt verbundenen Nachbarknoten erhält, revanchiert er sich, indem er andere nützliche Daten zurück sendet und/oder dessen Anfragen bevorzugt behandelt (z. B. Positionen in einer Warteschlange hinaufrückt). Es findet also ein direkter, bilateraler Austausch statt. Da nur lokal verfügbare Informationen genutzt werden, sind Verfahren mit direkter Erwiderung relativ robust gegen Angriffe wie Sybil-Attacken und zusammenarbeitende Knoten, die versuchen ihre Reputation zu manipulieren.

- Indirekte Erwiderung: Leistungen können auch aufgrund Erfahrungen anderer Teilnehmer einem Knoten gewährt werden. Anfragen des Knotens werden bedient, weil er bereits anderen Knoten seine Ressourcen zur Verfügung gestellt hat. Ein Knoten baut also durch den Austausch mit einem Knoten eine Reputation nicht nur mit diesem, sondern indirekt auch mit dritten Knoten auf. Lokale Beobachtungen sind nicht ausreichend, um indirekte Erwiderungen zu leisten. Es müssen zusätzliche Informationen über die Tauschhistorie mit Nachbarn ausgetauscht werden, um das eigene Tauschverhalten steuern zu können. Der Schutz vor Angriffen ist bei solchen Systemen erheblich aufwendiger, da Knoten gezielt falsche Angaben über andere Knoten machen können, um deren Reputation zu steigern oder zu senken.
- Virtuelle Währung-basiert: Vom Handel mit herkömmlichen Gütern gegen Geld inspiriert, gibt es auch eine Klasse von Ansätzen, die die Leistungen mit einer virtuellen Währung vergüten. Der Sender einer Datei oder eines Datenblocks wird vom Empfänger mit einer vorbestimmten oder ausgehandelten Menge der virtuellen Währung bezahlt. Ersterer kann sie verwenden, um selbst Dateien herunterzuladen. Um Betrug zu erkennen ist eine Kontrollinstanz in Form einer Bank nötig. Diese kann zentral verwaltet oder dezentral auf Knoten verteilt sein.

Der vielleicht wichtigste Aspekt ist die Resistenz von Belohnungssystemen gegen das Ausnutzen dieser zum eigenen Vorteil einzelner Benutzer. Je komplexer ein Belohnungssystem ist, desto schwieriger ist es, mögliche Angriffe auszuschließen. Einige Angriffsszenarien sind dem jeweiligen Typ des Belohnungsmechanismus immanent und somit bei der Planung jedes Belohnungssystems zu berücksichtigen. Eine Reihe gängiger Angriffe wird im Folgenden beschrieben:

- Whitewashing: Durch die Tatsache, dass Identitäten in P2P-Systemen leicht zu erzeugen sind, ist es ebenso leicht, die eigene Identität zu jedem beliebigen Zeitpunkt zu wechseln. Im Hinblick auf Belohnungssysteme kann eine Motivation hierfür sein, die eigene negative Reputation zu verlieren, die sich aufgrund negativen Tauschverhaltens ergeben hat. Durch einen Wechsel der Identität kann sich der Benutzer die eigene Weste rein waschen und sich wieder eine neutrale Reputation verschaffen. Dies wird als Whitewashing bezeichnet. Whitewashing ist immer dann möglich, wenn neue Identitäten beim Betreten des Systems eine bessere Reputation haben, als die schlechtest mögliche.
- Strategische Clients: Rationale Benutzer versuchen immer ihren eigenen Profit vom System zu maximieren und gleichzeitig so wenig wie möglich zum System beizutragen, um die Profitmaximierung zu erreichen. Dazu kann eine strategische Implementierung des P2P-Clients verwendet werden, die anderen Peers gegenüber gerade nur soviel leistet, dass die gewünschte Service-Qualität erreicht wird, also z. B. Peers nicht aufhören, Daten an den eigenen Client zu senden. Ein robustes System macht strategisches Handeln entweder un-

möglich oder zum Teil des Systems, sodass alle Clients strategisch (d.h. profitmaximierend) arbeiten und sich deshalb unkooperatives Verhalten nicht lohnt.

- Sybil-Attacken: Die Einfachheit, Identitäten zu erzeugen, erlaubt es einem Benutzer auch, sich selbst eine beliebig große Anzahl von Identitäten zuzulegen. Diese können gezielt dafür verwendet werden, das Belohnungssystem zu unterlaufen. So könnte ein Benutzer eine zweite Identität dafür benutzen, um seiner Hauptidentität eine gute Reputation bei anderen Netzwerkknoten zu verschaffen ohne tatsächlich etwas dafür zu leisten und daraus Vorteile ziehen, z. B. schnellere Downloads. In einem resistenten System zieht ein Angreifer daraus jedoch keine Vorteile, sondern erreicht maximal die gleiche Performanz, wie ein einzelner Knoten.
- Konspiration: Unterschiedliche Knoten könnten auch zusammenarbeiten, um sich gegenseitig oder einem bestimmten anderen Knoten zu einer guten oder schlechten Reputation zu verhelfen. Auch könnten sie für die Netzwerkstruktur wichtige Knoten – bei auf virtueller Währung basierenden Systemen z. B. die zentrale Bank – mit Anfragen überlasten und einen Denial of Service auslösen.

4.2 Tit-for-tat in BitTorrent

Tit-for-tat (sinngemäß übersetzt: wie du mir, so ich dir) ist die Strategie in BitTorrent, um die eigene Download-Geschwindigkeit zu optimieren. Dabei lädt jeder Peer immer zu einer begrenzten Anzahl derjeniger Nachbarn Daten hoch, von denen er im Moment die meisten Daten erhält. Dies tut er mit sog. Chokes und Unchokes, die Nachrichten sind, welche den Nachbarn mitteilen, dass sie nun Daten erhalten bzw. keine weiteren Daten erhalten können.

Zusätzlich gewährt jeder Peer in regelmäßigen Zeitabständen einem Nachbarn Upload-Bandbreite, der gerade keine Daten zurücksendet. Dies wird als Optimistic Unchoke bezeichnet. Dadurch sollen potenziell geeignetere Tauschpartner gefunden werden und es soll neuen Peers ermöglicht werden, zu einem existierenden Schwarm hinzuzustoßen und erste Blöcke zu erhalten, die er dann weiterverteilen kann. Ohne Optimistic Unchokes wäre ein Einstieg nur äußerst schwer möglich. Diese Funktion wird jedoch von strategischen BitTorrent-Implementierungen ausgenutzt, um Freeriding zu betreiben, indem der Client deutlich mehr Verbindungen aufbaut als üblich, um möglichst viele Optimistic Unchokes zu erhalten. Aber auch der reguläre Choking-Mechanismus lässt sich ausnutzen wie BitTyrant [IBR07] beweist. Dieser Client minimiert die hochgeladene Datenmenge gerade so weit, dass er von anderen Peers keinen Choke erhält und somit das Erhalten von Daten fortgesetzt wird.

Tit-for-tat wirkt folglich nur solange einem unfairen Verhalten entgegen, solange sich alle Peer an die spezifizierte Upload-/Downloadstrategie halten. Peers die dies nicht tun, können auf Kos-

ten anderer mit geringem bis mäßigem Aufwand einen Torrent deutlich schneller fertigstellen und aus dem Schwarm verschwinden ohne eine ausreichende Gegenleistung erbracht zu haben.

4.3 eMule Belohnungssystem

Jeder Knoten speichert für jeden seiner Peers die zu ihm herauf- und von ihm heruntergeladene Datenmenge in der Vergangenheit. Beide zusammen werden als Credits bezeichnet. eMule entscheidet anhand einer Prioritätsfunktion, welche Requests wann bedient werden. Die Priorität hängt von den Credits, einiger Konstanten und der Ankunftszeit des Requests ab. Je länger ein Request also bereits in der Warteschlange wartet und je besser der Creditstand, desto eher wird der Request bedient. In [IIE12] wird anhand eines Modells und anhand von Messungen gezeigt, dass eMule das Problem des Verhungerns aufweist. Peers mit niedriger Bandbreite müssen sehr lange warten, bis sie von anderen Peers Daten erhalten. Der Einfluss der Up/Downloadhistorie in Form von Credits ist zwar nach oben und unten hin beschränkt, allerdings sind die Länge der Wartezeit des Requests und die Credits Koeffizienten einer Multiplikation. Die Priorität eines Peers mit mittlerer Bandbreite wächst somit erheblich schneller mit der Länge der Wartezeit, als die eines langsamen Peers. Ein mittelschneller Peer, dessen Request erst deutlich später in die Warteschlange aufgenommen wird, kann einen lange wartenden langsamen Peer in der Prioritätsbewertung in kurzer Zeit überholen. Die Autoren schlagen einen verbesserten Ansatz für die Prioritätenberechnung vor, der im Wesentlichen darauf basiert, dass die Wartezeit und Credits addiert, anstatt multipliziert werden. Dadurch erhöht sich die Priorität mit zunehmender Wartezeit für alle Peers gleichermaßen, unabhängig von ihren Credits. Letztere haben somit nur einen Einfluss auf die initiale Prioritätsbewertung, mit der ein Request bei seiner Ankunft einsortiert wird.

eMules Belohnungssystem, das auf Credits basiert, die auch über einen längeren Zeitraum und über unterschiedliche Dateien hinweg bestand haben, stellen im Zeitalter kostengünstig mietbarer externer Bandbreite ein Problem dar. Auf BitTorrent-Trackern mit geschlossenen Benutzergruppen müssen die Benutzer meist dafür sorgen, dass ihr heruntergeladenes Datenvolumen das hochgeladene Datenvolumen nicht signifikant übersteigt. Je mehr das Uploadguthaben das Downloadvolumen übersteigt, desto weniger muss der Benutzer fürchten, vom Tracker-Betreiber sanktioniert zu werden. Einige Benutzer mieten sich zeitweise günstig Bandbreite bei Hosting-Providern, wo sie innerhalb kurzer Zeit große Datenmengen an die anderen Benutzer des Trackers hochladen und somit ihr Uploadguthaben erheblich steigern (oft im Terabyte-Bereich). Danach stellen diese Benutzer kaum noch oder gar keine Bandbreite mehr zur Verfügung, sondern nutzen ihr aufgebautes Kontingent. Dasselbe ist bei eMules Belohnungssystem möglich. Innerhalb kurzer Zeit kann sich ein Benutzer mit gemieteter Bandbreite eine hohe Creditbewertung bei vielen Peers verschaffen, welche er in der Zukunft verbrauchen kann,

ohne noch etwas beitragen zu müssen, auf Kosten der Performanz bei der Verteilung der zukünftigen Dateien. Daher ist es zweifelhaft, ob ein Belohnungssystem sinnvoll ist, bei dem das eigene Verhalten zu Auswirkungen über einen langen Zeitraum (Wochen, Monate) führt.

4.4 BAR Gossip

BAR ist ein Modell, das Knoten in einem Peer-to-Peer Netzwerk in 3 Klassen unterteilt:

- Byzantinisch: Knoten mit beliebigen Fehlfunktionen oder manipulierte Knoten
- Altruistisch: Sich unter allen Umständen an die Protokoll-Vorgaben haltende Knoten
- Rational: Knoten, der danach strebt, seinen eigenen Profit zu maximieren

Die Autoren von [BAR06] beschreiben ein Gossip Protokoll, das BAR-tolerant ist, also mit fehlerhaften und selbstsüchtigen Knoten umgehen kann. Es wurde entworfen, um ein robustes Live Streaming Protokoll darauf aufzubauen. BAR Gossip ist darauf ausgelegt, den Datendurchsatz kurzfristig zu optimieren, Anreize innerhalb kurzer Intervalle zu schaffen und die rechtzeitige Übermittlung von Daten zu sichern. Resistenz gegen byzantinische Knoten ist durch die Verwendung von Gossip gegeben, da die Peers, mit denen kommuniziert wird, per Zufall gewählt werden. BAR Gossip verwendet jedoch verifizierbaren Pseudozufall für die Peer-Auswahl, um rationale Knoten daran zu hindern, ihre egoistische Tätigkeit in legitimem Verhalten zu verstecken. Clients generieren ein Public-Private-Schlüsselpaar und meldet sich mit dem Public Key beim Broadcaster an. Wenn alle Clients angemeldet sind, veröffentlicht der Broadcaster eine Liste mit den Public Keys und IP-Adressen der Teilnehmer. Es wird davon ausgegangen, dass nicht-byzantinische Knoten bis zum Ende des Live Streams im System bleiben und keine neuen Knoten hinzukommen. Die gesendeten Nachrichten werden von jedem Client mit seinem privaten Schlüssel signiert. Wenn genügend signierte Nachrichten von einem Client auftauchen, die das Protokoll verletzen, ist dies ein „Proof of Misbehavior“ (POM), welches dazu führt, dass der Knoten ausgeschlossen wird. Das Protokoll ist rundenbasiert. In jeder Runde werden Updates, also Stücke des Streams, übertragen, die bis zu einer Deadline bei allen Empfängern ankommen sollten. Updates werden vom Broadcaster an eine zufällige Auswahl von Knoten geschickt, von denen mindestens einer nicht-byzantinisch ist. Updates zwischen Knoten werden per „Balanced Exchange“ ausgetauscht. Wenn Knoten A Knoten B beispielsweise 5 Updates anbieten kann, Knoten B hat aber nur 3 für A interessante Updates, tauschen die beiden 3 Updates aus. Für zurückgefallene Clients gibt es zudem das Prinzip des „Optimistic Push“. Knoten A sendet freiwillig nützliche Updates an B, in der Hoffnung, dass B sich später dafür revanchiert.

Der Ansatz von BAR Gossip ist für den Einsatz in P2P CDNs nicht geeignet, da eine zentrale Stelle die Schlüssel verwaltet, kein Betreten oder Verlassen des Systems während eines laufenden Streams vorgesehen ist und weil das Protokoll auf zeitnahen Datenaustausch optimiert ist.

Die einzig interessanten Funktionen sind Balanced Exchange und Optimistic Push, die zusammen eine ähnliche Funktion wie die Auswahl der zu senden Chunks in FairTorrent haben (siehe 2.4).

4.5 Dandelion

Dandelion ([DAN07]) ist ein auf kommerziellen Einsatz ausgerichtetes Content Distribution System. Es soll Zugriffe bei Lastspitzen von einem zentralen Server an Clients delegieren, die den Content bereits heruntergeladen haben. Es kommt ein Belohnungssystem zum Einsatz, das Clients zum Upload motivieren soll, aber gleichzeitig keinen Anreiz zum unauthorisierten Verteilen des Contents gibt. Honorierung von Upload erfolgt nur für das Hochladen zu authorisierten Clients, z.B. von Benutzern, die für den Content bezahlt haben. Ein Dandelion Server unter normaler Auslastung liefert die Daten auf Anfrage direkt aus. Erst bei Überlastung wird der Datentransfer an Clients deligiert. Clients erhalten für die Bereitstellung von Bandbreite virtuelles Guthaben, das für die Zahlung späterer Downloads verwendet werden kann. Der Content wird, wie bei BitTorrent, in Chunks aufgeteilt, sodass während des Downloads bereits wieder Teile der Datei angeboten werden können. Der Dandelion Server ist zentrale Instanz für die Steuerung der dezentralen Datenübertragungen.

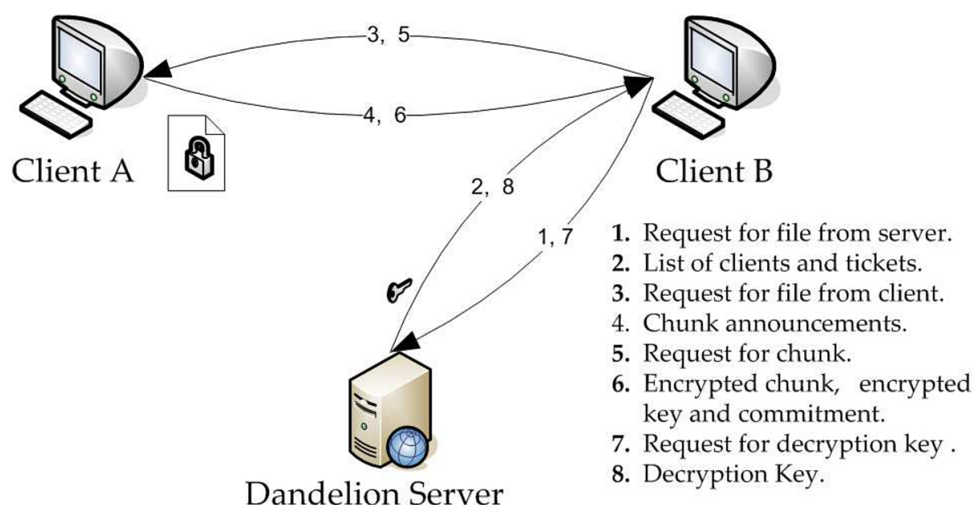

Abbildung 4.1: Dandelion Nachrichtenaustausch beim Download zwischen Clients [DAN07, Seite 2]

Abbildung 4.1 zeigt, welche Nachrichten ausgetauscht werden, damit ein Client von einem anderen Client ein Chunk an Daten erhält und der Datenaustausch vergütet wird. Client B möchte eine Datei erhalten und fragt diese beim Server an. Er erhält eine Liste mit Hashes der Chunks zum Überprüfen der Dateiintegrität, eine Liste von Adressen anderer Clients und Tickets, die Client B zum Download berechtigen. B fragt A nach der Datei, A prüft, dass Bs Ticket vom Server stammt und informiert B über die verfügbaren Chunks. B sendet ein Request, um einen Chunk

herunterzuladen. A verschlüsselt den Chunk mit einem zufälligen Schlüssel k, welcher mit dem Session Key K_{SA} verschlüsselt wird, den A zuvor mit dem Server ausgehandelt hat. A erstellt eine Signatur durch Berechnung eines MAC über dem Hash der verschlüsselten Daten und dem verschlüsselten Schlüssel k unter Verwendung von K_{SA}. A sendet die verschlüsselten Daten, den verschlüsselten Schlüssel k und den MAC an B. Um die Daten zu entschlüsseln benötigt B den Schlüssel k. B sendet dafür eine Schlüsselanfrage an den Server, welche den verschlüsselten Schlüssel k, den von B berechneten Hash über Daten und k, sowie den MAC von A. Der Server berechnet aus Bs Hash und dem ihm bekannten Schlüssel K_{SA} den MAC und vergleicht ihn mit dem von A gesendeten. Wenn der MAC korrekt ist und B genügend Guthaben besitzt, reduziert er das Guthaben von B, schreibt A Guthaben gut und sendet B den entschlüsselten Schlüssel k. B entschlüsselt die Daten und prüft die Integrität mittels der Chunk Hashes, die er eingangs vom Server erhalten hat. Schlägt die Prüfung fehl, sendet B eine Beschwerde an den Server. Der Server prüft, ob A vorsätzlich falsche Daten gesendet hat, indem er den Chunk selbst mit Schlüssel k verschlüsselt und den MAC mittels K_{SA} berechnet. Stimmt der MAC mit dem von A gesendeten MAC überein, versucht B zu betrügen, andernfalls hat A vorsätzlich falsche Daten gesendet, denn nur A und der Server kennen K_{SA} und bei einem Übertragungsfehler zwischen A und B hätte B den Schlüssel k gar nicht erst erhalten.

Der zentralistische Ansatz von Dandelion macht es für vollständig verteilte P2P Systeme unattraktiv. Zudem besitzt Dandelion mindestens zwei Schwachstellen, die sich vergleichsweise leicht ausnutzen lassen. Die erste liegt darin, dass der Server dem Konto des Senders A eines Chunks sein Belohnungsguthaben gutschreibt, bevor der Empfänger B den Entschlüsselungsschlüssel hat und den Chunk verifizieren kann. Dies könnte A ausnutzen, indem er falsche Daten an B sendet, daraufhin Guthaben vom Server erhält und dieses Guthaben verbraucht, bevor B die Chance hatte eine Beschwerde an den Server zu senden. Das zweite Problem ist die Beschwerdeprüfung auf der Serverseite. Da der Server für jede Beschwerde den beanstandeten Chunk verschlüsseln und einen HMAC berechnen muss, ist dies eine rechenintensive Operation. Mehrere zusammenarbeitende Clients können eine verteilte Denial-of-Service Attacke (DDoS) gegen den Server durchführen, indem sie dem Server fingierte Übertragungen in schneller Folge vorspielen und der vermeintliche Empfänger daraufhin jeden Chunk beanstandet. Der Server wird durch teure Krypto-Operationen überlastet und kann legitime Anfragen nicht mehr rechtzeitig beantworten.

4.6 Karma

Karma ([KAR03], [KAR05]) ist eine digitale Währung, die voll dezentral verwaltet wird, offline verwendbar ist und für den Einsatz in P2P und Grid Systemen bestimmt ist. Die normalerweise zentrale Rolle der Bank wird von einer zufälligen, deterministisch bestimmten Menge

von Knoten übernommen. Zahlungen finden für die Nutzung von Ressourcen statt, z.B. Dateien, Bandbreite, CPU-Zeit. Beim Betreten des Netzwerks erhält jeder Benutzer eine kleine Menge Karma für den Einstieg ins System. Für das Beitragen von Ressourcen erhält er Karma, für das Nutzen von Ressourcen wird Karma vom Konto abgezogen. Wenn nicht genug Karma für eine Transaktion vorhanden ist, findet sie nicht statt. Folglich sind Teilnehmer gezwungen, einen Beitrag zu leisten. Karma setzt auf einem bestehenden DHT-Overlay mit Routing und zuverlässiger Bestimmung von Nachbarn auf. Jeder Benutzer benötigt einen öffentlichen, einen privaten Schlüssel und ein Zertifikat zum öffentlichen Schlüssel. Die verteilte Bank $Bank_A$ für einen Benutzer A besteht aus den k nächsten Knoten zu $HASH(nodeId(A))$, wodurch jeder Knoten die Menge der Bankknoten für jeden Knoten im Netzwerk effizient bestimmen kann. Jeder Knoten in der Menge $Bank_A$ speichert die Menge an Karma, die durch den privaten Schlüssel von A signiert ist, und ein Transaktionsprotokoll. Zudem speichert jeder Bankknoten die Nummer der aktuellen Epoche für De-/Inflationsausgleiche und die letzte verwendete Sequenznummer, die bei jedem Transfer vom eigenen auf ein anderes Konto erhöht wird, um Replay-Attacken zu verhindern. Betritt ein Knoten das Netzwerk, tauschen sich die Banknoten in $Bank_A$ untereinander über den Kontostand des Knotens aus, wobei jede Nachricht signiert ist. Wenn mehr als die Hälfte der Bankknoten identische Kontostände und aktuelle Sequenznummer senden, werden diese als aktueller Stand übernommen. Wenn ein neuer Bankknoten ins Netzwerk eintritt oder ein bestehender Knoten einen verlassenden Bankknoten ersetzt, erfährt der neue Bankknoten die Kontostände auf die gleiche Weise. Transaktionen sind atomar, was bedeutet, dass der Kosument einer Ressource den Schlüssel zum Entschlüsseln der Ressource erhält, während dem Bereitsteller der Ressource gleichzeitig ein Zertifikat über den Erhalt auf der Empfängerseite zugestellt wird. Will A eine Datei von B herunterladen, sendet A den eigenen Kontostand nach der Transaktion und eine Sequenznummer an B, der die Anfrage zusammen mit seinem eigenen signierten Kontostand an $Bank_B$ sendet. Jeder $Bank_B$ Knoten kontaktiert jeden $Bank_A$ Knoten und übermittelt die Anfrage von A. Wenn die Mehrheit aus $Bank_A$ den Kontostand verifiziert hat, wird das Konto von A mit dem entsprechenden Betrag belastet, A wird informiert. Alle Knoten aus $Bank_B$ schreiben dem Konto von B den Betrag gut und informieren B, der nun mit dem Dateitransfer beginnt. Am Ende einer Epoche, die mehere Monate lang ist, findet ein De- bzw. Inflationsausgleich statt, damit der Wert des im Umlauf befindlichen Karmas konstant bleibt. Jeder Bankknoten sendet am Epochenende Informationen über die Anzahl aktiver Konten und die Summe deren Karmas an alle Knoten im Netzwerk. Jeder Knoten lernt so die Anzahl aktiver Konten und die Karmamenge im gesamten Netzwerk. Daraus wird ein Korrekturfaktor berechnet, der auf die vom Knoten verwalteten Kontostände angewendet wird.

Dieser Ansatz hat einige Schwachpunkte, die negative Auswirkung auf Performanz und/oder Zuverlässigkeit haben können:

- Jede Transaktion hat eine Nachrichtenkomplexität von $O(k^2)$, wobei k die Anzahl Bankknoten eines Peers ist, da alle Bankknoten eines Transaktionspartners mit allen Bankknoten des anderen kommunizieren müssen.
- Es ist nicht spezifiziert, was geschieht, wenn die Kontostände zwischen Bankknoten inkonsistent sind und es keine Mehrheit von Bankknoten mit einem aktuellen Stand gibt.
- Es wird eine Möglichkeit beschrieben, die mittels Datenverschlüsselung und Empfängersignatur das Stattfinden einer Übertragung beweisbar macht und damit Rückbuchungen bei nicht erbrachten Leistungen ermöglicht, jedoch wird die Korrektheit des Inhalts nicht geprüft. Daher kann ein Knoten auch nutzlose Daten senden und dafür Karma erhalten.
- Ohne Karma erhält ein Knoten keinerlei Dienste, auch nicht wenn im Netzwerk genügend brach liegende Ressourcen vorhanden sind.
- Jeder Knoten erhält beim erstmaligen Beitritt ins Netzwerk eine kleine Menge Karma. Eine Gegenmaßnahme, die verhindern würde, dass ein Benutzer mit beliebig vielen Identitäten das Netzwerk betritt, um quasi kostenlos an zusätzliches Karma zu gelangen, ist nicht vorhanden. Um dies zu verhindern, wäre eine PKI mit Zertifizierungsstelle (und somit einer zentralen Komponente) nötig, die die Identitäten der Benutzer prüft, was für die meisten Content Distribution Szenarien unpraktikabel ist.

4.7 e-cash

Mit e-cash wird in [MPA07] versucht, ein System wie BitTorrent mit starker Fairness und Anreizen zum Leisten von Beiträgen auszustatten, das auf Guthaben basiert. Ziele von e-cash sind unter anderem anonymes Bezahlen, Nichtverknüpfbarkeit eines e-coin mit dem zahlenden Benutzer und Fälschungssicherheit. E-cash stellt Anonymität dadurch sicher, dass e-coins nicht zu dem Benutzer zurückzuverfolgen sind, der sie ausgegeben hat, auch nicht von der zentralen Bank, zumindest solange nicht versucht wird, den e-coin doppelt auszugeben. Die Bank kontrolliert den Zahlungsverkehr und kann doppeltes Ausgeben und gefälschte e-coins erkennen. Transaktionen können online stattfindet, in welchem Fall der Zahlungsempfänger bei jeder Transaktion die Bank kontaktiert. Dadurch ist die Anonymität des Senders aber nicht mehr gegeben. Daher wird die offline-Variante bevorzugt, bei der die Transaktion ohne die Bank abgewickelt wird. Der Empfänger kann zu einem späteren Zeitpunkt die erhaltenen e-coins bei der Bank deponieren, welche die Coins auf Betrug prüft. Da Daten gegen e-coins ausgetauscht werden und jede Seite betrügen könnte, gibt es für den Disputfall einen vertrauenswürdigen Schlichter, der nicht die Bank sein muss. Jeder Benutzer hat bei der zentralen Bank ein Konto, von der er mehrere e-coins auf einmal abheben kann, die er sodann ausgeben kann. Beim Erhalt von e-coins müssen diese zunächst bei der Bank deponiert werden, um sie wieder ausgeben zu können. Wenn Alice einen e-coin gegen einen Datenblock von Bob tauschen möchte, sind fünf Schritte nötig:

- Bob wählt einen zufälligen Schlüssel, verschlüsselt den von Alice gewünschten Block und sendet ihr diesen.
- Alice erzeugt einen zufälligen Schlüssel, verschlüsselt ihren e-coin und erzeugt einen Kontrakt über den e-coin und Hashes über die gewünschten Daten und den erhaltenen verschlüsselten Block, über den der Vermittler ggf. die Transaktion nachvollziehen kann. Der Kontrakt und der erzeugte Schlüssel werden mit dem öffentlichen Schlüssel der Vermittlers verschlüsselt. Die Vermittlerdaten, der verschlüsselte e-coin und der Kontrakt werden an Bob geschickt.
- Bob prüft, ob die Daten von Alice formal korrekt sind und, falls ja, sendet Bob den Datenschlüssel an Alice.
- Wenn Alice die Daten entschlüsseln kann, sendet sie den Schlüssel für den e-coin an Bob, ansonsten ruft sie den Vermittler an.
- Erhält Bob den Schlüssel nicht, ruft er mit Hilfe der von Alice mitgesendeten, verschlüsselten Vermittlerdaten und des Datenschlüssels den Vermittler an.

Weigert sich Alice, Bob den e-coin trotz erfolgter Datenübertragung zu übergeben, kann Bob durch Senden des Datenschlüssels an den Vermittler beweisen, dass sich die gesendeten Daten entschlüsseln lassen. Der Vermittler kann zufällige Teile von beiden Parteien anfragen und prüfen und überträgt den e-coin an die berechtigte Seite. Wenn der Empfänger den Datenschlüssel nicht freigibt, wird der e-coin weder durch den Sender noch den Vermittler auf ihn übertragen. Wichtig ist hierbei, dass der Vermittler nur im Konfliktfall eingeschaltet wird, anderfalls aber nichts von der Transaktion mitbekommt, womit der Nachrichtenaufwand minimal bleibt. Beim Scheitern einer Transaktion verschlüsselt Alice ihren e-coin mit einem neuen Schlüssel, damit keine Verbindung zwischen der gescheiterten Transaktion und einer späteren Transaktion mit demselben e-coin hergestellt werden kann. Im Paper wird auch ein ähnliches Verfahren präsentiert, das benutzt werden kann, wenn Alice und Bob jeweils für den anderen interessante Daten zum Austauschen haben. In dem Fall hinterlegen beide beim jeweils anderen einen e-coin, der eingelöst wird, falls einer der beiden sich nicht an den fairen Austausch hält.

Wie die Autoren bei ihrer Evaluation selbst festgestellt haben, skaliert e-cash nicht bei größeren Systemen, da die zentrale Bank jeden e-coin beim Deponieren einzeln verifizieren muss, wobei für jeden Coin ungefähr genauso viel Rechenaufwand nötig ist, wie der Zahlungsempfänger während des Protokollablaufs aufwenden muss, was die Autoren mit 1,5 Sekunden auf einer Pentium M 1.6 GHz CPU angeben. Der Einstieg ins Netzwerk ist eine weitere Hürde. Neue Benutzer erhalten kein Startguthaben, sondern sind auf soziale Kontakte angewiesen, die ihnen entweder e-coins oder Dateien als Startkapital bereitstellen oder sie müssen sich beispielsweise bei der Bank e-coins gegen echte Währung kaufen. Solch ein Modell dürfte auf viele Benutzer abschreckend und als zu kompliziert wirken. Zudem besteht die Gefahr, dass einige

Teilnehmer, die deutlich mehr Ressourcen bereitstellen als sie verbrauchen, langfristig andere Teilnehmer behindern, weil sie kein ausreichendes Guthaben aufbauen können. Es gibt zwar Systeme, die sich einigen der Probleme annehmen, wie z. B. PPay [PPA03], jedoch sind die meisten auf virtuellen Währungen basierenden Systeme, mit Ausnahmen wie Bitcoin [BTC08], von einer zentralen Kontrollstelle abhängig und somit für komplett dezentrale Content Distribution Systeme ungeeignet.

4.8 One Hop Reputation

Die Autoren von [OHR08] haben umfangreiche Messungen einiger zehntausend BiTorrent-Schwärme vorgenommen und haben dabei folgende Beobachtungen gemacht:

- Die mittlere Downloadrate in BitTorrent-Netzen beträgt lediglich 14KBps bei 100KBps Uploadrate.
- Es gäbe genug Kapazität für bessere Performanz, wenn Nutzer auch nach Abschluss des Downloads einen Anreiz hätten, im Schwarm zu bleiben.
- 80% der Bandbreite stammt von 10% der Benutzer, die meist hohe Kapazitäten haben.
- Die Wahrscheinlichkeit, dass zwei Peers, die in einem Torrent direkt miteinander interagieren, dies in einem anderen Torrent erneut tun, ist kleiner als 10%.
- Die meisten Peers sind aber indirekt über andere intermediäre Knoten zeitversetzt und schwarmübergreifend miteinander verbunden.

One Hop Reputation soll solche Indirektionen finden und nutzen, indem Clients nicht nur lokal verfügbare Informationen nutzen, die aus Datentransfers über Direktverbindungen hervorgehen, sondern auch Informationen von Nachbarn. Dazu tauschen Peers beim Verbindungsaufbau und während Datentransfers Nachrichten aus. One Hop Reputation limitiert das Propagieren von Informationen auf direkte Nachbarn, um den Overhead zu begrenzen. Der Zweck hiervon ist, dass Knoten A eine Menge von Daten an einen Zwischenknoten I überträgt, I wiederum an Knoten B, worauf B sich bei A revanchiert. Knoten speichern Übertragungsvolumina zu anderen Peers über einen längeren Zeitraum, sodass die indirekte Erwiderung von Traffic auch erst deutlich später stattfinden kann. Clients haben jeweils ein Public/Private-Schlüsselpaar als Identität und speichern Information über Peers, mit denen sie direkt (in Form von Datentransfers) oder indirekt (in Form von Zwischenknoten, die das Verhalten anderer attestieren) interagiert haben. Jeder Client ordnet seine Liste bekannter Peers nach einem Kriterium (z.B. Häufigkeit von Begegnungen) und bildet eine Menge aus den oberen k Knoten. Diese wird beim Verbindungsaufbau zwischen zwei Peers A und B mit dem jeweils anderen ausgetauscht. Es wird die Schnittmenge zwischen der empfangenen Liste und den lokal bekannten Knoten gebildet, um gemeinsame Zwischenknoten zu finden. Für jeden so gefundenen Zwischenknoten I wird beim entfernten Knoten ein Beleg für die mit I ausgetauschte Datenmenge angefordert. Dieser Be-

leg ist mit einem Zeitstempel und der Signatur von I versehen. Gibt es mehrere gemeinsame Zwischenknoten I, wird die zwischen A und B transferierte Datenmenge anteilsmäßig auf diese verteilt. Den Zwischenknoten I werden regelmäßig Update-Nachrichten geschickt, um ihnen mitzuteilen, welche Datentransfers ihnen zugeschrieben werden und um Mehrfachverwendung von Belegen zu vermeiden. Auf dieser Basis sind viele verschiedene Policies möglich, um zu entscheiden, welche Anfragen bedient werden. Beispielsweise kann zunächst nur die durch direkte Interaktion zwischen A und B entstandene Differenz an übertragenem Datenvolumen (falls vorhanden) abgegolten werden, bevor indirekt über I entstandene Schulden zum Tragen kommen.

Es sind einige Angriffe auf One Hop Reputation möglich, wie das Verbessern oder Verschlechtern des Standes von Peers oder Zwischenknoten durch Sybil-Attacken und zusammenarbeitende Knoten. Die Autoren argumentieren jedoch, dass deren Effektivität eingeschränkt ist, da Peers erst einmal eine hohe Reputation durch legitimen Austausch aufbauen und auch erhalten müssten. Zwischenknoten I müssen aktiv sein, während die Knoten A und B über I entstandene Schulden ausgleichen, um Updates mit dem Sender auszutauschen; ein Nachteil, der die Effektivität beeinträchtigen könnte. Bei jedem Verbindungsaufbau zwischen zwei Peers werden k öffentliche Schlüssel mit je 128 Bit versendet, wobei $k = 2000$ vorgeschlagen wird. Dadurch müssen beim Verbindungsaufbau jeweils etwa 32KiB in jede Richtung versendet werden. Bei 1 Mbit Verbindungsgeschwindigkeit benötigt also jeder Verbindungsaufbau mindestens 0,25 Sekunden, vorausgesetzt, dass keine anderen Übertragungen stattfinden, was bei einer P2P-Anwendung unwahrscheinlich ist. Bei ISDN-Geschwindigkeit wäre ein Verbindungsaufbau sogar nur höchstens alle 4 Sekunden möglich, was die Benutzbarkeit relativ stark einschränkt.

4.9 Simple Trust Exchange Protocol (STEP)

Ein relativ einfaches P2P-Reputationssystem wird in [STE06] vorgestellt. Beim Simple Trust Exchange Protocol (kurz: STEP) handelt es sich um ein Protokoll, das auf signierte Quittungen setzt, die belegen, dass ein Peer A eine bestimmte Leistung gegenüber Peer B erbracht hat.

Dazu besitzt jeder Peer ein Schlüsselpaar bestehend aus öffentlichem und privatem Schlüssel. Möchte Peer B eine Leistung (z.B. einen Download) von Peer A erhalten, so füllt er ein Token aus, das einer Quittung ähnelt. Der Aufbau des Tokens zeigt Abb. 4.2. Peer B füllt die Felder für die Identitäten beider Peers, den Zeitstempel und die zu übertragende Datenmenge aus. Das Rating-Feld bleibt zunächst unausgefüllt. Peer B signiert das Token und sendet es an A. A prüft das Token, sendet die gewünschten Daten, signiert das Token und sendet es wieder zurück an B. Wenn der Download in Ordnung war, trägt Peer B eine positive Bewertung im Feld Rating ein, andernfalls eine negative. B signiert das Token erneut und ersetzt damit seine alte Signatur.

Name	Description
CID	Consumer's identity
PID	Provider's identity
TS	Timestamp of token creation
Length	Length/Duration of service (e.g. File size)
Rating	Consumer's rating ({*good*,*bad*})
CSig	Consumer's Signature
PSig	Provider's Signature

Abbildung 4.2: Aufbau eines Tokens in STEP [STE06, Seite 2]

Das Token als Quittung für den Datentransfer wird über das Gnutella-Netzwerk per Flooding verbreitet.

Jeder Peer priorisiert eingegangene Anfragen basierend auf der Reputation der anfragenden Peers, die sich aus den bisher erhaltenen positiven und negativen Tokens ergibt. Wie die Tokens zu bewerten sind, wird im Paper nicht genauer definiert. Als Beispiel werden aber alle positiven Tokens aufsummiert und negative Tokens subtrahiert.

Neue Peers erhalten immer die minimal mögliche Reputation, damit ein häufiger Wechsel der Identität durch Generierung eines neuen Schlüsselpaares unattraktiv wird. Jedoch ist das Protokoll offen für einige einfache Manipulationsmöglichkeiten. Beispielsweise lässt sich die eigene Reputation sehr leicht verbessern, indem mehrere Peers miteinander kollaborieren und sich gegenseitig Tokens ausstellen, die ihnen einen positiven Beitrag bescheinigen. Auch fehlt für den Empfänger einer tatsächlich erbrachten Leistung eine Motivation, die ihn dazu bewegt, eine Bewertung, die zweite Signatur und die Verbreitung des Tokens zu leisten. Beim Konkurrieren um Ressourcen dritter Peers kann es dem Leistungsempfänger gelegen kommen, wenn der Sender keine verbesserte Reputation erhält (oder sogar eine verschlechterte durch Ausstellen einer falschen, negativen Quittung).

4.10 FairSwarm.KOM

In der Doktorarbeit [PAP11] wird ein vollständig dezentrales Belohnungssystem vorgestellt, dessen Ziel es ist, langfristige Anreize zum Bereitstellen von Bandbreite zu liefern, die, im Gegensatz zu anderen Ansätzen, zum Seeden fertiggestellter Torrents motivieren und somit Content verfügbarer halten. FairSwarm.KOM setzt sowohl auf direkte als auch auf indirekte Erwiderung beigetragener Bandbreite. Bandbreite, die ein Knoten A einem Knoten B zur Verfügung stellt, kann durch einen Knoten C erwidert werden. Indirekte Erwiderung funktioniert, indem Knoten A nützliche Daten an B sendet, B an C und C an A. Damit A und C erkennen kön-

nen, dass eine Kette von Datentransfers über Knoten B sie verbindet, sendet einer der beiden (z. B. A) dem anderen (C) eine Liste mit Peers, mit denen er zuvor in Verbindung stand. Letzterer sucht im lokalen Datenbestand nach Coupons, die einen der Peers in der Liste enthalten, und sendet sie als Antwort. Der Sender, hier A, kann nun erkennen, dass C eine indirekte Schuld gegenüber A hat und kann diese einfordern. Einige Coupons, nämlich diejenigen, die ein Knoten am meisten versucht sein könnte zu unterdrücken, werden per Gossiping an direkte Nachbarn propagiert, um Täuschungsversuche besser erkennen zu können. Falls ein Peer B einen Coupon C_{AB} für einen stattgefundenen Datentransfer zwischen A und B unterschlägt, kann Peer C dies erkennen, wenn er C_{AB} bereits kennt (weil z. B. durch A erhalten) und B den Coupon in seiner Antwort nicht mitsendet. Gleiches gilt für veraltete Versionen von Coupons, die B senden könnte, weil er darin besser dasteht. Das Erzeugen von Coupons bei einem Datentransfer funktioniert sehr ähnlich, wie das Übertragen eines e-coins im e-cash System (siehe 4.7), nur dass anstelle des Kontrakts der Coupon und anstelle des e-coins die Signatur des Coupons tritt. Nach diesem Modell kann ein Benutzer sich selbst mittels mehrerer Identitäten Coupons ausstellen, die einer seiner Identitäten einen guten Stand zusprechen. Um den Effekt davon zu minimieren kommt das Max-Flow-Min-Cut-Theorem zum Einsatz, das besagt, dass der maximale Fluss in einem Netzwerk so groß ist, wie sein Flaschenhals. Konkret bedeutet dies: Wenn Knoten A einen weiteren Knoten A' kontrolliert, A' laut Coupons dem Knoten A 100 Einheiten an Daten schuldet (die nie tatsächlich übertragen wurden), B den (für ihn unterschiedlichen) Knoten A 5 Einheiten und A' 1 Einheit schuldet, kann A trotzdem nur die Schulden einfordern, die B gegenüber seinen direkten Nachbarn gemacht hat, also 6.

Das System erscheint gut durchdacht. Der einzige erkennbare Schwachpunkt (wie bei allen System, die Beiträge langfristig honorieren) ist, dass sich ein Benutzer mit Mietserver schnell eine gute Reputation aufbauen kann, von der er dann längere Zeit zehren kann, ohne einen weiteren Beitrag zu leisten. Aber gerade Peers mit hoher Bandbreite, die deutlich mehr hoch- als herunterladen tragen zu performanten Downloads im Schwarm bei. Es sollte also eher ein kontinuierlicher Beitrag belohnt werden, als ein einmaliger extrem höher.

4.11 Zusammenfassung

Einige der betrachteten Belohnungssysteme eigenen sich aufgrund ihrer Eigenschaften nicht bzw. nur eingeschränkt für P2P Content Distibution, wie z.B. BAR Gossip, das nicht die P2P-typische Dynamik neuer und das Netzwerk verlassender Peers unterstützt und Dadelion oder e-cash, die eine zentrale Instanz benötigen.

Die auf Krediten basierenden Verfahren belohnen immer nur den absoluten Beitrag eines Peers, fördern aber weniger den konstanten, regelmäßigen Beitrag, sodass es attraktiv ist, sich durch einen hohen, frühen Beitrag ein sicheres Reputationspolster aufzubauen, das später ge-

nutzt werden kann, ohne dann noch einen Beitrag zu leisten. Dass Kreditsysteme langfristig nicht zu einem gerechteren Verteilungsverhalten führen zeigen die Anti Leech Tracker (ALT) bei BitTorrent. Liegt dort die vom Tracker erfasste heruntergeladene Datenmenge eines Benutzers signifikant über der hochgeladenen wird das Benutzerkonto gesperrt und der ALT kann nicht mehr verwendet werden. Als Gegenmaßnahme haben sich immer mehr Benutzer Server mit 100 Mbit/s Anbindung gemietet, weil sie mittels herkömmlicher DSL-Anschlüsse nicht ausreichend schnell Daten hochladen konnten, um einen positiven Kontostand zu halten. Benutzer, die bei diesem Wettrüsten nicht mithalten konnten, wurden vom System ausgeschlossen. Ein ähnliches Risiko besteht bei allen kreditbasierten Belohnungssystemen, die absolute Traffic-Zahlen langfristig speichern.

Das System, das die zu Beginn dieses Kapitels genannten Kriterien am besten erfüllt, ist das von FairTorrent. Es benötigt keine zusätzliche Kommunikation, sondern verwendet nur beim Datenaustausch mit Nachbarn ohnehin anfallende Informationen, gehört also in die Kategorie der Systeme mit direkter Erwiderung. Dadurch ist es äußerst gut vor Manipulation durch Dritte geschützt, erlaubt aber trotzdem eine faire Priorisierung tauschwilliger Peers. Ein Nachteil ist jedoch, dass Peers nach dem vollständigen Download keinen signifikanten Anreiz mehr für das Hochladen haben, es sei denn es gibt Überlappungen der Peers in den Schwärmen, sodass die während des einen Downloads aufgebaute Reputation beim anderen für einen schnelleren Download genutzt werden kann. Von Vorteil ist jedoch, dass kein langfristiges Polster aufgebaut werden kann und keine Profilbildung über das gesamte Übertragungsvolumen eines Peers möglich ist, da diese keine Identität besitzen.

5 Anonymität

Bestehende P2P-Systeme bieten oft keine Möglichkeit, zu verschleiern, wer mit wem Dateien austauscht. Deshalb versuchen Nutzer sich Abhilfe zu verschaffen, indem sie den Datenverkehr über VPN-Verbindungen oder Anonymierunsnetzwerke wie TOR oder I2P leiten. In der Praxis führt das zu sehr geringen Übertragungsraten und einer oft nur vermeintlichen Anonymität, da weder die Anonymierunsnetzwerke noch P2P-Programme für eine solche Verkopplung entwickelt wurden.

In diesem Kapitel wird bestimmt, was Anonymität überhaupt ist, welche Arten von Anonymität sich unterscheiden lassen und wo die Grenzen der Anonymität liegen. Es werden einige Systeme untersucht, die in aktueller Literatur beschrieben sind und die Anonymität in P2P-Netzen zum Ziel haben.

Mit den gewonnen Erkenntnissen soll eine optional nutzbare Anonymierungsfunktion für ByteStorm entwickelt werden, welche ein grundlegendes Maß an Anonymität bietet, aber gleichzeitig die Übertragungsraten nicht übermäßig beschränkt.

5.1 Was ist Anonymität?

Unter Anonymität versteht man intuitiv einen Zustand, in dem eine Handlung und das diese Handlung ausführende Subjekt nicht miteinander in Verbindung gebracht werden können. Die Handlung selbst oder das Resultat der Handlung kann offen einsehbar sein. Das handelnde Subjekt ist in den meisten Fällen eine Person, kann aber auch eine Gruppe oder Institution sein. In demokratischen Prozessen – insbesondere bei Wahlen – spielt Anonymität eine zentrale Rolle. Sie schützt die Beteiligten vor Diskrimierung und möglichen Sanktionen durch Dritte.

In verteilten Netzwerken kennen sich die beteiligten Parteien in der Regel untereinander nicht persönlich. Jeder Netzwerkteilnehmer kann daher ein potenzieller Gegner sein, der ein Interesse daran hat, die Quelle oder das Ziel einer Information aufzudecken und zu sanktionieren. Dies betrifft insbesondere Staaten, in denen der Internetzugriff starker Überwachung unterliegt und die Verbreitung bestimmter Informationen mit Repressalien und harten Strafen belegt ist. Betroffene Personen, z.B. Dissidenten oder Whistleblower, die derlei Informationen öffentlich machen wollen, können dies meist nur in anonymer Form tun. Eine Möglichkeit dafür sind Peer-to-Peer Netzwerke.

Jedes Peer-to-Peer Netzwerk setzt auf eine eigene Definition von Anonymität. Diese Definitionen unterscheiden sich im Wesentlichen in den Antworten auf folgende Fragen: Wessen Identität soll verborgen werden? Vor wem soll sie verborgen werden? In welchem Umfang soll Anonymität gewährleistet werden?

In der gängigen Literatur finden sich unterschiedliche Typen von Anonymität wieder, die sich nach den zu schützenden Teilnehmern richten. Allgemein lassen sich diese Schutzziele in die folgenden drei Klassen unterteilen:

1. Senderanonymität: Der Sender einer empfangenen Nachricht soll anonym bleiben.
2. Empfängeranonymität: Der Empfänger einer Nachricht soll anonym bleiben.
3. Nichtverknüpfbarkeit von Sender und Empfänger: Die Kenntnis der Identität des Empfängers einer Nachricht und der Nachricht selbst soll einem Angreifer keine Erkenntnisse über die Identität des Senders liefern und umgekehrt.

Oft werden diese Klassen miteinander kombiniert, um den Schaden im Falle einer Kompromittierung zu begrenzen. Insbesondere die Nichtverknüpfbarkeit wird in der Regel mit der Anonymisierung von mindestens einem der beiden Kommunikationspartner verbunden.

5.2 Anonymität vs. Glaubhafte Abstreitbarkeit

Anonymität im informationstechnischen Kontext schützt lediglich vor der Identifizierung der Quelle und/oder des Ziels einer Information. Wird eine der Parteien dennoch identifiziert, ist es ohne gezielte Gegenmaßnahmen relativ leicht möglich, nachzuweisen, dass diese Partei ein bestimmtes Datum angefragt, zur Verfügung gestellt, gespeichert oder Kenntnis vom Inhalt der bei ihr (zwischen-)gespeicherten Daten hat. Glaubhafte Abstreitbarkeit (engl. plausible deniability) bezeichnet ein Konzept, das Spuren gezielt vermeidet, die derartige Tatsachen nachweisbar machen. Eine Person ist dadurch in der Lage, Sachverhalte mit plausibler Argumentation zu bestreiten. In einem Netzwerk, das Informationen hauptsächlich weiterleitet, kann ein Betreiber eines Knotens, der der Verbreitung einer bestimmten Information beschuldigt wird, dies mit der plausiblen Behauptung abstreiten, die Information lediglich im Auftrag eines Dritten weitergeleitet und keine Kenntnis des Inhalts zu haben.

5.3 Survey zur Anonymität in verschienen Systemen

In [SFS05] werden Peer-to-Peer Filesharing Systeme unter dem Gesichtspunkt Anonymität untersucht. Die populärsten Methoden eine gewisse Form von Anonymität zu gewährleisten werden beschrieben, eine Reihe implementierender Systeme vorgestellt und mögliche Angriffe untersucht.

Der Survey stellt fest, dass die meisten Peer-to-Peer Systeme Friend-to-Friend-Netzwerke sind. Dies sind solche Netzwerke, in denen jeder Knoten nur zu einer kleinen Anzahl bekannter Knoten eine direkte Verbindung aufbaut. Nur die direkten Nachbarn eines Knotens kennen dessen IP-Adresse. Nachrichten werden von Knoten zu Knoten anhand einer Pseudoadresse durch das Overlay-Netzwerk geroutet. Jeder Knoten leitet also Nachrichten für andere Knoten weiter, was ein gewisses Maß an Anonymität schafft, da der Betreiber eines Knotens glaubhaft behaupten kann, dass er lediglich Nachrichten und Dateien für andere Knoten weiterleitet. Die Pseudoadresse jedes Knotens ist leicht zu ermitteln, wenn er eine Datei sendet oder empfängt. Es ist jedoch schwer, eine Verknüpfung zwischen der Pseudoadresse und der richtigen IP-Adresse des Knotens herzustellen, zumal jeder Knoten jederzeit seine Pseudoadresse verwerfen und neue Adressen erzeugen kann. Einige Systeme enthalten jedoch Fehler, die es einem Angreifer dennoch erlauben, die IP-Adresse zu ermitteln. Im Allgemeinen erhöht sich die Wahrscheinlichkeit die Anonymität eines Systems bzw. einzelner Knoten zu gefährden, je mehr Angreifer dem Netzwerk beitreten. Wenn der Angreifer wählen kann, an welchen Punkten er dem Netzwerk beitritt, hat er die Möglichkeit, einen Knoten zu umstellen und damit seinen kompletten Nachrichtenverkehr zu überwachen. Die meisten Systeme bieten keinen Schutz, wenn der Angreifer in der Lage ist, die komplette Netzwerkkommunikation zu überwachen.

Im Folgenden sind die im Paper untersuchten Anonymisierungsverfahren beschrieben:

- **Ants** wurde als Protokoll für ad-hoc Netzwerke entwickelt. Jeder Knoten wählt eine Pseudoidentität unter der er Nachrichten versenden kann, ohne seine Identität preiszugeben. Die Suche im Netzwerk erfolgt per Broadcast an alle Nachbarn, die wiederum an ihre Nachbarn broadcasten, bis der in der Nachricht enthaltene Time-to-live Counter abgelaufen ist. Jeder Knoten merkt sich beim Empfang einer Nachricht die Pseudoadresse des Senders, die Verbindung über die die Nachricht eintraf und wie oft Nachrichten dieser Pseudoadresse über diese Verbindung eingetroffen sind. Beim Routing einer Nachricht zu einer Pseudoadresse wird diese über diejenige Verbindung weitergeleitet, über die die meisten Nachrichten von dieser Adresse eingegangen sind. Gibt es keinen Routingeintrag für die Zieladresse, wird die Nachricht an alle Nachbarknoten weitergeleitet. (genutzt von: Mute, ANTS, Mantis)
- **Onion Routing** erlaubt das Herstellen anonymer Verbindungen über öffentliche Netzwerke. Dazu wendet es das Zwiebelschalenprinzip an. Dem Sender ist die Empfängeradresse bekannt. Zusätzlich wählt er zufällig weitere Knoten, sog. *Core Onion Routers (CORs)*, aus, die die Route bilden. Jeder dieser CORs und der Empfänger besitzen einen öffentlichen Schlüssel, die dem Sender bekannt sein müssen. Die Nachricht wird rekursiv unter Verwendung der öffentlichen Schlüssel der CORs zusammen mit Adressinformationen des jeweils nächsten Routers verschlüsselt. Erhält ein Router eine Nachricht, dann „schält“ er

eine Schicht ab, indem er die Nachricht mit seinem privaten Schlüssel entschlüsselt. Er sieht nun eine wiederum verschlüsselte Nachricht und die Adressinformationen des nächsten Routers, an den er die Nachricht weiterleitet. Ein Router erhält also immer nur Kenntnis vom nächsten Router, nicht jedoch vom Empfänger oder Inhalt der Nachricht, es sei denn er ist der Empfänger. (Anonymous Peer-to-peer File-Sharing (APFS), Tor, I2P, SSMP)

- **Freenet** ist ein durchsuchbares Peer-to-Peer System, das den Fokus aus zensurresistentes Speichern von Dokumenten legt. Ziel von Freenet ist, es einem Angreifer unmöglich zu machen, alle Kopien eines Dokuments zu finden. Jeder Knoten speichert alle Dateien, die ihn passieren und löscht nur die am seltensten abgerufenen Dateien, wenn nötig. Dateien werden anhand eines Hashwertes über den Dateinamen bzw. Schlüsselwörter identifiziert. Jeder Knoten hält eine Liste der Hashes der Dateien aller umgebenden Knoten vor. Die Suche erfolgt anhand der Hashes. Suchanfragen werden von jedem Knoten an den Nachbarknoten weitergeleitet, der eine Datei besitzt, dessen Hash dem gesuchten Hash am nächsten kommt. Dadurch werden mit der Zeit Dateien mit ähnlichen Hashwerten auf Knoten gruppiert, was die Effizienz einer Suche steigert. (Entropy, Freenet, Nodezilla)
- **Return Address Spoofing** bezeichnet das Verwenden einer gefälschten Absenderadresse in IP-Paketen. Für das Routing von Paketen wird die Absenderadresse nicht benötigt. TCP benötigt die Absenderadresse für das Senden von Steuersignalen, UDP jedoch nicht. Somit lassen sich UDP-Pakete mit einer zufälligen Absenderadresse verschicken. Der Absender hat jedoch keinerlei Informationen darüber, ob ein Paket den Empfänger erreicht. Eine bidirektionale Kommunikation ist ohne zusätzlichen Aufwand nicht möglich. Zudem unterbinden viele ISPs Return Address Spoofing. (Mantis)
- **Broadcast**, auch Flooding, kann verwendet wurden, um die Identität des Empfänger zur verschleiern, sofern genügend Teilnehmer die Nachricht erhalten. Dazu leitet jeder Knoten eingehende Nachrichten an alle Nachbarn weiter, ggf. unter Berücksichtigung eines Time-to-live Counters. Zur Vermeidung von Weiterleitungsschleifen kann jeder Nachricht eine eindeutige ID hinzugefügt werden.
- **Crowds** ist ein anonymes Protokoll für Verbindungen mit Webservern, bei denen die Identität des Aufrufers verschleiert wird. Jeder Knoten ist mit jedem anderen Knoten innerhalb einer Gruppe von Benutzern, der Crowd, verbunden. Statt direkt den Server zu kontaktieren wird die Anfrage (verschlüsselt) an einen zufälligen Knoten weitergeleitet. Jeder Knoten, der eine Anfrage erhält, entscheidet durch einen Zufallswurf, ob er die Nachricht an den Server oder einen weiteren zufällig gewählten Knoten sendet. Die Wahrscheinlichkeit für letzteres ist dabei >50%. Somit ist die Wahrscheinlichkeit, dass der Knoten, der den Webserver letztendlich kontaktiert, der anfragende Knoten ist, nicht höher als die Wahrscheinlichkeit, dass die Anfrage von ihm selbst stammt. Da jeder Knoten mit jedem anderen verbunden ist, skaliert Crowds schlecht in größeren Netzwerken. (AP3)

- **MIXes** bieten Anonymität durch das Weiterleiten von Knoten zu Knoten. Empfangene Nachrichten werden von Knoten jedoch nicht sofort weitergeleitet, sondern, erst wenn eine gewisse Anzahl Nachrichten empfangen wurde, werden die Nachrichten zusammengefasst und gebündelt weitergeleitet. Je nach Implementierung ist eine Anonymisierung von Sender und Empfänger, aber auch der Sender-Empfänger-Beziehung möglich. Ein Problem für Filesharing Systeme ist allerdings die Asymmetrie der Daten beim Herunterladen von Dateien, da deutlich mehr Nachrichten in die eine Richtung fließen als in die andere. (Free Haven, GNUnet)
- **DC-Nets** sind rundenbasiert. Sie setzen voraus, dass jedes Paar von Teilnehmern einen Satz von n geheimen Verschlüsselungsschlüsseln ausgetauscht hat. Jeder Teilnehmer sendet in jeder Runde einen Broadcast mit einem Vektor aus n Nachrichten. Dieser wird per XOR aus Kombinationen erhaltener Nachrichten gebildet. Will ein Knoten eine Nachricht senden, fügt er sie an einer zufälligen Position in den Vektor ein, unter Verwendung der ausgetauschten Schlüssel. Solange die Schlüssel und die gewählte Position geheim bleiben, ist es nicht möglich, den Sender oder Empfänger festzustellen. (CliqueNet)

5.4 BitTorrent + Tor = FAIL

Eine prinzipiell einfache Möglichkeit, BitTorrent zu anonymisieren ist, den BitTorrent-Traffic über das Tor-Netzwerk zu routen, das nach dem Onion Routing Prinzip funktioniert. Es bietet die Möglichkeit, über einen Proxy TCP-Verbindungen beliebiger Anwendungen auf der Transportebene zu tunneln. Weder Tor noch BitTorrent wurden jedoch entwickelt, um miteinander gekoppelt zu werden. Die TOR-Entwickler betonen seit Jahren, dass BitTorrent-Traffic im Tor-Netzwerk zu Verstopfungen führt und dadurch die Performanz für alle Benutzer auf ein Minimum reduziert wird, was zum Teil an Defiziten des TOR-Protokolls liegt.[1] Die Implementierungen von Tor und verschiedenen BitTorrent-Clients können leicht zur Deanonymisierung des Benutzers führen. In [CTA10] werden dafür drei Angriffe beschrieben:

1. In vielen BitTorrent-Clients lässt sich ein Proxy für die Kommunikation mit dem Tracker (und anderen Peers) definieren, was zum Ziel hat, die eigene IP-Adresse gegenüber dem Tracker zu verschleiern. Einige Implementierungen übermitteln die eigene IP-Adresse jedoch in der Anfrage selbst an den Tracker, welcher wiederum die IP an andere Peers weitergibt, was die Proxy-Nutzung zwecks Wahrung der eigenen Anonymität wirkungslos macht. Zudem nutzen einige Tracker UDP als Transportprotokoll. Socks Proxy Server – wie der von Tor – unterstützen allerdings nur TCP-Verbindungen. BitTorrent-Client-Entwickler haben sich für den Fall, dass ein Tracker UDP nutzt, für den benutzerfreundlichen Weg entschieden: Sie ignorieren den konfigurierten Proxy und senden die Anfrage direkt an den

[1] https://blog.torproject.org/blog/why-tor-is-slow

Tracker, anstatt sie fehlschlagen zu lassen, sodass auch in diesem Fall die eigene IP-Adresse trotz konfiguriertem Proxy bekannt wird.

2. Der zweite Angriff basiert auf dem vorherigen. Verbindungen zu anderen Peers werden in diesem Szenario über das Tor-Netzwerk geroutet. Viele Clients wählen den Port, über den sie erreichbar sind, zufällig aus, z.B. 4223. Kennt der Tracker aufgrund der oben genannten Lücken die richtige IP-Adresse eines Peers, kann ein Angreifer die IP-Adresse leicht ermitteln, auch wenn er sich nur über Tor mit dem Peer verbinden kann. Mit der Kenntnis des Ports, auf dem das Opfer auf eingehende Verbindungen wartet, kann der Angreifer eine Anfrage nach weiteren Peers an den Tracker stellen. Mit einer gewissen Wahrscheinlichkeit ist dem Tracker nur ein Peer bekannt, der den Port 4223 verwendet. Die dazu gehörende IP-Adresse ist die echte Adresse des Peers. Der Angreifer hat somit mit hoher Wahrscheinlichkeit die vermeintliche Anonymität des Peers gebrochen.

3. Der dritte Angriff begründet sich in der Funktionsweise von Tor. Damit nicht für jede einzelne Verbindung eine neue Route erzeugt werden muss, werden mehrere TCP-Verbindungen über dieselbe Route geleitet. Die Route wird nur etwa alle 10 Minuten geändert. Der letzte Knoten der Route, der Exit-Knoten, vermittelt zwischen dem Tor-Knoten, der die Route aufgebaut hat, und dem normalen Internet. Dazu muss der Exit-Knoten alle übertragenen Daten im Klartext kennen. Da mehrere ausgehende Verbindungen eines Knotens dieselbe Route nehmen, kann der Exit-Knoten den Kommunikationsinhalt aller Verbindungen des Ursprungsknotens mitlesen. Wenn nun der Benutzer während einer laufenden BitTorrent-Übertragung über Tor auch noch unverschlüsselte Webseiten abruft, E-Mails liest oder Instant Messaging benutzt, kann der Exit-Knoten dies natürlich ebenfalls mitlesen. Sind in einem dieser Datenströme Informationen enthalten, die den Benutzer identifizieren, so weiß der Exit-Knoten, dass alle übrigen Verbindungen ebenfalls zu diesem Benutzer gehören.

Diese Fallen, die sich aus Implementierungsdetails beider Systeme ergeben, sind von durchschnittlichen Benutzern kaum zu erkennen. Benutzer, die auf Anonymität angewiesen sind, sind unwissend der Gefahr ausgesetzt, enttarnt zu werden. BitTorrent wurde nie mit einem Anspruch auf Anonymität entwickelt. Genausowenig wurde Tor entwickelt, um große Datenmengen zu übertragen, sondern hauptsächlich für anonymes, zensur-resistentes Surfen im Web. Wenn Anonymität ein wichtiges Kriterium bei der Datenübertragung ist, sollte sie bereits beim Protokollentwurf berücksichtigt werden, um derartige Fallen zu vermeiden.

5.5 Tarzan

Tarzan ([TAR02]) hat ein ähnliches Ziel wie Tor: Es bietet Anonymität auf der Transportschicht. Dadurch kann Tarzan transparent gegenüber Applikationen agieren. Die Daten werden über einen Tunnel per Onion Routing weitergeleitet, wobei ein Endpunkt (Sender oder Empfänger) auch außerhalb des Tarzan-Netzes liegen kann. Dafür ist jeder Knoten im Netzwerk gleichzeitig ein NAT-Router ins freie Internet. Daten werden schichtenweise symmetrisch verschlüsselt, wobei die Schlüssel beim Routenaufbau übermittelt werden. Eine zentrale Instanz, die alle Knoten und deren öffentliche Schlüssel kennt, gibt es nicht. Stattdessen wird ein Gossip-Algorithmus verwendet, der mit hoher Wahrscheinlich innerhalb kurzer Zeit dazu führt, dass jeder Knoten alle anderen kennt. Um die Anonymität von Sender, Empfänger und deren Nichtverknüpfbarkeit sicherzustellen, trifft Tarzan einige (im Vergleich zu Tor) zusätzliche Maßnahmen. Sollte ein Angreifer sowohl Eintritts- als auch Austrittsknoten einer Route kontrollieren, kann er aufgrund des Zeitverhaltens der Pakete die Beziehung zwischen Sender und Empfänger herstellen. Es wird davon ausgegangen, dass ein Angreifer, der mehrere Knoten betreibt, dies unter IP-Adressen mit demselben Präfix tut. Um dagegen zu schützen, teilt Tarzan den IP-Adressraum in Domänen (Subnetze) ein. Da die Größe eines Subnetzes aufgrund der heterogenen Verteilung des IP-Adressraums nicht einfach zu ermitteln ist, entspricht jede Domäne einem /16 Subnetz, welchem /24 Subnetze untergeordnet sind. Bei der Routenkonstruktion wählt der Sender nur maximal einen Knoten aus jeder Domäne aus, um das Risiko kollaborierender Ein- und Austrittsknoten zu reduzieren. Um einem Angreifer mit Einsicht in den kompletten Netzwerkverkehr die Deanonymisierung zu erschweren, wählt sich jeder Knoten k semifeste Nachbarn aus. Mit diesen tauscht er bidirektional zufallsgenerierten Cover Traffic aus, so dass die eingehende und ausgehende Datenmenge gleich bleibt. In diesen bettet er die Nutzdaten ein, was dazu führt, dass die Knotenauswahl für die Routenkonstruktion auf Knoten beschränkt ist, mit denen Cover Traffic ausgetauscht wird, dem Knotenbetreiber aber die glaubhafte Abstreitbarkeit der Existenz einer Kommunikation ermöglicht. Die k Nachbarn werden nicht zufällig gewählt, sondern pseudozufällig durch k-fach rekursives Anwenden einer Hashfunktion über die eigene IP, Dömane und das aktuelle Datum. Da jeder Knoten alle anderen kennt, ist jeder Knoten in der Lage, die k Nachbarn eines Knotens nachzuvollziehen und dementsprechend gültige Routen zu bilden.

5.6 Hordes

Ein Protokoll, das Senderanonymität und Nichtverknüpfbarkeit bietet, ist Hordes ([PAC00]). Es kombiniert die Ansätze von Crowds mit Multicasts und erhält dadurch die Nichtverknüpfbarkeit von Sender und Empfänger aufrecht, auch wenn die Anonymität des Senders nicht mehr gegeben ist. Für den Beitritt zum Netzwerk sendet der beitretende Knoten (Initiator) einem Mitglied des Netzwerks eine Nonce und seinen Public Key. Letzterer erzeugt ebenfalls eine Nonce,

welche er, zusammen mit der Nonce des neuen Knotens, signiert zurücksendet. Der Initiator sendet wiederum beide Nonces signiert zurück. Darauf übermittelt der bestehende Knoten eine Liste mit allen Knoten des Netzwerks inklusive Public Keys und informiert alle anderen Knoten per Multicast über den neuen Knoten und seinen Public Key. Hordes tunnelt Requests, ähnlich wie Crowds, durch zufällige Entscheidung jedes Knotens, ob er den Request an den Empfänger zustellt oder nochmals weiterleitet. Für die Weiterleitungen wählt jeder Knoten eine Teilmenge von Knoten aus. Jedem der ausgewählten Knoten sendet er einen symmetrischen Schlüssel, der mit dem jeweiligen Public Key verschlüsselt und dem eigenen privaten Schlüssel signiert ist. Der symmetrische Schlüssel wird für die Knoten-zu-Knoten-Verschlüsselung verwendet. Ein Request enthält die Empfängeradresse, eine zufällige Id, die Adresse einer Multicast Gruppe und die Requestdaten. Die Multicast-Adresse wird vom Sender ausgewählt. Der Sender wird temporäres Mitglied der entsprechenden Multicast-Gruppe, um die Antwort empfangen zu können. Damit nicht jeder Knoten alle Nachrichten erhalten muss, gibt es mehrere Multicast-Gruppen. Die Anzahl der vorhandenen Multicast-Gruppen richtet sich nach der Größe des Netzwerks, denn eine Multicast-Gruppe mit nur einem Mitglied bietet keine Anonymität. Der Empfänger eines Requests sendet seine Antwort unter Verwendung der im Request enthaltenen Id und Adresse an die Multicast-Gruppe. Der Sender erkennt die Antwort an der von ihm definierten Id.

Da IP-Multicast im Internet nicht verfügbar ist, lässt sich Multicast nur auf Applikationsebene betreiben. Die bedeutet, dass der Sender per Unicast jedem Mitglied der Gruppe eine Kopie der Nachricht senden muss. Die benötigte Bandbreite zum Senden einer Nachricht multipliziert sich also mit der Anzahl Mitglieder der Gruppe. Für die Übermittlung kleinerer Nachrichten mag dies interessant sein, bei der Übertragung großer Dateien an viele Teilnehmer würden die Übertragungsraten aufgrund der übermäßig redundanten Übertragungen jedoch auf inakzeptable Werte fallen.

5.7 PKI-loses Onion Routing

Ein an Onion Routing angelehntes Anonymisierungsverfahren wird in [STO05] beschrieben. Er kombiniert Information Slicing und Source Routing und ermöglicht damit anonyme Kommunikation ohne eine Public-Key-Infrastruktur. Es wird angenommen, dass dem Sender mehrere IP-Adressen zur Verfügung stehen, über die er Pakete senden kann. Der Sender erzeugt n disjunkte Pfade der Länge l, indem er zufällige Knoten auswählt und sie auf die Pfade verteilt. Der Empfänger der Nachricht liegt auf einem der disjunkten Pfade an beliebiger Position, also nicht notwendigerweise am Pfadende. Jeder Knoten muss in der Lage sein, wie beim Onion Routing, die IP-Adressen der Nachfolgerknoten zu ermitteln, die sich auf derselben Stufe befinden, aber keine der anderen.

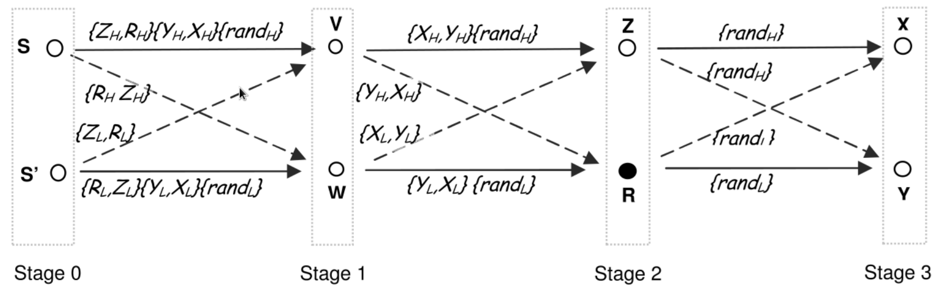

Figure 2—An example of anonymous routing with information slicing. Nodes S and S' are controlled by the sender. A message like $\{Z_l, R_l\}$ refers to the low-order words of the IDs of nodes Z and R, $rand$ refers to random bits.

Abbildung 5.1: Anonymes Routing mit Information Slicing [STO05, Seite 2]

Abbildung 5.1 zeigt anhand eines Beispiels mit zwei disjunkten Pfaden der Länge drei das Routing einer Nachricht. Der Sender kontrolliert S und S' und möchte eine Nachricht an R senden. Zu jedem Router auf dem Pfad sind die IP-Adressen bekannt. Die IP-Adressen werden in jeweils zwei gleichgroße Vektoren V_l und V_h zerteilt und mit einer invertierbaren Matrix M multipliziert, um zu verhindern, dass beliebige Knoten Kenntnis über die IP-Präfixe beteiligter Knoten erlangen können. Die resultierenden Vektoren V_L und V_H werden auf die beiden Nachrichten verteilt, die über die disjunkten Pfade verschickt werden. Die so kodierten IP-Adressen werden in Slices geteilt. Jede Nachricht enthält l Slices, eines für jede Stufe auf dem Pfad. Ein Slice enthält einen der errechneten Vektoren für jeden Knoten auf der nachfolgenden Stufe und die für die Berechnung des Vektors relevante Matrixzeile von M. Da jeder Knoten von seinem Vorgänger nur einen Teil der Adresse des Nachfolgers erhält, muss er auf andere Weise Kenntnis über die restlichen Teile erlangen, um die Adresse wiederherzustellen. Beim Empfang einer Nachricht liest ein Knoten immer das erste Slice, entfernt es und sendet den Rest der Nachricht an den Nachfolgerknoten weiter. Zusätzlich sendet er das erste Slice der Restnachricht an alle anderen Knoten, die sich auf derselben Stufe befinden, wie der Nachfolgerknoten. Dadurch erhält jeder Knoten auf einer Stufe alle Vektoren, um die IP-Adressen der Knoten auf der nächsten Stufe unter Verwendung der inversen Matrix M^{-1} zu berechnen. Da sich der Empfänger der Nachricht auf einer beliebigen Stufe befinden kann, muss diesem signalisiert werden, dass er der Empfänger ist. Ein entsprechendes Flag kann ebenfalls zerteilt und über die disjunkten Pfade versendet werden, sodass nur der Empfänger in der Lage ist, das korrekte Flag zu rekonstruieren.

Durch die Annahme, dass jeder Benutzer mehrere IP-Adressen zur Verfügung hat, ist das Verfahren in der Praxis wenig praktikabel, da die meisten Internetanschlüsse nur mit einer IP-Adresse ausgestattet sind. Mit steigender Anzahl disjunkter Pfade sinkt die Wahrscheinlichkeit gleichermaßen, dass die Nachricht den Empfänger nicht vollständig erreicht, weil ein Zwischenknoten nicht erreichbar oder fehlerhaft sein kann. Dadurch, dass das Weiterleiten eines

Slices den Erhalt einer Nachricht von allen Knoten auf der vorhergehenden Stufe bedingt, steigt die Wahrscheinlichkeit für ein Fehlschlagen der Übertragung weiter.

5.8 Anonymes Routing und Churn

Ein Problem beim anonymen, mix-basierten Routing über mehrere Zwischenrouter im praktischen Einsatz ist Churn, die Möglichkeit, dass ein Knoten jederzeit das Netzwerk verlassen kann oder anderweitig nicht mehr erreichbar ist. Fällt ein Knoten auf einer Route aus, so ist die komplette Route unterbrochen, so dass über diese keine weiteren Nachrichten mehr versendet werden können. Die Konstruktion neuer Routen, die einen ausgefallenen Knoten beinhalten, schlagen fehl. Da die Churn-Rate in populären P2P-Netzwerken relativ hoch ist, besteht auch eine hohe Wahrscheinlichkeit, dass Routen nach kurzer Zeit ausfallen oder gar nicht erst zustande kommen. Dieses Problem wird in [MPR07] aufgegriffen. Eine offensichtliche Lösung gegen ausgefallene Pfade ist Broadcasting an alle Nachbarknoten. Dies führt jedoch zu einer hohen zu sendenen Nachrichtenanzahl und ist daher in größeren Netzwerken nicht praktikabel. Stattdessen setzt der vorgestellte Ansatz auf Nachrichten- und Pfadredundanz, die zu einem moderateren Bandbreitenverbrauch führen. Nachrichtenredundanz wird durch den Einsatz von Erasure Codes erreicht. Diese erzeugen aus einer Nachricht M der Länge $|M|$ kodierte Segmente m mit der Länge $l = \frac{|M|}{n}$, wobei n Codesegmente ausreichen, um die Nachricht M wiederherzustellen. Daraus ergibt sich der Replikationsfaktor $r = \frac{n+x}{n}$ bei x zusätzlichen Codesegmenten. Für das Routing der Nachrichtensegmente werden k disjunkte Pfade gewählt. Dadurch liegt die Toleranzgrenze bei $k(1 - \frac{1}{r})$ ausgefallenen Pfaden. Durch Verändern von k und r kann zwischen der Zuverlässigkeit der Zustellung und der Bandbreitenkosten abgewogen werden. Um die Verfügbarkeit und Langlebigkeit von Pfaden zu erhöhen, werden Pfade nicht einfach aus zufällig gewählten Knoten konstruiert. Stattdessen wird versucht, eine Voraussage über die zukünftige Verfügbarkeit der einzelnen Knoten zu treffen. Je länger ein Knoten verfügbar ist, desto größer ist im Allgemeinen die Wahrscheinlichkeit, dass der betreffende Knoten auch noch eine Weile länger verfügbar bleiben wird [UCP06]. Knoten, die erst vor kurzem dem Netzwerk beigetreten sind, haben umgekehrt eine hohe Wahrscheinlichkeit, es bald wieder zu verlassen. Daher werden bevorzugt Knoten in die Routen aufgenommen, die sich bereits lange im Netzwerk befinden. Nach experimentellen Messungen erhöhte sich die durchschnittliche Rate für erfolgreich aufgebaute Ende-zu-Ende-Verbindungen dadurch von unter 10% auf über 95%.

5.9 Zusammenfassung

Aufgrund der Tatsache, dass P2P-Netze wie ByteStorm zum Übertragen großer Datenmengen bestimmt sind, ist ein geringer Overhead wichtig, um die Übertragungsraten nicht mehr als nötig einzuschränken. Anonymisierungsverfahren, die Broadcast oder Multicast verwenden, sind daher für das vorliegende Szenerio kaum geeignet. Wegen hoher Churn-Raten in allen größeren

P2P-Netzen würden Verfahren, die Datenpakete einer Verbindung über mehrere Routen verteilt zum Empfänger senden, zu kurzen und sehr häufig fehlschlagenden Verbindungen führen. Am attraktivsten erscheint Source-Routing, wie es u.A. TOR und Tarzan verwenden, da allein die Quelle für das Aufbauen einer geeigneten Tunnelroute verantwortlich ist. Um einen sicheren Tunnel aufbauen zu können, der nicht von Dritten eingesehen werden kann, ist jedoch eine Verschlüsselung aller Nachrichten nötig, wofür entsprechende Schlüssel nötig sind. Dazu ist ein Schlüsselaustausch erforderlich. Da das Finden von anderen Peers jedoch zu den Grundfunktionen jedes P2P-Systems gehört, liegt es nahe, Schlüsselaustausch und das Finden von Peers miteinander zu verquicken.

6 ByteStorm

Im Hauptteil dieser Arbeit wird ByteStorm vorgestellt, ein vielseitig einsetzbares Content Distribution System. Zunächst wird auf einige Arbeiten eingegangen, auf die ByteStorm aufbaut. Im zweiten Unterkapitel geht es um ein neues Konzept namens Tempest, das eine Menge von Dateien zu einer Einheit von Content zusammenfasst, darüber Prüfsummen bildet und durch Signaturen authentifizierbar ist. Danach wird vorgestellt, wie Tempests über ein verteiltes P2P-Netzwerk gefunden werden können. Unterkapitel vier zeigt, wie das aus einer Vorarbeit übernommene Verfahren zum Finden von Peers für einen P2P-Download für die Zwecke von ByteStorm modifiziert wurde. Wie die Datenübertragung zwischen diesen Peers erfolgt, beschreibt Unterkapitel fünf, gefolgt von einer Möglichkeit, um die Datenübertragung zu anonymisieren. Das siebte Unterkapitel stellt die Abonnementfunktion von ByteStorm vor, mit der sich, ähnlich wie Podcasts über RSS/Atom-Feeds, Tempests automatisch herunterladen lassen. Um ByteStorm zu nutzen, muss nicht jeder Beteiligter ein entsprechendes Programm installiert haben. Unterkapitel acht behandelt, wie per HTTP Dateien von Peers des Netzwerks heruntergeladen werden können und wie dabei automatisches Load Balancing zwischen den Peers erfolgt. Das letzte Unterkapitel beschreibt im Detail, wie das Kommunikationsprotokoll zwischen ByteStorm Peers aussieht und definiert Protokollnachrichten, um alle Funktionen von ByteStorm umzusetzen.

6.1 Vorarbeiten

In diesem Abschnitt werden Publikationen beschrieben, auf denen ByteStorm aufbaut. Dazu gehören das unstrukturierte P2P-Netzwerk BubbleStorm [BBS07], das Transportprotokoll CUSP [CUS10], eine Methode zum Publizieren und Suchen von Content sowie Tracking (das finden anderer Peers) über BubbleStorm [VSB10]. Letztere beiden sind das Resultat der Bachelorarbeit des Autors dieser Masterarbeit.

6.1.1 BubbleStorm

Dieser Abschnitt gibt eine kurze Zusammenfassung der Funktionsweise von BubbleStorm wieder, wie in [VSB10, Seite 7 ff.] respektive [BBS07] beschrieben. Der letzte Abschnitt beschreibt die neu hinzugekommenen Replikationsmodi, welche in [BRU12, Seite 41 ff.] genauer definiert werden.

BubbleStorm [BBS07] ist ein Peer-to-Peer Overlay-Netzwerk für das Speichern und Suchen von Daten in einem dezentral verteilten System. BubbleStorm verwendet eine unstrukturierte Netzwerktopologie und basiert auf probabilistischer, erschöpfender Suche sowie Replikation. Aufgrund der fehlenden Struktur können auch größere Ausfälle von Knoten schnell kompensiert werden.

BubbleStorms Topologie besteht aus einem heterogenen Zufallsgraphen. Die durchschnittlich übertragene Datenmenge kann über die Anzahl der Verbindungen zu Nachbarknoten reguliert werden. Beim Betreten eines bestehenden Netzwerks werden neue Verbindungen aufgebaut, indem eine bestehende Verbindung zwischen zwei anderen Netzwerkknoten durch zwei neue Verbindungen ersetzt wird, je eine vom neuen Knoten zu den beiden anderen Knoten. Beim Verlassen des Netzwerks passiert genau das Gegenteil: Verbindungen zu zwei Nachbarnknoten werden durch eine direkte Verbindung zwischen diese beiden Knoten ersetzt.

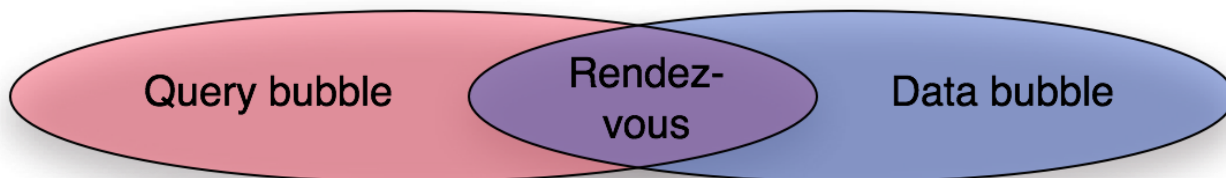

Abbildung 6.1: Aufeinandertreffen von Query Bubble und Data Bubble [BBS07, Seite 2]

Innerhalb von Bubbles werden beliebige Anfragen verteilt (z. B. Suchanfragen oder zu persistierende Daten). Bubbles bestehen aus einer Teilmenge von Netzwerkknoten. Alle Knoten im Bubble erhalten während eines sog. Bubblecasts die Anfrage. Der Bubblecast repliziert die Anfrage auf die Knoten des Bubbles, indem eine Mischung aus Random Walk und Flooding angewendet wird. Die Parameter eines Bubblecasts bestimmen, auf wieviele Knoten eine Anfrage repliziert wird und an wie viele Nachbarknoten jeder Knoten auf jedem Pfad die Anfrage sendet. Grob vereinfacht lassen sich zwei Arten von Bubbles unterscheiden: Data Bubbles und Query Bubbles. In Data Bubbles werden Anfragen transportiert, die andere Knoten anweisen, Daten lokal bei sich zu speichern. Query Bubbles versuchen Daten über eine Suchanfrage wieder abzurufen. Wie eine solche Suche erfolgt ist nicht durch BubbleStorm festgelegt. Persistierung und Suchalgorithmen müssen von der Anwendung implementiert werden. Dies erlaubt beliebig komplexe Anfragen. Auch eine verteilte SQL-Datenbank ist so denkbar. Suchanfragen sind immer genau dann erfolgreich, wenn der Bubblecast über eine Query Bubble einen Knoten erreicht, der einen passenden Datensatz kennt, den er zuvor über ein Data Bubble erhalten hat. Dieser Knoten wird als Rendezvous-Knoten bezeichnet. Formaler ausgedrückt gibt es also genau dann Treffer, wenn die Vereinigungsmenge eines Data Bubbles und eines Query Bubbles nicht

die leere Knotenmenge ist, es also mindestens einen Rendezvous-Knoten gibt. (siehe auch Abb. 6.1)

Es gibt derzeit vier Replikationsmodi für Bubblecasts, die sich an Benutzeranforderungen orientieren und für Suchanfragen und Datenreplikation gleichermaßen genutzt werden können:

- Instant Replication: Dieser Modus ist für alle Arten von Daten geeignet, die nicht vom Empfänger gespeichert werden sollen, sondern bei Erhalt von ihm nur verarbeitet werden, um beispielsweise Antworten zu erzeugen. Er eignet sich somit für das Senden sowohl von Queries als auch von Publikationen in Publish-Subscribe-Systemen.

- Fading Replication: Der Fading Modus ist für kurzlebige Informationen geeignet, die zwar von jedem Empfänger-Peer gespeichert werden, für die es aber nicht wichtig ist, dass sie für längere Zeit erhalten bleiben, z. B. weil sie nach kurzer Zeit veraltet sind. Mit jedem Knoten, der das Netzwerk verlässt, verschwindet eine Replika. Alternativ ist auch die Implementierung eines expliziten Verfallsdatums möglich.

- Managed Replication: Bei Managed Replication ist ein bestimmter Knoten dafür zuständig, die Anzahl vorhandener Replikas zu steuern und Änderungen an die Halter der Replikas zu propagieren. Die Information bleibt so lange verfügbar, bis der Verwalter das Netzwerk verlässt.

- Durable Replication: Bei Verwendung des Durable Modus soll die Information dauerhaft erhalten bleiben und nicht an einen bestimmten Knoten gebunden sein. Die Verwaltung der Information muss also dezentral geschehen, wofür BubbleStorm versucht, alle Halter einer Replika miteinander zu verbinden, sodass Änderungen über das spezialisierte Overlay per Flooding propagiert werden können. Dabei wird versucht, Konflikte aufzulösen, die bei mehreren gleichzeitigen Updates auftreten können.

6.1.2 CUSP

Das Channel-based Unidirectional Stream Protocol [CUS10], kurz CUSP, ist ein neues, verbindungsorientiertes Transportprotokoll, das die Vorteile von TCP und UDP mit Ideen aus neueren Transprotokollen wie SCTP und SST verbindet. Es wurde insbesondere mit den Bedürfnissen moderner P2P-Applikationen im Hinterkopf entworfen, ist jedoch für alle Zwecke verwendbar. Eine Verbindung zwischen zwei Endpunkten wird bei CUSP als Channel bezeichnet. Mehrere Streams werden per Multiplexing in einem Channel zusammengefasst. Streams können ohne

einen Roundtrip erzeugt werden und sind unidirektional. Channels sind nach einem Roundtrip aufgebaut.

Um in Applikationen einfach nutzbar zu sein, wird UDP als Unterbau von CUSP verwendet. Da UDP aber ein unzuverlässiges Transportprotokoll ist, das weder Verbindungen, exactly-once Delivery, Congestion Control, noch Authentizität kennt, werden diese Funktionen von CUSP implementiert. Congestion Control wird auf Channelebene behandelt, sodass neue Streams in bestehenden Channels nicht die Slow Start Phase abwarten müssen. Flow Control passiert auf der Streamebene.

Das Protokoll besitzt mobile Netzwerkunterstützung. Das bedeutet, dass der Benutzer seinen Netzwerkzugang bei einer bestehenden Verbindung wechseln (z.B. Wechsel von UMTS auf WLAN) oder die Netzwerkverbindung kurzzeitig unterbrochen sein kann, ohne dass der Stream abbricht. Eine Verschlüsselung des Nachrichtenstroms auf Channel-Ebene ist ebenfalls möglich. Die Verschlüsselungsparameter werden beim Aufbau des Channels ausgetauscht. Jedes Paket wird zudem per Message Authentication Code (MAC) vor Manipulationen geschützt und dem Channel zugeordnet.

Channels werden initial mittels einer UDP-Adresse hergestellt. Die in diesem Channel enthaltenen Streams sind aber nicht mit der UDP-Adresse des anderen Endpunkts assoziiert, sondern mit dessen Application Key. Dieser Sachverhalt erlaubt es, dass sich die Adresse des entfernten Endpunkts ändern kann, woraufhin automatisch ein neuer Channel zu der geänderten UDP-Adresse aufgebaut wird und alle Streams fortgesetzt werden können.

Streams können zu geringen Kosten erzeugt werden, da sie keinen Roundtrip benötigen. Daher können sie großzügig verwendet werden. Die Unidirektionalität von Streams ist in vielen P2P-Anwendungen von Vorteil, da oft Kommunikation stattfindet, bei der der Sender entweder gar keine Antwort erwartet oder eine Antwort auch erst deutlich später erfolgen kann. Dies spart Ressourcen auf Senderseite. Streams können zudem priorisiert werden, sodass zeitkritische Übertragungen zuerst erfolgen.

CUSP wird von BubbleStorm als Transportprotokoll für die Kommunikation zwischen Peers verwendet. Außerdem ist es das bevorzugte Protokoll für Implementierungen von ByteStorm. Das Protokoll lässt sich jedoch auch via TCP implementieren, jedoch unter Verlust der Vorteile von CUSP (Verschlüsselung, Multi-Streaming, Roaming, MACs) und eventuellen Performanzeinbußen.

6.1.3 Content-Suche via BubbleStorm

In [VSB10] wird ein System vorgestellt, mit dem es möglich ist, Torrents innerhalb eines BitTorrent-Clients über ein dezentrales Netzwerk zu publizieren und zu finden. Publikation und Suche finden über ein BubbleStorm-Netzwerk statt. Die Suchdaten sind somit redundant und dezentral über das Netzwerk verteilt. Jeder Peer speichert für die Suche relevante Informationen für eine Untermenge aller publizierten Torrents. Die Suchinformationen werden vom Peer für die Volltextsuche indiziert, sodass auch Fuzzy Queries (z.B. für die Suche nach Namesfragmenten und nicht genau bekannten Schreibweisen) möglich sind. Dafür wird die freie Bibliothek für Volltextsuche Apache Lucene[1] eingesetzt. Um die von jedem Peer vorzuhaltende Datenmenge so gering wie möglich zu halten, werden ausschließlich Metadaten zum Torrent gespeichert. Dazu gehören beispielsweise der Infohash des Torrents, sein Name, die Kategorie, Tags, die Größe und die enthaltenen Dateinamen des Torrents. Die Torrent-Datei selbst wird nicht indiziert.

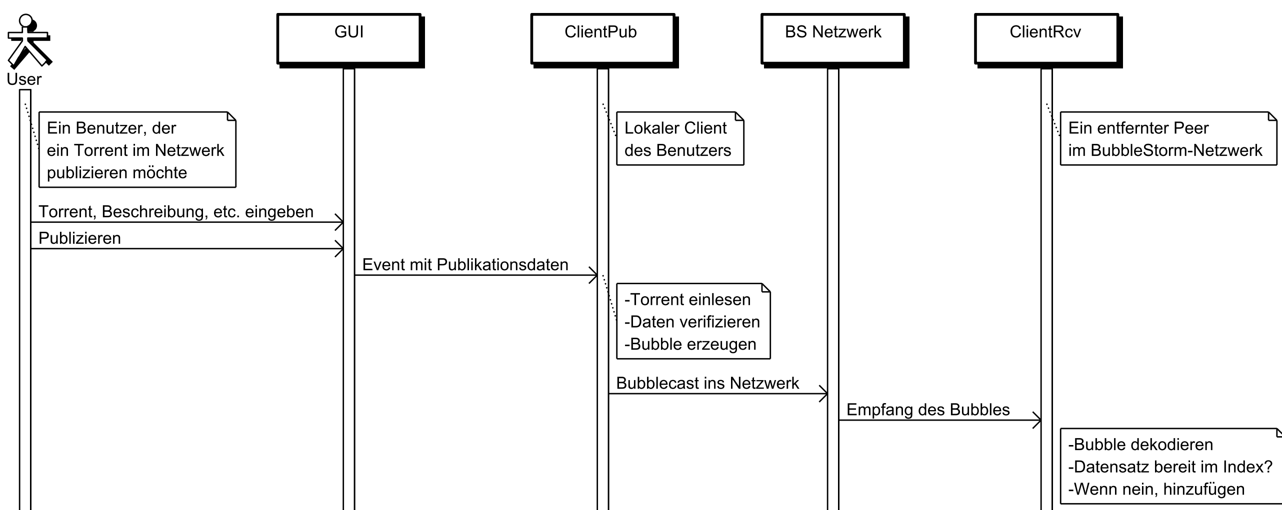

Abbildung 6.2: Veröffentlichung von Torrents im BubbleStorm-Netzwerk [VSB10, Seite 22]

Abbildung 6.2 fasst den Ablauf des Publizierens eines Torrents zusammen. Der Benutzer lädt eine Torrent-Datei aus der relevante Informationen extrahiert werden, gibt zusätzliche Daten wie Beschreibung und Kategorie ein und beauftragt dann seinen Client zur Publikation des Torrents. Dieser erzeugt einen Bubblecast, der über das BubbleStorm-Netzwerk verteilt wird. Jeder Knoten, der das Bubble empfängt, dekodiert es und prüft, ob das Torrent bereits im lokalen Index vorhanden ist. Wenn nicht, wird es diesem hinzugefügt und kann ab sofort von entsprechenden Suchanfragen gefunden werden.

[1] http://lucene.apache.org/

Apache Lucene besitzt eine eigene Query Sprache, mit der komplexe Suchanfragen gestellt werden können, was Inklusionen, Exklusionen, Wertebereiche, Teilstrings, längere Phrasen sowie Schreibfehler (Fuzzy Query) einschließt. Suchanfragen des Benutzers über die Eingabefelder der Suchansicht des BitTorrent-Clients werden in diese Query Sprache übersetzt. Die Suchanfrage wird vom Client via Bubblestorm an andere Peers gesendet, die ihren lokalen Index durchsuchen und Treffer an den Anfragenden zurückliefern. Wenn sich der Benutzer dafür entscheidet, das zu einem Treffer gehörende Torrent herunterzuladen, benötigt der Client noch die entsprechende Torrent-Datei, da diese ja nicht im Suchindex gespeichert ist. Wie bei der Verwendung von Magnet Links heute üblich, wird die Torrent-Datei direkt von anderen Peers heruntergeladen, die am selben Torrent interessiert sind, da diese die Datei ohnehin besitzen müssen. Somit wird die Torrent-Datei nur übertragen, wenn sie auch gebraucht wird. Mit der Torrent-Datei kann der Download gestartet werden. Abbildung 6.3 fasst diesen Ablauf anschaulich zusammen.

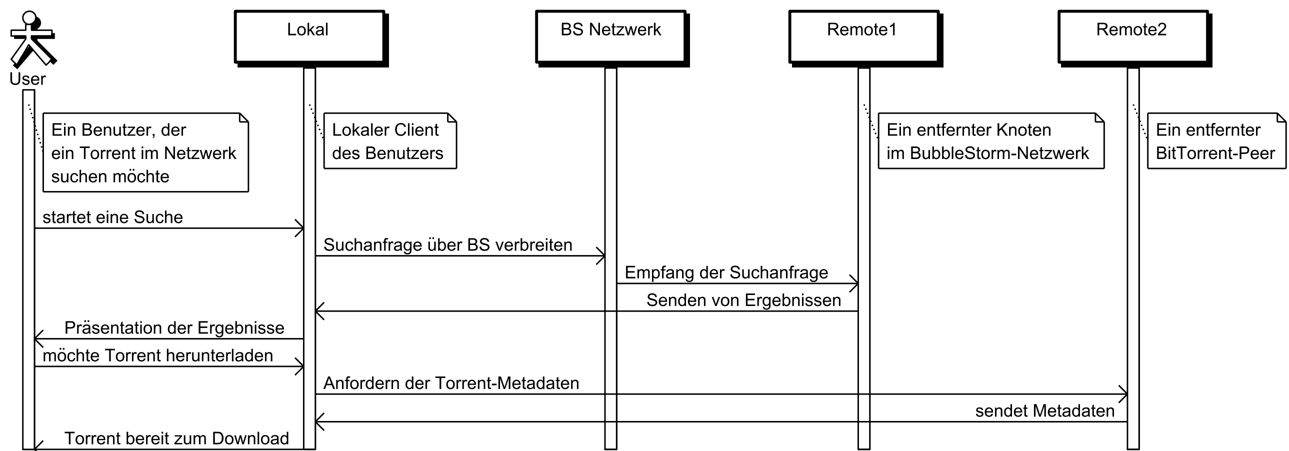

Abbildung 6.3: Suche nach Torrents im BubbleStorm-Netzwerk [VSB10, Seite 23]

6.1.4 BitTorrent-Tracking via BubbleStorm

Ebenfalls in [VSB10] wird ein Verfahren vorgestellt, mit dem via BubbleStorm weitere Peers zu einem Torrent gefunden und die eigenen Kontaktdaten bekannt gemacht werden können, um von anderen Teilnehmern gefunden zu werden. Dieser Vorgang wird als Tracking bezeichnet. Wie bei herkömmlichen BitTorrent-Trackern, die über HTTP erreicht werden, bedeutet das Senden einer Anfrage nach weiteren Peers gleichzeitig die eigene Registrierung als aktiver Peer im Schwarm des jeweiligen Torrents. Außerdem lernt jeder Peer mit jeder erhaltenen neuen Tracking-Anfrage oder Antwort auf eine Anfrage auch potenzielle BubbleStorm-Kontakte kennen. Tracking-Anfragen werden als Bubblecast gesendet und enthalten die folgenden Daten:

- Den Infohash des Torrents
- Die eigene, extern erreichbare IP-Adresse
- Den TCP-Port für eingehende BitTorrent-Verbindungen
- Den UDP-Port für eingehende BubbleStorm-Datagramme
- Ein aktueller Zeitstempel

Anfragen werden in regelmäßigen Intervallen gesendet. Erhält ein anderer Knoten die Anfrage, so speichert er die im Bubble enthaltenen Peer-Informationen im lokalen Cache unter dem Infohash als Schlüssel, um sie bei weiteren Anfragen an andere Peers weiterzugeben. Anhand des Zeitstempels wird bestimmt, wie lange der entsprechende Peer nach seiner letzten Anfrage im lokalen Peer-Cache verbleibt. Nach 30 Minuten ab Zeitstempel wird der Peer als inaktiv angenommen und aus dem Cache gelöscht, womit er nicht mehr Teil der Ergebnismenge von zukünftig eingehenden Anfragen ist. Das Ergebnis einer Anfrage besteht aus einer Liste der oben genannten Peer-Informationen für jeden im lokalen Cache enthaltenen Peer zum gegebenen Infohash.

6.2 Tempest

In diesem Unterkapitel wird das Konzept Tempest vorgestellt. In der Natur bezeichnet Tempest einen Sturm mit besonders starken Winden. In ByteStorm ist ein Tempest eine Einheit von Content. Zu einem Tempest gehört eine Menge von Dateien und Verzeichnissen, die zur gemeinsamen Übertragung vorgesehen ist. Man könnte Tempest also als Äquivalent zum Torrent-Konzept betrachten. Im Gegensatz zum Torrent, schließt der Begriff Tempest jedoch nicht die Metadaten ein. Die Menge der Metadaten wird als Squall bezeichnet, eine plötzlich auftretende starke Windbö als Vorläufer eines Sturms.

In den folgenden Abschnitten wird das Tempest-Konzept konkretisiert. Abschnitt 6.2.1 beschreibt die Metadaten zu einem Tempest und wie sie organisiert sind. Abschnitt 6.2.2 argumentiert, warum das Konzept zur Sicherstellung der Datenintegrität in BitTorrent nicht optimal ist und zu Defiziten in der Benutzerfreundlichkeit führt. Die Abschnitte 6.2.3 und 6.2.4 stellen Hash Trees und den Tempest Tree vor, ein auf dem Hash Tree aufbauendes Verfahren für die Datenintegritätsprüfung und die Adressierung von Datenblöcken in ByteStorm. In 6.2.5 wird beschrieben, wie Squall und Tempest miteinander verknüpft werden. Abschnitt 6.2.6 zeigt, wie mithilfe des Tempest Trees die Integrität der Tempest-Daten sichergestellt wird. Schließlich wird in 6.2.7 darauf eingegangen, wie in Erfahrung gebracht wird, welcher Peer welche Datenblöcke des Tempests hat.

6.2.1 Squall

Ein Squall enthält Metadaten zu einem Tempest. Er enthält alle Informationen, die es einem Benutzer ermöglichen, einen gewünschten Tempest zu finden, etwas über dessen Inhalt zu erfahren und andere Peers zu finden, die an demselben Tempest interessiert sind. Im Folgenden wird beschrieben, wie Squalls kodiert sind, welche Felder sie enthalten und welche Bedeutung diese haben.

Bencoding

Squalls werden für die Übertragung mittels Bencoding[2] kodiert, das auch das Format ist, in dem Torrent-Dateien kodiert sind. In Bencoding gibt es die Datentypen String, Integer, List und Dictionary. Strings sind binäre Zeichenfolgen und nicht unbedingt als vom Menschen lesbare Texte zu verstehen. Integer sind Ganzzahlen, die mittels ASCII-Zeichen im Dezimalsystem dargestellt werden. Sie besitzen daher keine feste Länge. Listen können beliebig viele Einträge jedes der vier Datentypen enthalten, auch weitere Listen. Dictionaries bestehen aus Schlüssel-Wert-Paaren. Der Schlüssel ist ein Benconding String, der Wert kann von beliebigem Datentyp sein. Die einzelnen Datentypen werden wie folgt kodiert:

- **String:** Strings werden gemäß folgendem Schema kodiert:

  `<Stringlänge als Dezimalzahl in ASCII>:<Stringdaten>`

  Beispiel: *`9:ByteStorm`* für den String „ByteStorm“.

- **Integer:** Ganzzahlen wird ein `i` vorangestellt und werden durch ein `e` terminiert. Die Zahl wird im Dezimalsystem im ASCII-Format kodiert. Integer werden also gemäß dem folgenden Schema kodiert:

  `i<Dezimalzahl in ASCII>e`

  Beispiel: *`i42e`* steht für die Ganzzahl 42. Integer sollten keine führenden Nullen enthalten. Der einzige Integer, der mit einer Null starten darf, ist `i0e`. Da es keine Längenbegrenzung für Integer gibt, muss beim Dekodieren darauf geachtet werden, dass der verwendete Datentyp ausreichend große Werte erlaubt. Bei einer Implementierung muss mindestens ein vorzeichenbehafteter 64 Bit Datentyp verwendet werden, damit Dateigrößen über 4 GiB möglich sind.

[2] `http://wiki.theory.org/index.php?title=BitTorrentSpecification&oldid=12399#Bencoding`

- **List:** Listen können beliebig viele Einträge beliebigen Typs enthalten. Listen wird ein `l` vorangestellt und werden durch ein `e` abgeschlossen. Sie haben also das Format:

  l<Listeneinträge>e

  Beispiel: *l9:ByteStorm8:bencodede* für eine Liste mit den Strings „ByteStorm“ und „bencoded“.

- **Dictionary:** Dictionaries wird ein `d` vorangestellt und werden durch ein `e` abgeschlossen. Ein Dictionary-Eintrag besteht aus Schlüssel-Wert-Paar, wobei der Schlüssel ein String sein muss und der Wert von jedem Datentyp sein kann. Ein Dictionary kann beliebig viele Einträge enthalten. Das Format eines Dictionaries lautet also:

  d<kodierter String><kodiertes Element>e

  Beispiel: *d9:ByteStorml8:bencoded10:dictionaryee* für ein Dictionary mit einem Schlüssel-Wert-Paar, wobei der Schlüssel „ByteStorm“ lautet und der Wert eine List mit den Strings „bencoded“ und „dictionary“ ist.

Als Alternative zum Bencoding ist auch der Einsatz von Google Protocol Buffers[3] denkbar, die Äquivalente zu den oben genannten Datentypen besitzen. Es handelt sich dabei um ein Binärformat, für das automatisch Parser und Encoder auf Basis einer Spezifikation generiert werden können. Mittels Google Protocol Buffers kodierte Nachrichten wären kürzer als per Benconding kodierte. Da die Vorarbeiten jedoch auf dem Einsatz von Bencoding basieren, wird das Format der Einfachheit halber beibehalten.

Squall Metadaten

Ein Squall hat als Wurzelelement ein Dictionary im Sinne von Bencoding. Dieses Dictionary enthält die folgenden Schlüssel-Wert-Paare.

- `uuid`: Eine zufällig generierte UUID als Binärstring (16 Bytes)
- `name`: Name des Tempests als UTF-8 String (max. 500 Bytes)
- `created`: Zeit der Veröffentlichung in Sekunden seit der UNIX-Epoche als Integer
- `expires`: Zeitpunkt in Sekunden seit der UNIX-Epoche als Integer ab dem der Squall nicht mehr in Suchergebnissen auftauchen darf (optional, muss größer als created sein)
- `desc`: Beschreibung des Tempests als UTF-8 String, kann BBCode[4] enthalten
- `catids`: Liste mit Nummern vordefinierter Kategorien als Integer (optional)
- `tags`: Liste mit Tags als String (optional)

[3] http://code.google.com/p/protobuf/
[4] http://de.wikipedia.org/wiki/BBCode

- `isfeed`: Ein optionaler Integer. Ist er vorhanden und der Wert gleich 1, zeigt er an, dass der Squall einen abonnierbaren Feed beschreibt und keine Inhalte. Das Feld `files` ist dann nicht vorhanden. (optional)
- `feedids`: Liste mit UUIDs von Abonnement-Feeds, zu denen dieser Squall gehört (optional)
- `files`: Liste von Dateien des Tempest. Für jede Datei existiert ein Dictionary mit folgenden Einträgen:
  - `path`: Dateiname inklusive relativem Pfad als UTF8-String. Pfadtrennzeichen ist der einfache Schrägstrich (/). Es wird nicht zwischen Groß-/Kleinschreibung unterschieden. „Name“, „name“ und „nAmE“ bezeichnen also dieselbe Datei.
  - `size` Dateigröße als Integer
- `defaultHeight`: Integer mit einem vorgeschlagenen Wert für die Höhe des von jedem Client zu speichernden Hashbaums (Tempest Tree).
- `runoncomplete`: Ein UTF-8 String mit Kommandos, die nach Fertigstellung des Downloads ausgeführt werden sollen. Bis auf CR und LF sind keine Steuerzeichen erlaubt. Arbeitsverzeichnis ist dasjenige, in das die unter dem Schlüssel `files` spezifizierten Dateien heruntergeladen wurden. (optional)
- `pk`: Öffentlicher Schlüssel des zum Signieren der Squall-Tempest-Bindung verwendeten Schlüsselpaares als Binärstring.

Ein Tempest wird über eine Tempest ID (kurz: TPID) eindeutig identifiziert. Diese wird berechnet, indem mithilfe des Tiger Hashalgorithmus ein 192 Bit Hashwert über die Konkatenation der Werte der Felder `uuid` und `pk` gebildet wird. Für jeden Tempest ist ein Public-Private-Schlüsselpaar erforderlich, um ECDSA-Signaturen erzeugen bzw. verifizieren zu können. Der öffentliche Schlüssel ist im Squall im Feld `pk` enthalten. Es kann für jeden Tempest ein eigenes Schlüsselpaar verwendet werden, wenn die Anonymität des Erzeugers eines Tempests gewahrt werden soll. Es kann aber auch dasselbe Schlüsselpaar für mehrere Tempests benutzt werden. In diesem Fall kann verifiziert werden, dass mehrere Tempests von derselben Quelle stammen. Dies kann z. B. dafür benutzt werden, um regelmäßig Dateien unter Realnamen bzw. einem Pseudonym zu verteilen oder um signierte Software-Updates auszuliefern.

Das Schlüsselpaar wird während der Erstellung eines Tempests für die Erzeugung von zwei ECDSA-Signaturen benötigt. Es kommen dabei SHA-256 als Hashverfahren und prime256v1 (auch: P-256) als Kurve zum Einsatz. Für die Übertragung öffentlicher Schlüssel wird das Format ASN.1 nach X.509 Spezifikation verwendet. Der komplette Squall wird per ECDSA, mithilfe des privaten Schlüssels, digital signiert und die Signatur an das Dictionary angehängt. Diese Signatur schützt vor Manipulation der Metadaten durch andere Peers im BubbleStorm-Netzwerk, die Squalls zwischenspeichern. Wird ein beliebiges Feld geändert, so ist die Squall-Signatur ungültig. Ein Angreifer könnte jedoch ein eigenes Schlüsselpaar erzeugen, im Feld `pk` den

öffentlichen Schlüssel eintragen und eine neue gültige Signatur erzeugen. Damit ändert sich jedoch auch die TPID. Daher ist die Manipulation eines Squalls äquivalent zur Erzeugung eines Squalls zu einem neuen Tempest mit identischen Metadaten, wie die Metadaten zu einem bestehenden Tempest.

Mit der zweiten Signatur wird ein Tempest an einen Squall gebunden. Dazu wird die TPID zusammen mit einem von den Tempest-Daten abhängenden Hash signiert, wobei für die Signatur derselbe private Schlüssel verwendet werden muss, wie für die Signatur des Squalls. Auf die Squall-Tempest-Bindung und die Erzeugung des besagten Hashes wird in Abschnitt 6.2.5 genauer eingegangen.

Der Squall enthält dank der Signaturen keinerlei direkten Bezug zu den Dateidaten. Dadurch bietet ByteStorm einen Vorteil in der Benutzbarkeit gegenüber z. B. BitTorrent, denn der Squall kann bereits erzeugt und veröffentlicht werden, bevor der initiale Hashvorgang auf den Dateien beim Erzeuger abgeschlossen ist. Andere Peers können also früher mit dem Download beginnen und der Erzeuger muss nicht auf den Abschluss des Hashvorgangs warten, um eine Metadatei auf eine Internetseite hochzuladen oder anderweitig zu verbreiten.

Das Feld `runoncomplete` kann Kommandos enthalten, die nach Abschluss des Downloads ausgeführt werden können. Damit kann ByteStorm z. B. für automatisiertes Deployment von Software-Updates verwendet werden. Standardmaßig sollte `runoncomplete` aus Sicherheitsgründen ignoriert werden. Es empfiehlt sich, die Kommandos zudem nur dann auszuführen, wenn alle Signaturen korrekt sind und mit einem als vertraut eingestuften privaten Schlüssel signiert worden sind.

6.2.2 Blöcke & Prüfsummen

In BitTorrent wird ein Torrent in Blöcke unterteilt. Diese Blöcke werden auch als Pieces bezeichnet. Ihre Größe entspricht dem Vielfachen der Größe eines Chunks, also 16 KiB, der kleinsten Übertragungseinheit in BitTorrent. Jedes Piece, mit Ausnahme des letzten, hat innerhalb eines Torrents dieselbe Größe. Die Piece-Größe wird von demjenigen Benutzer festgelegt, der die Torrent-Datei erzeugt. Sie kann auch mehrere MiB betragen. Zu jedem Piece gehört ein Hashwert, der der Sicherstellung der Integrität des Pieces dient. Jedesmal, wenn der BitTorrent-Client ein Piece vollständig heruntergeladen hat, wird es anhand des zum Piece gehörendes Hashes verifiziert. Im Fehlerfall wird das Piece erneut heruntergeladen. Für jedes Piece gilt, dass es erst dann anderen Peers zur Verfügung gestellt werden darf, wenn seine Integrität überprüft wurde.

Das Konzept der Pieces führt zu einer ganzen Reihe von Nachteilen:

- Da die Größe der Pieces vom Ersteller eines Torrents festgelegt wird, hat keiner der anderen Teilnehmer einen Einfluss darauf. Ist sie ungünstig gewählt, kann es zu reduzierter Performanz im Schwarm kommen.
- Jeder Client, ob mit hoher oder geringer Upload-Bandbreite, muss die gleiche, durch die Piece-Größe bestimmte, Menge an Overhead in Form von Bitmasken und HAVE-Messages an andere Peers senden, was bei langsamen Peers zur Verlängerung der Downloadzeit führen kann, da sie länger mit dem Senden des Overheads beschäftigt sind.
- Der Speicher, der für den aktuellen Zustand der Pieces eines im Download befindlichen Torrents benötigt wird, hängt von der festgelegten Piece-Größe ab und nicht von den verfügbaren Ressourcen des Rechners, auf dem der BitTorrent-Client ausgeführt wird.
- Benutzer, die dem Schwarm gerade beigetreten sind, haben in der Regel noch keine Pieces, die sie anderen Peers anbieten können. Sie können also noch nicht am Tit-for-tat teilnehmen und sind auf das Wohlwollen anderer Peers angewiesen (in Form von Optimistic Unchokes). Je größer die Pieces sind, desto länger dauert es, bis das erste Piece komplett heruntergeladen wurde und somit angeboten werden kann. Je nach Bandbreite der Schwarm-Teilnehmer kann dies einige Zeit dauern. Bis dahin bleibt die Upload-Bandbreite ungenutzt.
- Bei mehreren Gigabytes großen Torrents dauert es in der Regel mehrere Minuten, bis der Ersteller des Torrents mit dem initialen Seeden beginnen kann. Dies liegt daran, dass für das Erzeugen der Torrent-Datei die kompletten Dateidaten des Torrents einmal eingelesen werden müssen, um die Piece Hashes zu generieren, die Teil der Torrent-Datei sind. Andere Clients benötigen diese Torrent-Datei, um mit dem Download beginnen zu können. Da der Infohash, der ein Torrent eindeutig identifiziert, ein Hash ist, der u.A. über die Piece Hashes gebildet wird, ist der Torrent-Ersteller gar nicht in der Lage, sich beim Tracker zu registrieren, bevor alle Piece Hashes generiert wurden. Große Torrents bereitzustellen bedeutet also einen relativ langen zeitlichen Vorlauf.

ByteStorm eliminiert all diese Probleme, indem es sich von einem starren Blockkonzept loslöst. An dessen Stelle tritt ein flexibles Konzept, dass sich an den verfügbaren Ressourcen und den Präferenzen der Nutzer orientiert.

6.2.3 Hash Tree

Ein Hash Tree [DSB87], auch Merkle Tree genannt, ist eine Datenstruktur zur Sicherstellung von Datenintegrität. Entwickelt wurde diese Datenstruktur im Jahr 1979 von Ralph Merkle, um Lamport-Einmalsignaturen einfacher handhabbar zu machen. Ein Hash Tree ist ein Baum, des-

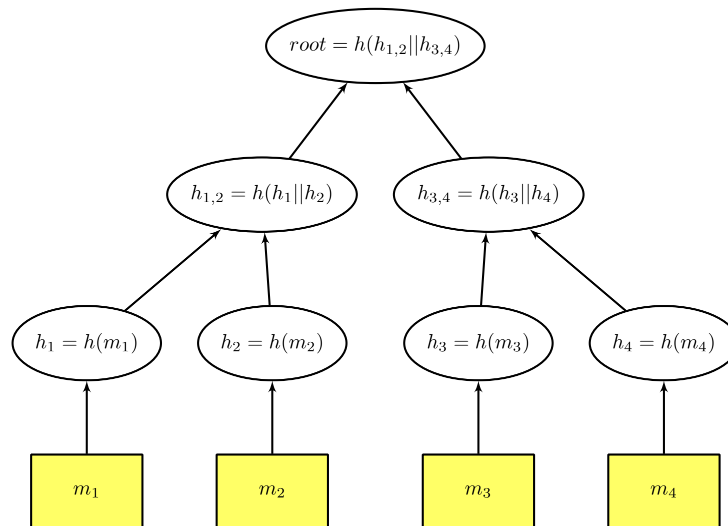

Abbildung 6.4: Beispiel eines Hash Trees

sen Blätter aus Hashes von Datenblöcken bestehen. Jeder innere Knoten entspricht dem Hash über die Konkatenation der Hashes seiner Kindknoten. Zum Generieren der Hashes wird meist eine kryptografische Hashfunktion (z. B. SHA-1) eingesetzt, aber auch andere Prüfsummenverfahren wie CRC sind möglich. Die Anzahl der Kindknoten jedes Knotens im Hash Tree ist nicht festgelegt, oft wird jedoch ein Binärbaum verwendet. Abbildung 6.4 zeigt einen binären Hash Tree über eine Nachricht m, die in vier Blöcke m_1 bis m_4 unterteilt wurde. h_1 bis h_4 entsprechen den Hashwerten dieser vier Blöcke. $h_{x,y}$ bezeichnet einen Hash, mit dem die Datenblöcke m_x bis m_y verifizierbar sind. Der Wurzelhash $root$ hat in einem Hash Tree eine besondere Stellung, denn mit seiner Kenntnis lässt sich dreierlei überprüfen:

1. Wenn die Nachricht m komplett ist, also alle vier Blöcke m_1 bis m_4 vollständig sind, dann kann die Integrität der Nachricht verifiziert werden, ohne die Werte der tieferen Baumknoten a priori kennen zu müssen, denn sie lassen sich aus der Nachricht berechnen. h_1 bis h_4 lassen sich direkt durch Anwendung der Hashfunktion h auf die jeweiligen Datenblöcke berechnen. Durch Anwenden der Hashfunktion auf die Konkatenation von h_1 und h_2 erhält man $h_{1,2}$. $h_{3,4}$ wird analog berechnet. Aus diesen beiden lässt sich wiederum die Wurzel berechnen. Bei Übereinstimmung des berechneten Wurzelwertes und dem als korrekt bekannten Wert $root$ wurde die Datei bei der Übertragung nicht verändert.
2. Kennt man den Wurzelhash durch eine sichere Quelle, kann man auch die Integrität der Hashwerte in einzelnen Baumknoten überprüfen, die beispielweise von anderen Peers übermittelt wurden. Um einen beliebigen Nicht-Wurzelknoten zu überprüfen, muss der komplette Pfad von diesem Knoten bis zur Wurzel berechnet werden. Dafür werden die Werte einiger weiterer Knoten benötigt, nämlich die Werte aller Kindknoten jedes Knotens auf dem Pfad zur Wurzel, die sich nicht mit vorhandenen Informationen berechnen lassen.

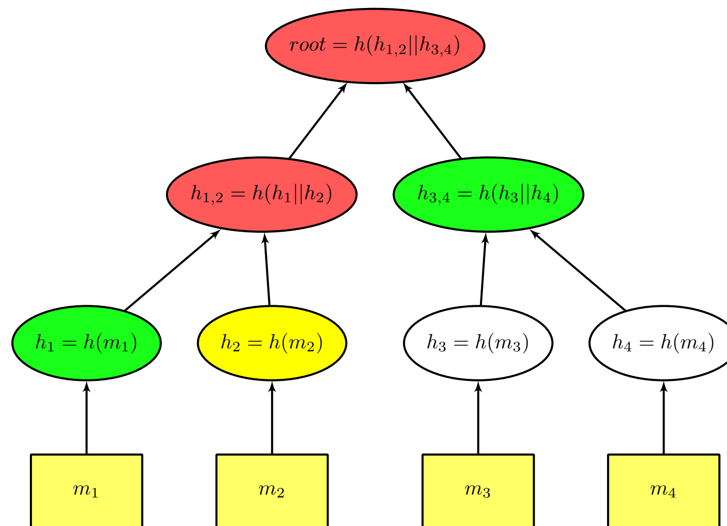

Abbildung 6.5: Verifikation von Knoten h_2 (gelb) über Wurzelpfad (rot) mithilfe von zusätzlichen Knoten (grün)

Möchte man beispielsweise die Integrität des Knotens h_2 prüfen (Abb. 6.5), so liegen der Knoten $h_{1,2}$ und die Wurzel selbst auf dem Pfad zur Wurzel. Alle Kinder dieser beiden Knoten müssen bekannt sein, um sie zu berechnen. Es werden also zusätzlich die Hashwerte h_2 und $h_{3,4}$ benötigt. Sind diese bekannt, kann zunächst $h_{1,2}$ und dann die Wurzel berechnet werden. Ist der Wurzelwert korrekt, waren alle berechneten und zur Berechnung verwendeten Werte ebenfalls korrekt.

3. Auf die gleiche Weise kann auch die Integrität einzelner Datenblöcke überprüft werden. Im letzten Beispiel lässt sich h_2 als zu verifizierenden Knoten durch den Nachrichtenblock m_2 ersetzen. h_2 lässt sich per Hashfunktion über m_2 trivial berechnen, sodass die Verifikation eines Blattknotens und eines Datenblocks äquivalent sind.

6.2.4 Tempest Tree

Der Tempest Tree ist die zentrale Datenstruktur von ByteStorm. Mit ihm wird die Integrität der Nutzdaten eines Tempest sichergestellt. Er dient aber auch als Adressierungsschema für die einzelnen Datensegmente, in die die Nutzdaten von jedem Tempest unterteilt werden. Peers tauschen den Inhalt von Knoten untereinander aus, um Teile des Downloads bereits vor Abschluss des Downloadvorgangs verifizieren zu können. Der Tempest Tree basiert auf dem Hash Tree, genauer gesagt dem Tiger Hash Tree. Tiger bezeichnet die für die Berechnung der Baumknoten verwendete kryptografische Hashfunktion [THF96]. Die Tiger Hashfunktion gehört zu den am effizientesten zu berechnenden Hashfunktionen und ist auf 64-bit Plattformen hin optimiert. [5] Die Verwendung einer schnellen Hashfunktion ist im Falle von Hash Trees besonders relevant,

[5] `http://www.cryptopp.com/benchmarks.html`

da für größere Tempests die Anzahl von Hashoperationen zur Verifizierung die Millionengrenze überschreiten kann.

	Binärbaum		Quaternärbaum	
Dateigröße	**Höhe**	**Knotenzahl**	**Höhe**	**Knotenzahl**
1 MiB	7	127	4	85
10 MiB	11	1281	6	854
100 MiB	14	12.802	8	8.535
500 MiB	16	64.001	9	42.668
1 GiB	17	131.071	9	87.381
1 TiB	27	134.217.727	14	89.478.485
1 PiB	37	137.438.953.471	19	91.625.968.981
1 EiB	47	140.737.488.355.327	24	93.824.992.236.885
1 ZiB	57	144.115.188.075.855.871	29	96.076.792.050.570.581

Tabelle 6.1: Größenvergleich zwischen binärem und quaternärem Baum

Der Tempest Tree ist kein Binärbaum, sondern ein quaternärer Baum, also ein Baum mit vier Kindknoten je innerem Knoten. Alle Zwischenknoten sind immer voll besetzt, mit Ausnahme der Knoten entlang des äußersten rechten Pfades im Baum. Die Wahl eines quaternären Baums anstatt eines Binärbaums reduziert die Anzahl notwendiger Knoten um ein Drittel und halbiert die Höhe, wodurch sich die Anzahl von Hashoperationen zur Verifizierung eines Datensegments ebenfalls halbiert und die Menge des zum Speichern des kompletten Baumes nötigen Platzes um ein Drittel schrumpft. Tabelle 6.1 zeigt einen Vergleich der Höhe und Knotenzahlen bei binären und quaternären Bäumen. Jedes Blatt des Baumes enthält den Hashwert h_x zum zugehörigen Datensegment m_x. Ein Segment ist die bei einem einzelnen Datentransfer übertragene Menge an Daten. Bis auf das letzte Segment eines Tempest haben alle Segmente eine feste Größe von 16 KiB.

Speicherung des Tempest Trees

Die Größe des Tempest Trees wächst proportional mit der Größe des Tempest. Jeder Tiger Hashwert im Baum ist 192 Bit (24 Byte) lang. Damit belegt der komplette Baum für einen Tempest ca. 0,195% der Größe des Tempest an Speicherplatz. Für kleine Tempests lässt sich der komplette Tempest Tree problemlos im Arbeitsspeicher halten. ByteStorm unterstützt jedoch Tempest-Größen von bis zu 1 Zettabyte (10^{21} Byte). Der für große Tempest Trees nötige Speicher kann somit die Menge des verfügbaren Arbeitsspeichers überschreiten. Bei heute bereits vorkommenden Dateigrößen von 50 GiB (z. B. bei Blu-ray Images), belegt der Tempest Tree ca. 100 MiB an Speicherplatz. Werden mehrere große Tempests gleichzeitig heruntergeladen oder ist nur wenig Arbeitsspeicher verfügbar, kann es also zu Problemen kommen. Eine naheliegende Lösung hierfür ist, den Baum auf ein Massenspeichermedium wie eine Festplatte oder SSD aus-

zulagern und zu jeder Zeit nur Teile des Baums im RAM zu halten. Die Flexibilität des Tempest Trees erlaubt aber auch eine weitere Lösung. Diese besteht darin, nicht alle Ebenen des Baumes zu speichern, sondern nur die oberen n Ebenen inklusive der Wurzel. Es ist auch möglich, nur den Wurzelhash zu speichern. In diesem Fall lässt sich der Download aber erst verifizieren, wenn alle Segmente vollständig heruntergeladen wurden.

Beim Handshake zwischen zwei Peers teilt jeder Peer dem anderen mit, bis zu welcher Baumebene er die Hashes speichert. Dadurch weiß der jeweils andere Peer, ob es sinnvoll ist, diesen Peer nach im lokalen Baum noch fehlenden Hashwerten zu fragen. Im zum Tempest gehörenden Squall wird eine empfohlene Baumhöhe definiert. Clients sind angehalten, die dort definierte Baumhöhe zu verwenden, um eine optimale Performanz zu erreichen, müssen dies aber nicht. Es steht jedem Client frei, gemäß seiner Ressourcen einen höheren oder niedrigeren Baum zu speichern. Genauso steht es Clients frei, Verbindungen mit Peers abzulehnen, die eine zu große oder zu geringe Baumhöhe verwenden (Höhe 1 kann beispielsweise auf Freeriding hindeuten). Die im Squall definierte empfohlene Baumhöhe muss jedoch von jeder Implementierung mindestens akzeptiert werden, sollte aber nicht die einzig akzeptierte sein. Je mehr Peers dieselbe Baumhöhe verwenden, desto eher kann ein Paar von Peers den Datenaustausch beginnen, denn es gilt immer der „kleinste gemeinsame Nenner“ für den Austausch von HAVE-Nachrichten, deren (Nicht-)Versand eng an die Baumhöhe beider Verbindungspartner gekoppelt ist und die für den Download von Segmenten unabdingbar sind (für Details siehe Abschnitt 6.2.7).

Der Tempest Tree enthält sowohl berechnete, als auch von anderen Peers erhaltene Hashwerte. Da diese Hashwerte potenziell falsch sein können, muss jeder erhaltene oder berechnete Hash bis zur Wurzel verifiziert werden. Ist das mit den aktuell vorhandenen Informationen nicht möglich, wird der Hashwert trotzdem gespeichert, aber als schwebend markiert. Als schwebend markierte Hashes dürfen nicht an andere Peers übermittelt werden. Jedes mal, wenn ein neuer Hash in den Baum eingefügt wird, wird versucht, die als schwebend markierten Hashes erneut zu verifizieren. Wird ein Hashwert als falsch erkannt, wird er aus dem Baum entfernt. Stammt der inkorrekte Hashwert von einem anderen Peer, kann eine bestehende Verbindung zu diesem getrennt und jeder weitere Verbindungsaufbau als Sanktion abgelehnt werden.

Adressierung eines Knotens

Um Informationen über Baumknoten mit anderen Peers auszutauschen, muss jeder Knoten adressiert werden können. Dazu ist ein entsprechendes Adressierungsschema notwendig. Das vom Tempest Tree verwendete Schema soll hier beschrieben werden.

Jeder Knoten befindet sich in einer Baumtiefe t und hat zusätzlich in seiner jeweiligen Tiefe einen Index i. Der Index ist ein Zähler für die in einer bestimmten Tiefe t befindlichen Knoten.

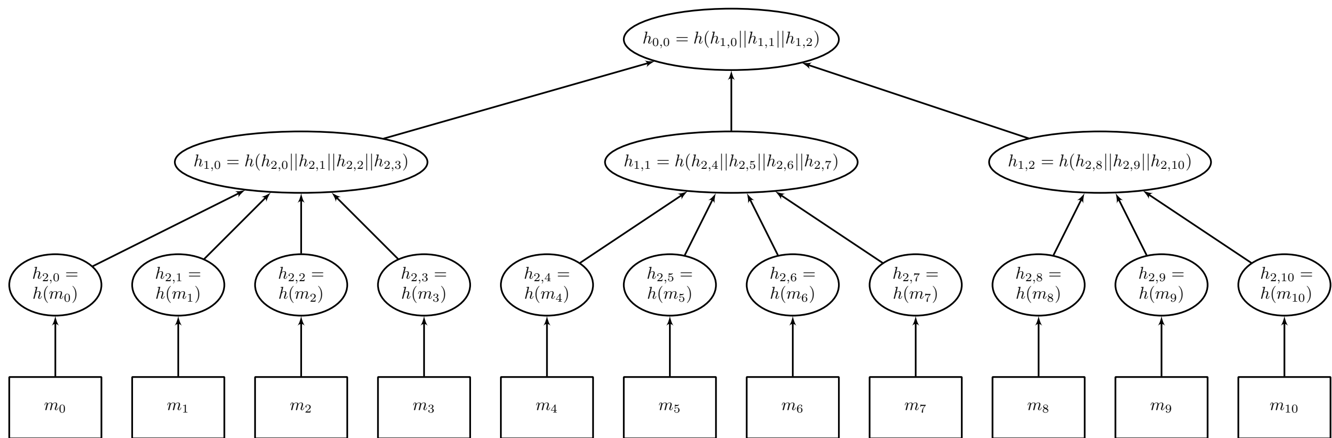

Abbildung 6.6: Beispiel eines nicht voll besetzten Tempest Trees

Der Knoten, der sich in dieser Tiefe im linkesten Pfad befindet, trägt den Index $i = 0$. Von links nach rechts werden die Knoten in dieser Tiefe durchnummeriert, sodass der Knoten im rechtesten Pfad den höchsten Index erhält. Die Tiefe wird von der Wurzel beginnend durchnummeriert. Die Wurzel hat immer die Tiefe $t = 0$ und den Index $i = 0$. Die Werte von t und i zusammen werden als Baumkoordinate oder schlicht Koordinate bezeichnet und in der Form (t, i) geschrieben. Der Hashwert an der Koordinate (t, i) heißt $h_{t,i}$. Die Baumkoordinate identifiziert je nach Kontext entweder den Hashwert des Knotens, der von ihr addressiert wird, also $h_{t,i}$, oder aber alle Datensegmente, die zu den Blättern gehören, die sich im Teilbaum unterhalb des Knotens (t, i) befinden. Für den Beispielbaum in Abb. 6.6 und einem Download-Request mit der Koordinate $(1, 0)$ bedeutet das, dass die Segmente m_0, m_1, m_2 und m_3 angefragt wurden. Ein Request nach $(0, 0)$ würde alle Segmente auf einmal anfordern.

Für die Übertragung von Baumkoordinaten wird ein 64 Bit langes Datenfeld verwendet. Das Byte mit der höchsten Ordnung enthält die Baumtiefe t. Die übrigen 3 Bytes enthalten den Index i. Die maximal adressierbare Datenmenge beträgt 1 ZiB, denn es gilt 2^{56} Segmente $\cdot$ 2^{14} Bytes/Segment $= 2^{70}$ Bytes $= 1$ ZiB. Die maximale Baumtiefe beträgt $\log_4(2^{56}) + 1 = 29$, wofür nur 6 Bits benötigt werden. Die beiden Bits höchster Ordnung bleiben somit unbelegt.

6.2.5 Squall-Tempest-Bindung

Der Wurzelhash des Tempest Trees ist nicht Teil des Squalls. Ein Squall identifiziert einen Tempest durch eine TPID eindeutig, die sich anhand des Squalls berechnen lässt (siehe Abschnitt 6.2.1). Diese hat jedoch keinerlei Bindung zu den Dateidaten bzw. zum Tempest Tree. Zu jedem Squall könnte also jede beliebige Menge von Dateien gehören. Da der Squall jedoch von dessen Erzeuger digital signiert wurde, lässt sich eine verifizierbare Zuordnung in einer

zweiten Stufe schaffen. Diese besteht darin, die Konkatenation der TPID mit dem beim Hashen berechneten Wurzelhash zu signieren. Die Signatur wird als $RootSig$ bezeichnet und muss mit demselben privaten Schlüssel erfolgen, mit dem auch der Squall signiert wurde und zu dem sich der entsprechende öffentliche Schlüssel im Squall befindet. Auf Anfrage eines Clients an einen anderen Peer im Schwarm des Tempest wird der Wurzelhash zusammen mit der Signatur übermittelt. Jeder Client im Schwarm ist verpflichtet, mindestens diese beiden Daten zu speichern und anderen Peers auf Anfrage zu senden, sobald er sie selbst kennt.

Durch die erfolgreiche Verifizierung von $RootSig$ und der Squall-Signatur sind folgende Sachverhalte garantiert, sofern der private Schlüssel nicht kompromittiert und der Signaturalgorithmus nicht gebrochen ist:

- Der Squall wurde nicht manipuliert.
- Der Wurzelhash wurde nicht manipuliert.
- Der Wurzelhash und der Squall sind vom Erzeuger miteinander verknüpft worden.
- Der Erzeuger des Squalls hat die zum Wurzelhash gehörenden Dateien mit dem Squall verknüpft.
- Falls der Public Key bereits aus einem anderen Squall bekannt ist: Beide Squalls/Tempests stammen von demselben Erzeuger.
- Falls bekannt ist, wem der private Schlüssel gehört: Squall/Tempest und die zugehörigen Dateien stammen von einer bekannten Quelle.

Die folgenden Aussagen sind jedoch **nicht** garantiert:

- Zu jedem Wurzelhash gibt es genau einen Squall. Es können mehrere Tempests mit identischen Dateien erzeugt werden, welche zum gleichen Wurzelhash führen.
- Zu jedem Squall kann es nur einen Wurzelhash mit gültiger $RootSig$ geben. Der Besitzer des privaten Schlüssels hat prinzipiell die Möglichkeit, mehrere TPID-Wurzelhash-Tupel zu signieren, zu verteilen und somit mehrere Verknüpfungen herzustellen. Dies zu tun wäre jedoch kontraproduktiv, da dann unterschiedliche Peers verschiedene Dateimengen für denselben Tempest halten, was zur Übertragung korrupter Daten und damit fehlschlagender Verifikation der Daten führen würde.
- Ein Dritter kann den Tempest nicht unter eigenem Namen veröffentlichen. Dies ist leicht möglich, indem der öffentliche Schlüssel im Squall durch einen eigenen ersetzt wird und neue Signaturen generiert werden. Der Tempest erhält dadurch aber auch eine neue TPID und entspricht somit der Erzeugung eines neuen Tempests.
- Es wurde der korrekte Wurzelhash signiert. Der Erzeuger könnte jeden beliebigen Wert signiert haben, auch eine Zufallszahl. Somit garantiert die Signatur des Wurzelhashes we-

der, dass sie zu den Dateidaten passt, noch dass dem Wert überhaupt Dateien zugrunde liegen. Dies kann nur überprüft werden, indem alle Dateien heruntergeladen werden und daraus der Wurzelhash berechnet wird.

- Die Dateien des Tempest enthalten keinen Schadcode. ByteStorm hat keinen Einfluss auf die übertragenen Inhalte von Dateien.

6.2.6 Tempest-Integrität

Initiales Hashen eines Tempest

Um ein Tempest verteilen zu können, muss, wie bei BitTorrent, dessen Content zuvor gehasht werden, wozu alle Dateien eines Tempest einmal eingelesen werden. Die Dateien werden in genau der Reihenfolge eingelesen, wie sie im Squall auftauchen. Der Hashwert jedes Datensegments wird einem Blattknoten des Baums zugeordnet. Die Zuordnung erfolgt dabei von links nach rechts in der Reihenfolge des Einlesens. Der Blattknoten, den man über den Pfad erreicht, wenn von der Wurzel ausgehend immer der linke Nachfolgerknoten gewählt wird, wird also zuerst befüllt, der Blattknoten, der bei steter Wahl des rechten Nachfolgerknotens erreicht wird, als letztes. Nach Beendigung des Hash-Vorgangs müssen alle Knoten des Baumes ihre entsprechenden Hashwerte enthalten. Ob die inneren Knoten erst berechnet werden, wenn alle Blätter gefüllt sind, oder ob sie bereits sukzessive während des Einlesens der Dateien berechnet werden, beeinflusst das Ergebnis nicht. Als eine kleine Optimierung kann letztere Variante gewählt werden, wenn die Implementierung anderen Benutzern erlaubt, das Tempest bereits herunterzuladen, während der Hashvorgang auf dem Rechner des Tempest-Erzeugers (dem initialen Seeder) noch läuft, da dadurch andere Peers ihren lokalen Tempest Tree früher mit ersten Hashwerten füllen können.

Verifizieren eines Tempest mithilfe des Tempest Trees

Um sicherzustellen, dass die im Tempest enthaltenen Dateien korrekt übertragen wurden, werden sie mittels des Tempest Trees verifiziert. Die Verifikation eines Tempests ist genau dann erfolgreich, wenn die Verifikation aller Datensegmente erfolgreich ist. Ein Leecher wird dann zum Seeder, da er nichts mehr herunterlädt, sondern nur noch verteilt. Um überhaupt irgendein Segment verifizieren zu können, wird mindestens der Wurzelhash benötigt. Wenn der Download eines Tempest gerade erst gestartet wurde, kennt der Client den Wurzelhash noch nicht, sondern nur den Squall. Aus diesem lässt sich die TPID berechnen. Anhand dieser sucht der Client Peers, die zum Schwarm des Tempests gehören und lädt sich als erstes den Wurzelhash zusammen mit seiner digitalen Signatur *RootSig* von einem der gefundenen Peers herunter. Der Tempest-Erzeuger hat die Signatur über die Konkatenation von TPID und Wurzelhash mithilfe desselben

privaten Schlüssels gebildet, mit der er auch den Squall signiert hat. Es wird daher mithilfe des öffentlichen Schlüssels, der ebenfalls im Squall enthalten ist, geprüft, ob *RootSig* eine korrekte Signatur über die Konkatenation von TPID und Wurzelhash ist. Falls ja, gehören Squall und Tempest Tree zusammen und der Download kann verifiziert werden.

Um bereits fertiggestellte Datensegmente anderen Peers zur Verfügung zu stellen, müssen diese zuvor als korrekt verifiziert werden. Wie in Unterabschnitt 6.2.3 bereits festgestellt, sind zur Verifizierung eines Segments mehrere Hashwerte entlang des Wurzelpfades im Tempest Tree notwendig. Diese können, genauso wie der Wurzelhash, von anderen Peers heruntergeladen werden. Nicht jeder Peer kann gefragt werden, sondern nur diejenigen, die das zu verifizierende Segment oder den zu verifizierenden Block von Segmenten bereits fertiggestellt haben. Für diese Peers ist garantiert, dass sie alle Hashwerte für die Berechnung des Wurzelpfades kennen, denn sie haben die betreffenden Segmente bereits selbst verifiziert. Andernfalls dürften sie nicht angeboten werden. Bevorzugt werden diejenigen Peers nach Hashwerten gefragt, die Hashwerte mindestens bis zur gleichen Baumtiefe speichern, wie der eigene Client. Dies vermeidet redundante Anfragen und damit verbundene Verzögerungen, wenn der gefragte Peer nicht in der Lage ist, Hashwerte bis zur gewünschten Tiefe zu liefern.

6.2.7 Annoncierung fertiger Tempest-Teile

Damit Segmente von anderen Peers heruntergeladen werden können, muss zunächst erstmal bekannt sein, welche Segmente diese Peers überhaupt bereits fertiggestellt haben. Dazu sendet jeder Peer bei Fertigstellung von Segmenten regelmäßig Updates an seine verbundenen Nachbarn. Eine Update-Nachricht, auch HAVE-Nachricht genannt, beinhaltet eine 64-Bit Koordinate von einem Knoten des Tempest Trees. Es gibt zwei HAVE-Nachrichtentypen mit unterschiedlicher Semantik: HAVE_DOWNLOADED und HAVE_VERIFIED.

Die Semantik von HAVE_DOWNLOADED ist, dass alle Datensegmente, die sich an den Blättern im Teilbaum unterhalb des Knotens befinden, der durch die 64-Bit Koordinate spezifiert ist, vollständig vom Sender der HAVE-Nachricht heruntergeladen wurden.

HAVE_VERIFIED signalisiert zusätzlich, dass die Integrität des kompletten Teilbaums beginnend ab der Koordinate anhand der Datensegmente und dem Wurzelhash korrekt verifiziert wurde. Der Empfänger kann davon ausgehen, dass der Sender im Besitz der korrekten Hashwerte aller Baumknoten des Teilbaums ist, sofern der Sender den Baum tief genug speichert (Parameter *MaxTreeDepth* aus HANDSHAKE-Nachricht).

Für den Baum in Abbildung 6.6 würde eine HAVE_DOWNLOADED-Nachricht mit der Koordinate $(1,0)$ also bedeuten, dass der Sender der Nachricht die Segmente m_0, m_1, m_2 und m_3 heruntergeladen hat. Eine HAVE_VERIFIED-Nachricht mit derselben Koordinate würde bedeuten, dass diese Segmente heruntergeladen und als korrekt verifiziert wurden sowie, dass andere Peers korrekte Hashwerte aus diesem Teilbaum vom sendenden Peer beziehen können. Auf eine HAVE_DOWNLOADED-Nachricht darf eine HAVE_VERIFIED-Nachricht mit einer Koordinate folgen, die den von der ersten Nachricht spezifizierten Teilbaum ganz oder teilweise überdeckt. Erhaltene HAVE_DOWNLOADED-Nachrichten für vom entfernten Peer bereits als verifiziert bekannte Teilbäume werden jedoch ignoriert.

Der Zweck für die Existenz zweier HAVE-Nachrichtentypen ist, dass Client-Implementierungen möglich sind (z.B. auf Mobilgeräten), die einen minimalen lokalen Speicherbedarf für die Vorhaltung des aktuellen Baumzustandes haben, indem sie ausschließlich den Wurzelhash speichern. Solche Implementierungen können den Download erst verifizieren, wenn alle Segmente vollständig heruntergeladen wurden, da sie nur den Wurzelhash kennen. Gäbe es nur die HAVE_VERIFIED-Nachricht, dann könnten solche Clients erst nach vollständigem Download bei der Verteilung von Daten mitwirken.

Da das Auswählen von herunterzuladenden Segmenten auf Basis erhaltener HAVE-_DOWNLOADED-Nachrichten eine erhöhte Chance in sich birgt, fehlerhafte Daten zugesendet zu bekommen, steht es Implementierungen frei, diese Nachrichten zu beachten oder zu ignorieren. HAVE_VERIFIED-Nachrichten sind jedoch immer zu beachten und der bevorzugte HAVE-Nachrichtentyp. Wenn ein Baumknoten verifiziert wurde, dann muss zwingend eine HAVE_VERIFIED-Nachricht gesendet werden. Diese Information durch das Senden einer HAVE-_DOWNLOADED zu unterdrücken ist unzulässig.

HAVE-Nachrichten werden möglichst zeitnah nach dem Herunterladen respektive Verifizieren von Segmenten gesendet. Es wird immer die Koordinate des Knotens gesendet, dessen Kindknoten alle heruntergeladen/verifiziert sind und welcher sich auf der höchstmöglichen Ebene auf dem Wurzelpfad befindet, also möglichst viele fertige Segmente zusammenfasst. Die Häufigkeit, mit der HAVE-Nachrichten tatsächlich gesendet werden, hängt jedoch von zwei Parametern ab:

1. Während des Handshakes beim Verbindungsaufbau zwischen zwei Peers sendet jeder Peer dem anderen, in welcher Granularität er HAVE-Nachrichten zu erhalten wünscht. Dadurch können Peers mit sehr niedriger Downloadbandbreite vermeiden, unnötig viel Overhead-Traffic zu erhalten, da HAVE-Nachrichten sehr häufig versendet werden können. Der Handshake-Parameter *MaxInHaveDepth* gibt an, bis zu welcher Baumtiefe der Peer HAVE-Nachrichten erhalten will. Als möglicher Standardwert kann die maximale Baum-

tiefe angenommen werden, bis zu der Hashwerte vom Peer lokal gespeichert werden (vgl. Seite 66, Paragraph „Speicherung des Tempest Trees“). Der Wert von *MaxInHaveDepth* ist von jedem Peer zu respektieren. HAVE-Nachrichten für tiefere Baumebenen als gewünscht zu senden ist unzulässig.

2. Es steht jedoch jedem Peer frei, HAVE-Nachrichten nur auf höheren Ebenen als der vom entfernten Peer gewünschten zu senden, z. B. wenn die Upload-Bandbreite gering ist oder die lokal gespeicherte maximale Baumtiefe geringer ist als der *MaxInHaveDepth*-Parameter des entfernten Peers. Die maximale Tiefe bis zu der ein Peer bereit ist, HAVE-Nachrichten zu senden, gibt er im Handshake-Parameter *MaxOutHaveDepth* bekannt. Die tatsächliche maximale Tiefe für den HAVE-Nachrichtenversand ergibt sich aus dem Minimum der eigenen *MaxOutHaveDepth* und der *MaxInHaveDepth* des entfernten Peers. Tabelle 6.2 zeigt für verschiedene Beispiele, die sich an Abbildung 6.6 orientieren, ob eine HAVE-Nachricht gesendet wird und wenn ja mit welcher Koordinate.

$S_{SegCompleted}$	S_{SegIn}	$S_{MaxOutHaveDepth}$	$R_{MaxInHaveDepth}$	S_{HAVE}
$\{\}$	m_1	2	2	$(2,1)$
$\{\}$	m_1	2	1	
$\{\}$	m_1	1	2	
$\{m_0, m_3\}$	m_2	2	2	$(2,2)$
$\{m_0, m_3\}$	m_2	2	1	
$\{m_0, m_2, m_3\}$	m_1	2	1	$(1,0)$
$\{m_0, m_1, m_2, m_3, m_4, m_6, m_7, m_8, m_9\}$	m_5	1	1	$(1,1)$
$\{m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9\}$	m_{10}	2	2	$(0,0)$

Tabelle 6.2: Beispiele für das Senden von HAVE-Nachrichten. S ist der Sender von Nachrichten, R der Empfänger. $S_{SegCompleted}$ ist die Menge bereits zuvor fertiggestellter Segmente von S, S_{SegIn} ein gerade frisch erhaltenes Segment, $S_{MaxOutHaveDepth}$ die maximale Tiefe, bis zu der S HAVEs senden will, und $R_{MaxInHaveDepth}$ die von R gewünschte maximale Tiefe für erhaltene HAVE-Nachrichten. S_{HAVE} gibt die Koordinate in der HAVE-Nachricht an oder ist leer, falls keine HAVE-Nachricht gesendet wird.

Nach dem Handshake tauschen sich zwei verbundene Peers einmalig über alle bis zu diesem Zeitpunkt fertiggestellten Segmente aus, sofern sie welche besitzen. Dies kann entweder über eine HAVE-Message geschehen, die alle fertigen Segmente als minimal mögliche Menge von Koordinaten gemäß oben beschriebener Regeln abbildet, oder über eine BITMASK-Nachricht. Diese BITMASK-Nachricht enthält für jeden Knoten auf der Ebene, die dem Granularitätsparameter g entspricht, ein Bit. Eine 1 symbolisiert einen Knoten, dessen untergeordnete Datensegmente alle vollständig und verifiziert sind, eine 0 das Gegenteil. Die Sendereihenfolge der Bits entspricht der Traversierung der Knoten auf Ebene g von links nach rechts wie bei einer beschränkten Tiefensuche. Unvollständige Bytes werden aufgefüllt (Padding). Es sollte bevorzugt diejenige Nachricht gesendet werden, welche kürzer ist. Seeder, also Peers mit vollständigem Tempest,

werden in den meisten Fällen die Nachricht `HAVE (0, 0)` senden. Im Falle des letzten Beispiels aus Tabelle 6.2 nach dem Empfang der Nachricht m_{10} kann entweder die Nachricht `HAVE (0, 0)` oder `BITMASK 11111111 11100000` gesendet werden. Eine Koordinate ist 64 Bits lang, also würde hier die BITMASK-Nachricht gesendet werden, da die Bitmaske nur 16 Bits (inkl. 5 Padding Bits) lang ist.

6.3 Tempest-Suche

In ByteStorm sind keine Dateien notwendig, die zum Starten eines Downloads benötigte Metadaten enthalten, wie in BitTorrent. Anstelle Tempests umständlich über dritte Internetseiten suchen zu müssen, ist eine dezentrale Suchfunktion eine Kernkomponente von ByteStorm. Um einen Tempest herunterladen zu können, muss man zuvor erst einmal den zugehörigen Squall gefunden haben. Dafür wird das in Abschnitt 6.1.3 beschriebene Verfahren aus [VSB10] weitgehend übernommen.

Zu jedem Tempest gibt es einen Squall mit Metadaten. Dieser Squall wird nach seiner Erstellung per Bubblecast wahlweise im Durable Replication oder Managed Replication Modus im BubbleStorm-Netzwerk publiziert und von den Peers innerhalb der Bubble gespeichert. Alle Felder werden für eine schnelle Suche indiziert.

Squalls können anhand beliebiger Kombinationen von Kriterien über alle Felder des Squalls gesucht werden. Eine Suche erfolgt mittels eines Prototypen eines Squalls. Das bedeutet, dass für die Suche dieselbe Datenstruktur verwendet wird, wie für Squalls. Allerdings sind nur diejenigen Felder enthalten, nach denen gesucht wird. Soll beispielsweise ein Tempest lediglich anhand seines Namens gesucht werden, so besteht die Suche aus einem Dictionary mit nur einem Schlüssel-Wert-Paar `name`.

Für alle Integer-Felder ist eine Bereichssuche mit oberer und/oder unterer Schranke möglich. Die Schranken für den Wertebereich werden mittels zusätzlichen Dictionary-Felder angegeben, deren Schlüssel um die Zeichenkette `atleast` (Semantik: größer gleich) oder `atmost` (Semantik: kleiner gleich) ergänzt wird. Gibt es mindestens einen solchen Bereichsschlüssel in der Suche, wird ein eventuell vorhandenes Feld mit dem Basisnamen ignoriert. Anstelle also nach einem Tempest über seine exakte Größe über das Feld `size` zu suchen, kann über die Felder `sizeatleast` und `sizeatmost` ein Wertebereich definiert werden. Ist das Feld `size` in der Suche ebenfalls enthalten, wird es ignoriert.

Alle Stringfelder werden per Stichwortsuche durchsucht. Jedes durch Leerzeichen getrennte Wort in einem String-Feld wird also standardmäßig als ein separater Suchbegriff behandelt. Je größer die Übereinstimmung mit allen Suchbegriffen ist, desto besser ist die Positionierung im

Suchergebnis. Das Suchverhalten der Stringfelder kann jedoch mittels drei Operatoren modifiziert werden:

- +*Begriff*: Mittels Plusoperator werden Suchbegriffe explizit inkludiert. *Begriff* muss im entsprechenden Feld jedes Treffers enthalten sein.
- -*Begriff*: Der Minusoperator bewirkt die explizite Exklusion. *Begriff* darf also im entsprechenden Feld keines Treffers enthalten sein.
- "*Eine Phrase*": Das Umschließen mehrerer Wörter mit doppelten Anführungszeichen erfordert, dass alle Wörter in genau dieser Abfolge im entsprechenden Feld jedes Treffers enthalten sein müssen. *Phrase*, *Eine längere Phrase* und *Eine Phase* sind keine Treffer.

Die Suche wird per Bubblecast an andere BubbleStorm-Peers propagiert, welche die Suche auf ihrem lokalen Datenbestand durchführen. Für jeden Treffer wird der Squall an den Absender zurück geschickt und dem Benutzer angezeigt. Im Gegensatz zu dem in [VSB10] beschriebenen Vorgehen kann bei ByteStorm sofort mit dem Suchen von Peers zum Tempest und dessen Download begonnen werden, da alle notwendigen Informationen bereits vorliegen.

6.4 Tracking

Das Tracking, also das Finden anderer Peers, die dieselben Dateien austauschen, entspricht in Teilen dem in Abschnitt 6.1.4 bereits beschriebenen zum Auffinden von BitTorrent-Peers über das BubbleStorm-Netzwerk. Für ByteStorm wurde das Nachrichtenformat, sowie die Art des Nachrichtenaustauschs jedoch modifiziert und an die Bedürfnisse und Möglichkeiten angepasst. Diese Änderungen sollen in diesem Kapitel besprochen werden.

6.4.1 Modifikationen des Protokolls

Es gibt nach wie vor nur eine einzige Nachricht, welche gleichzeitig die Existenz eines Peers annoncieren, weitere Peers anfragen und Ergebnisse zurückliefern *kann*. Das Wort „kann“ ist hier deshalb hervorzuheben, weil in der in [VSB10] beschriebenen Variante das Senden einer Tracking-Nachricht immer zu einer Annoncierung der eigenen Adresse und dem gleichzeitigen Anfordern einer Liste von Peers äquivalent ist. Dies ist in ByteStorm nicht der Fall. Eine Tracking-Nachricht kann zum Annoncieren des eigenen Peers, zum Anfragen weiterer Peers oder beides auf einmal benutzt werden. Das Entkoppeln dieser beiden Zwecke führt dazu, dass die folgenden Einsatzzwecke möglich werden:

- In ByteStorm hat jeder Benutzer die Option, seine Identität geheim zu halten. Dazu gehört, dass er Tracking-Nachrichten versenden kann, ohne die eigenen Adressdaten darin

preiszugeben. In diesem Fall kann der Client des Benutzers nur durch eigene Initiative eine Verbindung zu anderen Peers herstellen, andere Peers können von außen aber keine Verbindung aufbauen.

- Damit festgestellt werden kann, ob ein Peer überhaupt noch verfügbar ist, wird der Managed Replication Modus von BubbleStorm verwendet. Jeder Peer muss regelmäßig eine Nachricht mit seinen Adressdaten senden, die signalisiert, dass er noch verfügbar ist. Jeder BubbleStorm-Knoten, der Adressdaten anderer Peers speichert, hält diese nur solange vor, bis er entweder selbst das Netzwerk verlässt oder 6 Stunden seit der letzten Republikation vergangen sind. Dabei hat der fragliche Peer mit hoher Wahrscheinlichkeit noch genügend aktive Peers in seinem lokalen Cache und benötigt keine weiteren. Somit kann das Antworten mit einer Peer-Liste entfallen.

ByteStorm verwendet einen Publish-Subscribe-Ansatz zum Anfordern neuer Peers. Dabei werden alle bestehenden Peers im Schwarm eines Tempest, die Subscriber sind, automatisch benachrichtigt, sobald ein neuer Peer dem Schwarm beitritt. Peers werden zu Subscribern, indem sie eine Tracking-Nachricht senden. BubbleStorm-Knoten, die eine Tracking-Nachricht erhalten, registrieren den Absender lokal als Subscriber. Jedes Mal, wenn eine Tracking-Nachricht eingeht, die Adressdaten eines neuen Peers enthält, erhalten alle bekannten Subscriber automatisch eine Tracking-Nachricht, die sie über die Adressdaten des neuen Peers informiert. Es werden eingehende Tracking-Nachrichten als Publikationen behandelt, Subscriber also informiert, wenn a) die Nachricht Adressdaten eines Peers enthält, b) die Adressdaten Verbindungen zum Peer zulassen, d.h. eine gültige IP-Adresse und mindestens eine echt positive Port-Nummer enthalten ist, und c) der Peer nicht mit denselben Daten bereits als Subscriber registriert ist (wg. wiederholt erhaltener Tracking-Nachricht). Trifft eine dieser Bedigungen nicht zu, wird also die Peer-Information nicht weitergeleitet, da sie für dritte Peers nicht von Nutzen ist. Unabhängig davon erhält der Absender beim Eintreffen einer Tracking-Nachricht eine Liste von bis zu 50 Peers, die denselben Tempest teilen. Subscriber unterliegen einem Timeout von 6 Stunden und werden danach aus der loakeln Subscriber-Liste automatisch entfernt. Um darüber hinaus über weitere neue Peers informiert zu werden, muss also alle 6 Stunden erneut eine Tracking-Nachricht gesendet werden. Dadurch ist sichergestellt, dass ausgefallene Subscriber nicht für unbegrenzte Zeit Subscriber bleiben und damit unnötigen Traffic bei anderen Peers erzeugen. Zudem gibt es ein NoSubscribe-Flag. Ist dieses aktiviert, wird der Absender nicht als Subscriber registriert, sondern erhält nur einmalig eine Liste von Peers. Das Setzen dieses Flags kann sinnvoll sein, wenn der Absender bereits ausreichend viele Peers kennt oder aufgrund einer sehr hohen Anzahl von Schwarm-Teilnehmern mit zu vielen Benachrichtigungen zu rechnen ist.

Nachrichtenformat

Hier wird nun das Format von Tracking-Nachrichten vorgestellt. Diese verwenden Bencoding und bestehen aus einem Dictionary mit den folgenden Einträgen:

- `tpid`: Die TPID als Binärstring (16 Bytes)
- `root`: Der Wurzelhash des Tempest aus dem Tempest Tree als Binärstring (24 Bytes, optional)
- `ip`: Die IP-Adresse des Peers als Text-String in Dezimalpunktschreibweise (IPv4) bzw. gemäß RFC 5952 (IPv6)
- `port`: Der abgehörte Port für eingehende Verbindungen zum Datenaustausch als Integer
- `bsport`: Der abgehörte Port für BubbleStorm-Kommunikation als Integer
- `gwport`: Der Port, auf dem der Peer als HTTP-Gateway für Tempests fungiert. Ist entweder 0 oder nicht vorhanden, wenn der Peer nicht als HTTP-Gateway bereit steht. (siehe Abschnitt 6.8)
- `timestamp`: Zeitstempel in Sekunden seit der Unix-Epoche als Integer
- `anonpk`: Binärstring mit einem öffentlichen Schlüssel, der von anderen Peers zur Verschlüsselung anonymisierter Verbindungen über diesen Peer verwendet wird. (siehe Abschnitt 6.6)
- `nosubscribe`: Integer, der bei einem Wert von 1 signalisiert, dass der Absender nicht als Subscriber registriert werden, sondern nur einmalig eine Liste von Peers erhalten möchte.

Die Felder `tpid` und `root` identifizieren beide den Schwarm eines Tempests. Das Feld `tpid` identifiziert ihn über die TPID, welche sich aus den Daten des Squalls berechnen lässt (siehe Abschnitt 6.2.1), `root` über den Wurzelhash über die Dateidaten. Mindestens die TPID muss in jeder Nachricht enthalten sein. Die TPID identifiziert einen Tempest eindeutig, jedoch nicht seine Daten. Die TPID ähnelt dem Infohash in BitTorrent, der einen Torrent identifziert, der aber nichts darüber aussagt, ob es sich bei zwei unterschiedlichen Torrents in Wirklichkeit um dieselben Nutzdaten handelt. Das Feld `root` kann deswegen zusätzlich verwendet werden, um Peers zu finden, die zwar dieselben Dateien tauschen, aber nicht denselben Tempest verwenden. Durch das Inkludieren des Wurzelhashes können die Teilschwärme mehrerer Tempests also zu einem großen Schwarm zusammengeführt werden, was in BitTorrent für unterschiedliche Torrents nicht möglich ist.

Die erste von einem Peer beim Start des Downloads gesendete Tracking-Nachricht kann nie das Feld `root` enthalten, denn er kennt zu diesem Zeitpunkt nur den Squall. Den Wurzelhash kann der neue Peer erst dann erfahren, wenn er sich erstmals mit einem anderen Peer des

Schwarms verbindet. Bis zu diesem Zeitpunkt ist daher kein schwarm-übergreifender Dateiaustausch möglich.

Die Felder `ip`, `port` und `bsport` dürfen in Anfragen weggelassen werden, wodurch ein Peer nicht von außen kontaktiert werden kann und die Tracking-Nachricht nicht als Publikation behandelt wird. Ist jedoch eines der Felder vorhanden, müssen alle drei Felder vorhanden sein und gültige Werte enthalten. Antworten/Benachrichtigungen enthalten alle drei Felder immer. Sie enthalten genau dieselben Felder und Werte, die in der letzten erhaltenen Tracking-Nachricht vom jeweiligen Peer enthalten waren.

6.5 Datenübertragung

Die Übertragung der Nutzdaten von einem Tempest erfolgt in 16 KiB großen Segmenten. Peers unterrichten ihre Nachbarn regelmäßig über die Teile des Tempest Trees, die sie bereits fertiggestellt haben. Anhand dieser Information werden Datensegmente ausgewählt, die vom jeweiligen Peer heruntergeladen werden sollen. Dadurch ist sichergestellt, dass nur Datensegmente versucht werden herunterzuladen, die der fragliche Peer auch tatsächlich hat bzw. vorgibt zu haben. Um Segmente herunterzuladen, werden sie mittels Request-Nachrichten beim entfernten Peer angefragt. Dieser platziert sie in einer Warteschlage für zu bedienende Requests oder er lehnt die Anfrage ab und teilt dies dem Sender mit. Mit jeder Request-Nachricht können entweder einzelne Segmente oder ganze Teilbäume aus dem Tempest Tree angefragt werden, wodurch sich der Nachrichten-Overhead stark reduzieren lässt, da Request-Nachrichten zu den am häufigsten versendeten Nachrichten in ByteStorm zählen.

6.5.1 Segmentauswahl-Strategie

Ein wichtiges Kriterium für die Performanz von ByteStorm ist die Reihenfolge, in der Segmente heruntergeladen werden. Würde der Tempest einfach linear heruntergeladen werden, dann kann das zu einer nicht optimal genutzten Upload-Bandbreite bei vielen Peers führen, da die meisten Peers ungefähr dieselben Segmente bereits heruntergeladen haben und somit keine Segmente besitzen, an dem andere Peers Interesse haben. Außerdem ist die Wahrscheinlichkeit groß, dass der Tempest nicht fertiggestellt werden kann. Das letzte Segment wäre immer nur dann verfügbar, wenn es noch mindestens einen Seeder gibt, der alle Segmente hat. Ist das nicht der Fall, können alle verbleibenden Peers den Download nicht abschließen.

Der Download der Segmente in zufälliger Reihenfolge löst das erste Problem. Die Verfügbarkeit aller Segmente wird im Mittel zwar verbessert, ist aber dennoch nicht optimal, da sie

vom Zufall abhängt. Diese Strategie ist daher als Minimalstrategie anzusehen, die nur dann verwendet werden sollte, wenn Speicher und Rechenleistung knapp sind.

Die empfohlene Strategie für ByteStorm ist eine modifizierte Variante von der „Local Rarest First"-Strategie, die auch in BitTorrent zum Einsatz kommt. Local Rarest First ist ein Kompromiss zwischen optimaler Verfügbarkeit und der Menge ausgetauschter Nachrichten. Bei Local Rarest First werden diejenigen Segmente zuerst angefragt, die die wenigsten der momentan direkt verbundenen Nachbarn besitzen. Es werden also gezielt Segmente heruntergeladen, die am ehesten davon bedroht sind, aus dem Schwarm zu verschwinden, um so die Verfügbarkeit des kompletten Tempests möglichst lange aufrecht zu erhalten. Diese Entscheidung wird allerdings nur anhand lokal verfügbarer Informationen getroffen. Bei größeren Schwärmen sind in der Regel nicht alle Peers untereinander direkt verbunden, sodass das seltenste Segment aus lokaler Sicht nicht notwendigerweise auch das seltenste im kompletten Schwarm ist. Für eine Optimierung von Rarest First auf den kompletten Schwarm wären erhebliche Mengen von Nachrichten nötig, um den globalen Zustand aller Segmente bei allen Peers zu bestimmen. Local Rarest First verwendet lediglich die Informationen, die ohnehin nötig sind, um sinnvoll Segmentanfragen stellen zu können. Eine mögliche Modifizierung besteht darin, dass bevorzugt Segmente aus Teilbäumen heruntergeladen werden, aus denen bereits Segmente fertiggestellt sind. HAVE-Nachrichten werden meist nicht nach dem Download einzelner Segmente gesendet, sondern nach der Fertigstellung von Teilbäumen, um den Protokoll-Overhead zu reduzieren (vgl. Abschnitt 6.2.7). Je früher ein Teilbaum fertiggestellt wird, desto eher können HAVE-Nachrichten gesendet werden, die andere Peers über Segmente informieren, die möglichweise für sie interessant sind. Die verfügbare Upload-Bandbreite kann dadurch besser ausgenutzt werden, was einen direkten Einfluss auf die Downloadgeschwindigkeit hat. Die Implementierung dieser Modifikation liegt somit im Interesse des Benutzers. Da jeder Client die Häufigkeit seiner empfangenen und gesendeten HAVE-Nachrichten selbst bestimmen kann, obliegt es der jeweiligen Implementierung, zu entscheiden, welche Teilbäume priorisiert werden. Zu deren Bestimmung könnten beispielsweise die Handshake-Parameter einbezogen werden.

Normalerweise sollte jedes Segment nur bei einem Peer gleichzeitig angefragt werden, um doppelte Übertragungen zu vermeiden. Kurz vor Fertigstellung des Downloads kommt es aber häufig vor, dass die Downloadrate stark einbricht, weil die letzten Segmente von Peers mit einer sehr niedrigen Datenrate gesendet werden. Daher dürfen Request-Nachrichten für dasselbe Segment an mehr als einen Peer gesendet werden, wenn der Download nahezu komplett ist, z. B. wenn für alle verbleibenden Segmente eine Request-Nachricht gesendet wurde.

6.5.2 Upload-Strategie

ByteStorm soll einen fairen Datenaustausch bieten, der das Bereitstellen von Upload-Bandbreite belohnt, ohne zusätzlichen Overhead zu verursachen. Je stärker ein Peer bei der Verteilung der Daten mitwirkt, desto schneller soll sein eigener Download voranschreiten. Freerider sollen dagegen möglichst wenig profitieren und nur überschüssige Übertragungskapazitäten nutzen können. Ein Umgehen der Beschränkungen durch Freerider soll weitgehend erschwert werden. Jeder Peer erhält dadurch einen Anreiz, möglichst viel Upload-Bandbreite bereitzustellen und somit dem kompletten Schwarm kürzere Downloadzeiten zu ermöglichen.

All diese Eigenschaften bietet der als FairTorrent vorgestellte Ansatz aus [FTP09]. Dieser stellt die Strategie dar, mit der auch ByteStorm alle eingehenden Requests abarbeitet. Trotz (oder gerade wegen) seiner Einfachheit macht der FairTorrent-Ansatz Clients mit abweichenden Strategien, die einen schnelleren Download bei gleichzeitig geringem Upload-Beitrag zum Ziel haben, weitgehend erfolglos, da er ausschließlich von lokal objektiv messbaren Größen abhängt.

Zur Implementierung der Strategie in ByteStorm werden zunächst alle angefragten Segmente aus erhaltenen Request-Nachrichten in einer Warteschlange platziert. Für jeden Peer wird die Differenz d der hoch- und heruntergeladenen Datenmenge gespeichert. Bei neuen Peers beträgt diese 0. Die Differenz wird Tempest-unabhängig gespeichert, sodass die Reputation berücksichtigt wird, wenn zwei Peers mehr als einen gemeinsamen Tempest haben. Die Anzahl der gleichzeitigen Datenübertragungen ist auf einen festen, sinnvollen Wert n beschränkt, der in Relation zur verfügbaren Upload-Bandbreite stehen sollte. Die Daten angefragter Segmente werden stets an diejenigen n Peers gesendet, deren Wert d am größten ist und von denen es Segmentanfragen in der Warteschlange gibt. Nach jedem gesendeten oder empfangenen Segment wird d aktualisiert. In welcher Reihenfolge die Anfragen eines Peers abgearbeitet werden ist undefiniert. Damit sind diverse lokale Optimierungen der Performanz möglich, z.B. das bevorzugte Senden von Segmenten, die sich noch im Dateicache des Betriebssystems befinden, um Festplattenzugriffe zu sparen.

Sobald ein Peer den Download abschließt und zum Seeder wird, ist obige Strategie nicht mehr sinnvoll, da der Peer selber keine Daten mehr herunterlädt. In diesem Fall werden angefragte Segmente per Round Robin verteilt. Jeder Peer erhält also der Reihe nach ein Datensegment unter Berücksichtigung des Limits von n gleichzeitigen Übertragungen.

Für Peers, die einen Tempest-Download gerade erst begonnen haben, kann es schwierig sein, an die ersten Segmente zu gelangen, um sie weiterzuverteilen und damit Reputation zu erlan-

gen. Dies ist allerdings nur dann der Fall, wenn es keine Seeder gibt, da diese Segmentanfragen aller Peers bedienen. Daher ist ein Bootstrapping-Prozess nicht teil der Spezifikation von ByteStorm. Sollte sich doch einer als notwendig herausstellen, so darf dieser sich nicht zum Beschleunigen des Downloads eignen. Der neue Peer sollte deshalb nicht in der Lage sein, selbst zu bestimmen, welche initialen Segmente er erhält. Insbesondere sollten alle neuen Peers innerhalb eines längeren Zeitraums stets dieselben Segmente fürs Bootstrapping erhalten, um zu verhindern, dass durch den Wechsel der Identität ein signifikanter Vorteil zu erlangen ist.

6.6 Anonymität

Für Anwendungsfälle, bei denen es wichtig ist, dass die Identität mindestens einer der beiden Verbindungspartner während der Übertragung eines Tempests vertraulich bleibt, bietet ByteStorm eine optional nutzbare Anonymisierungsfunktion. Ziel dieser ist eine Verschleierung des Kommunikationsinhalts zwischen zwei Endpunkten und ein ausreichendes Maß an Schutz gegen die Ermittlung des wahren Urhebers der Kommunikation. Ein Endpunkt, der seine Identität verschleiern möchte, wird im Rest dieses Kapitels als anonymer Peer bezeichnet. Alle anderen sind nicht-anonyme Peers.

Ziel der Anonymisierungsfunktion von ByteStorm ist nicht das absolute Verunmöglichen einer Deanonymisierung, sondern eine Balance zwischen dem Grad der Anonymität eines Peers, dem Datendurchsatz und der Interoperabilität zu bieten. Die beste Anonymisierung nützt nichts, wenn ein Download nicht in einem hinnehmbaren Zeitraum abgeschlossen werden kann. Da es in ByteStorm sowohl anonyme als auch nicht-anonyme Peers gibt, ist es im Sinne der Verfügbarkeit von Tempest-Daten naheliegend, dass Peers aus beiden Gruppen mit Peers aus ebenfalls beiden Gruppen kommunizieren können sollten. Dies wird in einer Weise umgesetzt, durch die jeder Peer, der als Relay-Knoten für anonyme Kommunikation fungiert, glaubhaft abstreiten kann, dass eine bestimmte Kommunikation von ihm ausging und nicht von einem anonymen Knoten, für den der Peer ein Relay-Knoten war. Zudem weiß keiner der Endpunkte einer ByteStorm-Verbindung, ob er mit einem anonymen Peer kommuniziert oder nicht, da die IP-Adresse der Gegenseite nicht notwendigerweise der Kommunikationsendpunkt ist.

Angriffsszenarien gegen die das hier beschriebene Verfahren einen Schutz bietet sind die Deanonymisierung durch einzelne Nachbarknoten, das Einblicknehmen in den Kommunikationsinhalt durch Parteien außer Sender und Empfänger, Manipulation von Kommunikationsinhalten durch Dritte, und das Beobachten von Netzwerkverkehr an einzelnen, zufälligen Punkten auf dem Kommunikationspfad. Als Kompromiss zugunsten der Performanz besteht jedoch kein ausreichender Schutz gegen Angreifer, die jeglichen Verkehr im Netzwerk (oder einem großen Teil davon) überwachen können oder mindestens den zweiten und vorletzten Knoten auf dem Pfad einer anonymisierten Verbindung kontrollieren. Es wird zudem nicht die Tatsache verschleiert,

dass ByteStorm als Protokoll genutzt wird. Dazu wäre eine Verschlüsselung auf Verbindungsebene erforderlich.

Für die Anonymisierung wird eine Variante von Onion Routing verwendet. Der Initiator baut also auf Protokollebene einen verschlüsselten Tunnel zum gewünschten Peer über einen oder mehrere Onion Router auf, welche aus anderen ByteStorm Peers bestehen. Die Tunnellänge, also die Anzahl Tunnelhops bis zum Ziel, ist variabel und kann vom Benutzer konfigurierbar sein. Vor jeder Verbindung ist eine Zufallszahl im Intervall der gewünschten Anzahl Onion Router zu ziehen. Die typische Anzahl für eine ausreichende Balance zwischen Anonymität und Performanz liegt erfahrungsgemäß zwischen 1 und 3 Zwischenstationen.

6.6.1 Tunneltypen

Eingangs wurde das Ziel formuliert, dass jede Kombination von anonymen und nicht-anonymen Peers in der Lage sein soll, miteinander zu kommunizieren. Onion Routing dient jedoch primär der Initiatoranonymität. Daher muss derjenige Peer, der eine Verbindung aufbauen möchte, IP-Adresse und Port seines Ziels kennen. Dies steht jedoch zunächst im Widerspruch dazu, dass ein anonymer Peer Ziel einer Verbindung sein kann, denn dazu müsste er seine Adresse preisgeben. Aus diesem Grund gibt es zwei unterschiedliche Typen von anonymen Verbindungen: Direkttunnel und Proxy-Tunnel.

Direkttunnel

Durch einen Direkttunnel stellt ein anonymer Peer eine verschlüsselte Ende-zu-Ende Verbindung zu einem anderen Peer her. Letzterer kann selbst anonym sein oder auch nicht. Dazu ermittelt der Initiator zunächst einmal IP und Port eines Peers für den gewünschten Tempest über das Tracking-Protokoll von ByteStorm. Dies kann er entweder selbst tun oder, was sicherer ist, einen anderen vertrauten Peer mit der Durchführung der Peer-Suche beauftragen, z.B. einen (oder mehrere) der direkten BubbleStorm-Nachbarn. Zudem werden weitere Peers benötigt, die als Onion Router für die Weiterleitung dienen. Da es für sie keine Rolle spielt, ob sie den entsprechenden Tempest besitzen oder nicht, können diese sowohl aktiv als auch passiv gefunden werden, z.B. durch via Tracking-Protokoll eingegangene Nachrichten. Es sind ausschließlich diejenigen Peers für die Einbindung in Tunnel geeignet, für die ein öffentlicher Schlüssel bekannt ist, deren Tracking-Nachricht also das Feld *anonpk* enthält.

Ist ein Direkttunnel erfolgreich aufgebaut, dann können über diesen beliebige Nachrichten des ByteStorm-Protokolls ausgetauscht und Tempests übertragen werden. Vom Initiator gesendete Handshake-Nachrichten können dabei beliebige Werte in allen Port-Feldern enthalten.

Proxy-Tunnel

Proxy-Tunnel ermöglichen anonymen Peers, eingehende Verbindungen zu erhalten, ohne dass sie Adressinformationen von sich preisgeben müssen. An der Stelle des anonymen Peers tritt ein anderer Peer im Schwarm des Tempests auf, im Folgenden Proxy genannt, zu dem der anonyme Peer einen Proxy-Tunnel aufbaut. Der Tunnelaufbau unterscheidet sich vom Vorgehen bei Direkttunneln nur darin, dass jeder Peer mit öffentlichem Schlüssel als Proxy (und damit als Tunnelziel) in Betracht kommt und nicht nur Peers mit identischem Tempest. Zudem zeigt der Tunnelhandshake an, dass es sich um einen Proxy-Tunnel handelt. Dies ist gleichzeitig die Aufforderung an den Proxy, dass er sich über das Tracking-Protokoll für den vom anonymen Peer gewünschten Tempest registrieren soll. Der Proxy nimmt eingehende Verbindungen über das herkömmliche ByteStorm-Protokoll an, welche auch durch einen anderen Direkttunnel getunnelt sein können, sodass zwei anonyme Peers miteinander kommunizieren können. Verweist die erste erhaltene HANDSHAKE-Nachricht auf den Tempest, für den sich der anonyme Peer interessiert, dann leitet der Proxy diese Nachricht und alle folgenden über den Tunnel an den anonymen Peer weiter. Umgekehrt entpackt der Proxy alle ByteStorm-Nachrichten, die vom anonymen Peer über den Tunnel ankommen, und leitet sie an den anderen Peer weiter. Falls der Proxy mehrere anonyme Peers bedient, die sich für den gleichen Tempest interessieren, werden eingehende Verbindungen nach dem Zufallsprinzip einem der Tunnel zugewiesen. In HANDSHAKE-Nachrichten, die vom anonymen Peer über den Tunnel zum Proxy gelangen, ersetzt der Proxy vor dem Weiterleiten die Port-Felder durch seine eigenen Ports. Über einen Proxy-Tunnel können mehrere Verbindungen durch Multiplexing abgewickelt werden, indem die Kombination aus IP und Sender-Port der eingehenden Verbindung mit jeder Nachricht übertragen wird.

Proxy-Tunnel ermöglichen dem Proxy plausibel abzustreiten, dass er den Datenaustausch bestimmter Inhalte verursacht hat. Von aussen betrachtet lässt sich nicht unterscheiden, ob der Proxy eine bestimmte Nachricht selbst gesendet oder nur weitergeleitet hat. Dies gilt aber nur solange keine Analyse des gesamten Traffics des Proxys stattfindet, da bei einem Proxy-Tunnel jede ankommende Nachricht eine gesendete zur Folge hat.

6.6.2 Nachrichtenformat

Sämtliche über die Anonymisierungsfunktion gestarteten Nachrichtenübertragungen werden über das ByteStorm-Protokoll mittels der TUNNEL-Nachricht abgewickelt. Der Payload besteht aus Nachrichten des ByteStorm-Protokolls gefolgt von einem Hashwert über diese. Der Payload wird nach dem Zwiebelschalenprinzip verschlüsselt. Für jeden Onion Router auf dem Tunnelpfad, einschließlich dem Ziel, gibt es einen symmetrischen Schlüssel, der beim Tunnelaufbau

vereinbart wird. Der Payload jeder nach dem Tunnelaufbau vom Initiator gesendeten TUNNEL-Nachricht wird iterativ mit dem symmetrischen Schlüssel jedes Onion Routers in umgekehrter Reihenfolge der Tunnelroute verschlüsselt und die Nachricht an den ersten Onion Router auf dem Pfad übergeben. Dieser entschlüsselt den Payload mit seinem symmetrischen Schlüssel und leitet die entschlüsselte Nachricht an den nächsten Onion Router weiter. Dies wiederholt sich, bis die Nachricht das Ziel erreicht. Dieses prüft die Korrektheit der ByteStorm-Nachrichten anhand des angehangenen Hashwerts. Bei erfolgreicher Verifikation wird die Nachricht verarbeitet, andernfalls wird die Verbindung beendet.

Der Payload von TUNNEL-Nachrichten vom Zielknoten an den Initiator wird von jedem Onion Router (einschließlich Zielknoten) einmal mit seinem symmetrischen Schlüssel verschlüsselt. Der Payload besteht ebenfalls aus mindestens einer ByteStorm-Nachricht gefolgt von einem Hashwert über diese, welche beim Erhalt vom Initiator geprüft wird.

Für die Verschlüsselung wird AES-CTR als Stromchiffre angewendet und Tiger als Hashfunktion eingesetzt. Handelt es sich um einen Proxy-Tunnel, dann beginnt der Payload jeder TUNNEL-Nachricht nach dem Tunnelaufbau in beiden Richtungen zusätzlich mit der IP-Adresse und dem Port des Peers, der mit dem Proxy verbunden ist, um Verbindungen durch den Tunnel zu multiplexen.

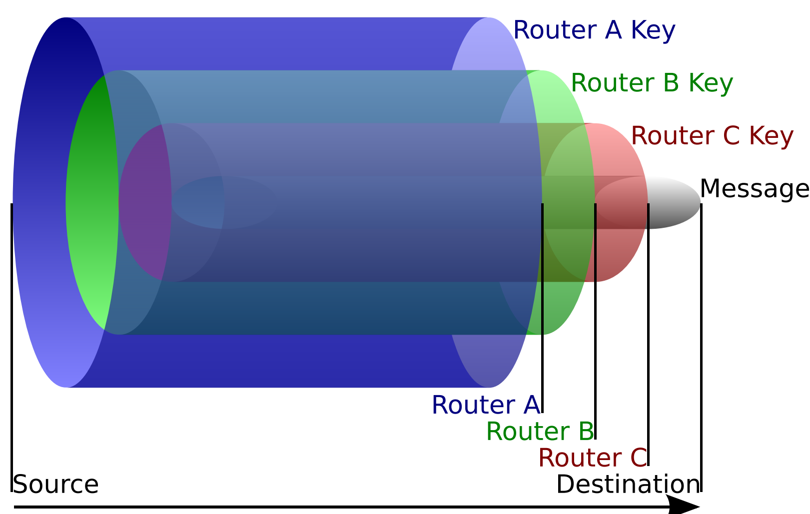

Abbildung 6.7: Schalenverschlüsselung beim Onion Routing (Harrison Neal, Lizenz: CC BY-SA)[6]

Abbildung 6.7 zeigt die Schalenverschlüsselung des Payloads beim Versand einer TUNNEL-Nachricht durch den Sender S über die Onion Router A bis C, wobei C der Empfänger ist. Die

[6] http://de.wikipedia.org/w/index.php?title=Datei:Onion_diagram.svg&filetimestamp=20080815233751 (abgerufen am 14.12.2012)

Message besteht dabei aus einer oder mehreren ByteStorm-Nachrichten, dem Tiger-Hash über diese und, im Falle eines Proxy-Tunnels, vorangesteller IP und Port.

Im Detail sieht der Payload von Tunnelnachrichten wie folgt aus:

- Länge der nachfolgenden IP-Adresse, 4 bei IPv4, 16 bei IPv6 (1 byte, nur falls Proxy-Tunnel)
- IP-Adresse des zum Proxy verbundenen Peers (variable Länge, nur falls Proxy-Tunnel)
- Port des zum Proxy verbundenen Peers (2 bytes, nur falls Proxy-Tunnel)
- ByteStorm-Nachricht m_1
- ByteStorm-Nachricht m_2
- ByteStorm-Nachricht ...
- ByteStorm-Nachricht m_n
- $h(m_1, m_2, ..., m_n)$

6.6.3 Tunnelaufbau

Bevor, wie oben beschrieben, Nachrichten durch den Tunnel gesendet werden, muss er natürlich zunächst einmal aufgebaut werden. Davon ausgehend, dass der Zielknoten und einige weitere Onion Router zufällig für den Tunnel ausgewählt wurden, werden alle Onion Router in eine Reihenfolge gebracht, wobei der Zielknoten das letzte Element ist. Damit ist eine Route für den Tunnel definiert, die die Nachrichten nehmen werden.

Da Tunnelnachrichten für jeden Onion Router separat mit einem symmetrischen Schlüssel verschlüsselt werden, muss mit jedem Router ein Schlüsselaustausch stattfinden. Das muss so passieren, dass jeder Router nur den für ihn bestimmten symmetrischen Schlüssel kennt. Da es sich bei ByteStorm um ein P2P-System für den Austausch von Dateien handelt und daher regelmäßig neue Verbindungen aufgebaut werden müssen, sollte dieser so schnell wie möglich erfolgen. Für einen erfolgreichen Tunnelaufbau ist, im Gegensatz z. B. zu TOR, nur ein Roundtrip über die Tunnelroute erforderlich.

Um dies zu ermöglichen, muss der Initiator alle nötigen Sitzungsschlüssel erzeugen. Für jeden der Onion Router werden zwei AES-Schlüssel und zwei Initialisierungsvektoren für den Counter-Modus erzeugt (jeweils einer pro Senderichtung). Zudem wird der öffentliche Schlüssel jedes Onion Routers benötigt, der im anonpk-Feld von Tracking-Nachrichten enthalten ist. Niemand außer dem Initiator darf die komplette Route kennen. Jeder Onion Router erfährt deswegen nur die Adressdaten des nachfolgenden Onion Routers.

Nun sind alle nötigen Artefakte vorhanden, um den Tunnelhandshake zu erzeugen. Der Tunnelhandshake ist immer die erste TUNNEL-Nachricht einer Verbindung. Mit ihr teilt der Initiator jedem Onion Router seinen symmetrischen Schlüssel, die Initialisierungsvektoren und die Adressdaten des nächsten Onion Routers auf dem Pfad mit.

Für jeden Onion Router des Tunnels wird ein eigener Handshake erzeugt. Zuerst wird der Handshake für den Empfänger erzeugt, dann für jeden Onion Router dem Pfad rückwärts folgend. Auf jeden Handshake, außer dem des Empfängers, folgt der Handshake des nachfolgenden Onion Routers.

Das Format des Handshakes h_x für einen Onion Router x lautet wie folgt:

- Die Summe len der Längen aller nachfolgenden Felder (1 Byte)
- Initialisierungsvektor iv_1 für die Senderichtung Initiator → Zielknoten (16 Bytes)
- Initialisierungsvektor iv_2 für die Senderichtung Zielknoten → Initiator (16 Bytes)
- Symmetrischer AES128-Schlüssel $k_{x,1}$ für die Senderichtung Initiator → Zielknoten (16 Bytes)
- Symmetrischer AES128-Schlüssel $k_{x,2}$ für die Senderichtung Zielknoten → Initiator (16 Bytes)
- Länge l_{ip} der IP-Adresse des nächsten Onion Routers, 4 bei IPv4, 16 bei IPv6 (1 Byte)
- IP-Adresse ip des nächsten Onion Routers (variabel)
- Port p des nächsten Onion Routers (2 Bytes)

Damit können Handshakes für alle Onion Router erzeugt werden. Eine Komponente fehlt aber noch. Wie erkennt der Empfänger, dass er der Empfänger ist? Da der Empfänger die Nachricht nicht an einen nächsten Hop weiterleiten muss, sind weder eine IP noch ein Port nötig. Daher werden die Felder l_{ip}, ip und p anders verwendet. Ist der Wert des Feldes l_{ip} gleich 0, dann handelt es sich um einen Direkttunnel. Ist der Wert 24 oder 48, dann handelt es sich um einen Proxy-Tunnel. Ist er 24, enthält das Feld ip die TPID des zu tauschenden Tempests. Ist er 48, so enthält es zusätzlich den Wurzelhash. Damit kann sich der Zielknoten sofort via Tracking-Protokoll für den Tempest registrieren, um im Namen des Initiators Verbindungen anzunehmen. Der Wert von p wird ignoriert.

Der Initiator hat nun für jeden Onion Router auf dem Pfad einen Handshake erzeugt. Damit jeder Router nur seinen Handshake lesen kann, wird jeder Handshake mit dem öffentlichen Schlüssel des jeweiligen Onion Routers per Elliptic Curve Integrated Encryption Scheme (ECIES) verschlüsselt. AES-CBC dient dabei als symmetrischer Algorithmus und SHA-1 als Hashfunktion für Schlüsselerzeugung und HMAC-Berechnung. Jetzt ist jeder Handshake unabhängig von

den anderen verschlüsselt. Dies ist ein Problem, da so jeder Router auf dem Pfad die Knotenreihenfolge ändern könnte, indem er die Reihenfolge der Handshakes vertauscht. Deshalb werden die Handshakes nach dem Zwiebelschalenprinzip verschlüsselt, genau wie alle nachfolgenden Nachrichten. Auf den, mit dem öffentlichen Schlüssel eines Knotens x verschlüsselten, Handshake-Block folgen die restlichen Handshakes. Diese werden per AES-CTR Stromchiffre verschlüsselt, welche mit iv_1 und $k_{x,1}$ initialisiert wird und für den Rest der Verbindung weiterverwendet wird (iv_2 und $k_{x,2}$ wird für die Rückrichtung verwendet).

Steht $E_{pub_x}(m)$ für das Verschlüsseln von m mittels ECIES mit dem öffentlichen Schlüssel des Routers x und $E_{k_{x,1}}(m)$ für das Verschlüsseln von m mit AES-CTR und dem an Router x zugewiesenen symmetrischen Schlüssel $k_{x,1}$ für den Hinweg, dann lässt sich der Handshake h_x für Onion Router x mit folgender rekursiver Formel beschreiben:

$$h_x = E_{pub_x}(len, iv_1, iv_2, k_{x,1}, k_{x,2}, l_{ip}, ip, p) \| E_{k_{x,1}}(h_{x+1})$$

Dabei ist h_{x+1} leer, falls x der Zielknoten ist.

Würde man den Tunnelhandshake in dieser Form in eine TUNNEL-Nachricht verpacken und senden, dann wäre jeder Onion Router in der Lange, aufgrund der Nachrichtenlänge die Länge des aus seiner Sicht verbleibenden Teils des Tunnels zu bestimmen. Dies stellt insbesondere beim ersten Onion Router ein Problem dar, da ihm dies die Möglichkeit einräumt, mit hoher Wahrscheinlichkeit zu bestimmen, dass sein Vorgänger der Initiator ist, wenn er die von letzterem maximal verwendete Tunnellänge abschätzen kann. Deshalb wird jedem Tunnelhandshake eine variable Menge von Zufallsdaten angehangen (Padding), um die tatsächliche Tunnellänge zu verschleiern. Wenn also A der erste Onion Router des Tunnels ist, dann ist $h_A \| pad$ Payload der TUNNEL-Nachricht.

Jeder Onion Router x, der den Tunnelhandshake erhält, entschlüsselt den ersten Teil von h_x mit seinem privaten Schlüssel $priv_x$ durch Anwenden der Funktion $D_{priv_x}(m)$ und initialisiert den AES-CTR Algorithmus mit iv_1 und $k_{x,1}$. Diesen nutzt er, um den Rest der Nachricht ($E_{k_{x,1}}(h_{x+1}) \| pad$) zu entschlüsseln. Dadurch erhält er $h_{x+1} \| D_{k_{x,1}}(pad)$. Router x stellt nun eine Verbindung zum nächsten Router mit der Adresse $ip : p$ her und sendet $h_{x+1} \| D_{k_{x,1}}(pad)$ in einer TUNNEL-Nachricht. Wird der Zielknoten schließlich erreicht, ist der Tunnel aufgebaut. Als Benachrichtigung über den erfolgreichen Tunnelaufbau und Beweis für die Fähigkeit zur korrekten Verschlüsselung wird eine zufällige Anzahl (zwischen 10 und 100) von IDLE-Nachrichten (ein Byte mit Wert 0) durch den Tunnel zum Initiator gesendet. Um Timing-Angriffe zu erschweren, sollte die Antwort mit einer zufälligen Zeitverzögerung gesendet werden.

Mit nur einem Tunnelroundtrip wurde der Tunnel erfolgreich eingerichtet und kann nun für den Austausch beliebiger ByteStorm-Nachrichten verwendet werden. Dazu wird das bereits be-

schriebene Nachrichtenformat verwendet. Bei Proxy-Tunneln ist darauf zu achten, die Adressinformationen zwecks Multiplexing in die Nachrichten zu inkludieren.

6.6.4 Tunnelabbau

Ein gesondertes Protokoll für den Abbau eines Tunnels ist nicht erforderlich. Wenn ein bereits aufgebauter Tunnel nicht mehr benötigt wird, reicht das Schließen der Verbindung aus. Jeder Onion Router ist dazu verpflichtet, die Verbindung in die andere Richtung ebenfalls zu trennen, falls einer der beiden Tunnelnachbarn sie trennt.

6.7 Abonnements / Feeds

Eine wünschenswerte, aber in anderen P2P-Systemen fehlende, Möglichkeit ist das regelmäßige und automatische Herunterladen von Dateien aus einer bestimmten Quelle oder Veröffentlichung. Beispiele sind das Herunterladen von Software-Updates oder regelmäßig erzeugten Medieninhalten (Serien, Podcasts, Video-Clips). Im Web werden zum Finden neuer Veröffentlichungen meist RSS/Atom-Feeds eingesetzt, die ein entsprechender Client in regelmäßigen Abständen abruft. Der Feed enthält unter anderem ein oder mehrere URLs, mit denen der entsprechende neue Inhalt heruntergeladen werden kann.

Mit ByteStorm lassen sich analog Feeds erzeugen, unter denen Inhalte in Form vom Tempests veröffentlicht werden können. Ein Feed ist nichts anderes als ein Squall, der aber keine Nutzdaten beschreibt, sondern eben einen Feed, welcher sich abonnieren lässt. Ein Squall, der einen Feed beschreibt, enthält das Feld `isfeed` mit dem Wert 1. Da keine Nutzdaten beschrieben werden, fehlen zudem die Felder `size` und `files`.

Da jeder Squall signiert sein muss, ist auch jeder Feed signiert. Um einen Tempest zu einem bestehenden Feed hinzuzufügen, muss für diesen ein neuer Squall erzeugt werden, dessen `feedids`-Liste den Wert aus dem `uuid`-Feld des Feeds enthält. Da es sich um eine Liste handelt, kann ein Tempest Teil mehrerer Feeds sein. Der neue Squall muss mit demselben Schlüsselpaar signiert werden, wie der Feed-Squall, um sicherzustellen, dass nur der Betreuer des Feeds neue Veröffentlichungen tätigen kann.

Feeds werden genauso via BubbleStorm publiziert wie jeder andere Squall. Zum Suchen von Feeds kann die reguläre Suchfunktion zum Suchen von Tempests verwenden werden (vgl. Abschnitt 6.3). Um ausschließlich Feeds zu finden, wird in der Suchanfrage einfach das Feld `isfeed` auf 1 gesetzt.

Durch das Abonnieren eines Feeds speichert der Client den Squall des Feeds lokal und prüft in regelmäßigen Abständen, ob neue Einträge für den Feed verfügbar sind. Die Einträge eines Feeds – also die dem Feed zugeordneten Tempests – lassen sich über die Suchfunktion ermitteln, indem eine Suche gestartet wird, die eine `feedids`-Liste mit der UUID des Feeds enthält. Optional kann die Suche weiter eingeschränkt werden, sodass z.B. nur Tempests gefunden werden, die seit der letzten bekannten Veröffentlichung dem Feed hinzugefügt wurden (Feld `createdateleast`). Für jede erhaltene Antwort muss überprüft werden, ob der Tempest mit demselben Schlüssel signiert wurde, wie der Feed. Ist dies der Fall, wird der Tempest automatisch heruntergeladen, andernfalls nicht.

Eine Besonderheit ist, dass ein Feed mehrere weitere Feeds zu einem einzigen Feed zusammenfassen kann, einem so genannten Compound-Feed. Dies geschieht, indem im Feed-Squall die `feedids`-Liste mit den UUIDs der zusammenzufassenden Feeds befüllt wird. Diese Feeds werden auch Sub-Feeds genannt und müssen nicht mit demselben Schlüssel signiert sein. Es gilt jedoch weiterhin, dass jeder Tempest in einem Feed mit demselben Schlüssel wie der Feed bzw. Sub-Feed signiert sein muss. Durch die Signierung des Compound-Feeds wird allen Sub-Feeds das Vertrauen ausgesprochen (Chain of Trust). Zu einem Compound-Feed können zusätzlich zu den Sub-Feeds auch eigene Tempests hinzugefügt werden.

6.8 HTTP-Gateways aka. ByteStorm CDN

Jeder ByteStorm Peer kann als HTTP-Gateway dienen. Das bedeutet, dass vollständig heruntergeladene Inhalte aus Tempests für den Download via HTTP angeboten werden können. Alle HTTP-Gateways aus dem Schwarm eines Tempests kommunizieren miteinander, um für ein Load Balancing zu sorgen. Soll eine Datei via HTTP heruntergeladen werden, so folgt der Benutzer einem Link auf die gewünschte Datei auf einem beliebigen HTTP-Gateway. Hat dieses genügend freie Kapazitäten, sendet es die Datei direkt an den Benutzer. Andernfalls sucht das Gateway im Tempest-Schwarm nach einem anderen Gateway und leitet die Anfrage an dieses weiter.

Mit ByteStorm lässt sich ein vollwertiges, automatisiertes Content Delivery Network (CDN) mit einer verteilten Infrastruktur für das Anbieten von HTTP-Downloads aufbauen, indem Abonnement- und Gateway-Funktion in Kombination genutzt werden. Durch das Abonnieren von Channels kann neuer Content automatisch auf beliebig viele CDN-Knoten repliziert und per HTTP-Gateway bereitgestellt werden. Die Knoten kümmern sich selbstständig um Lastverteilung und reagieren auf ausgefallene Knoten. Gleichzeitig kann der Content auch via P2P verteilt werden, sodass Content Distribution auf hybride Weise (P2P und HTTP) möglich ist.

Es gibt zwei Möglichkeiten, um herauszufinden, ob ein Peer im Schwarm eines Tempests ein HTTP-Gateway für diesen Tempest ist. Zum einen ist im Datensatz für jeden Peer, der über

das Tracking-Protokoll abgewickelt wird, der Port des HTTP-Gateways enthalten, sofern das Gateway aktiviert ist. Zum anderen wird der Port ebenfalls beim Handshake zwischen Peers übertragen. Jeder Peer kann also sowohl durch das Tracking als auch durch eingehende Verbindungen von neuen HTTP-Gateways erfahren. Sollte der über das Tracking-Protokoll erfahrene Port dem Port aus dem Handshake widersprechen, so gilt letzterer.

Herzstück des HTTP-Gateways ist ein minimalistischer HTTP-Server, der Teil jedes ByteStorm Clients ist. Dieser hat zwei Funktionen: Ausliefern von Dateien und Load Balancing zwischen HTTP-Gateways. Die HTTP-Gateway Funktion kann nach Belieben ein- und ausgeschaltet werden. Standardmäßig ist sie ausgeschaltet.

Jedes HTTP-Gateway kann eine begrenzte Anzahl gleichzeitiger Downloadvorgänge bedienen. Ein Downloadplatz wird auch als Slot bezeichnet. Die Slot-Anzahl kann individuell pro Gateway konfiguriert werden, um dem Benutzer eine angemessene Dienstgüte bieten zu können. Ist jeder Slot durch einen Download belegt, kann das Gateway keine weiteren Downloadanfragen bedienen. In diesem Fall wird die Anfrage an ein anderes Gateway weitergeleitet, das noch freie Slots besitzt (HTTP Statuscode 302 oder 307). Gibt es kein solches, wird der Benutzer entweder in eine Warteschlange eingereiht oder die Anfrage wird wegen Überlastung abgelehnt (HTTP Statuscode 503).

Um Weiterleitungen auf andere Gateways mit freien Slots effizient durchzuführen, muss ermittelt werden können, ob ein Gateway überhaupt noch über freie Slots verfügt. Dafür gibt es ein Paar aus Protokollnachrichten, nämlich GETGATEWAYSTATUS und GATEWAYSTATUS. Wie die Namen bereits suggerieren, kann mit ihnen der Status eines anderen Gateways erfragt bzw. der des eigenen übermittelt werden. Die wichtigsten Daten in dieser Nachricht sind die Anzahl der noch verfügbaren Slots, die Unterstützung von HTTPS durch das Gateway und ein URL-Präfix, das beim Generieren von URLs von Bedeutung ist. Um Weiterleitungsentscheidungen auf möglichst aktuellen Daten zu basieren, sollte der Status eines oder mehrerer Gateways unmittelbar vor dem Senden einer HTTP Weiterleitung ermittelt werden, um das Weiterleiten auf ein ausgelastetes Gateway zu vermeiden.

Um einen Download einer Datei aus einem Tempest via HTTP zu starten, benötigt der Benutzer eine URL zu der gewünschten Datei. Diese kann sich auf einer Internetseite befinden, in einem Updater-Programm eingebettet sein, usw. Es gibt zwei gültige URL-Muster:

- `http(s)://<ip>:<gwPort>/<TPID>/<relativeFilePath>`
- `http(s)://<ip>:<gwPort>/<urlPrefix><TPID>/<relativeFilePath>`

Das erste Muster ist zu verwenden, wenn die GATEWAYSTATUS-Nachricht des entsprechenden Gateways kein Präfix definiert. Andernfalls gilt das zweite Muster. Der Schema-Bezeichner lautet entweder „http“ oder „https“, falls das Gateway HTTPS unterstützt. IP und Port des Gateways werden aus dem Tracking-Protokoll bzw. dem letzten Handshake übernommen. Mit der TPID wird der Tempest identifiziert. Der letzte Pfadbestandteil der URL ist der relative Pfad einer Datei des Tempests, wie im Squall angegeben. Groß/Kleinschreibung ist zu beachten. Jedes Gateway kann ein Präfix definieren, das dem Pfadanteil der URL vorangestellt und durch die GATEWAYSTATUS-Nachricht mitgeteilt wird. Dadurch können Pfade an die Konventionen von Webseiten angepasst, Anfragen über einen herkömmlichen Webserver geleitet und beispielsweise Zählskripte verwendet werden. Für diesen Zweck wird an das Präfix nicht automatisch ein Schrägstrich als Pfadtrenner angefügt.

Das typische Verhalten eines HTTP-Gateways zur Beantwortung von HTTP-Anfragen ist im Algorithmus 6.1 festgehalten.

```
receive HTTP request
if tempest with requested TPID is not present locally then
    reject request (HTTP code 404)
    //alternatively: find a gateway that has the tempest and redirect (HTTP code 301)
    return ;
end if
if requested file does not exist in tempest then
    reject request (HTTP code 404)
    return ;
end if
if free slot available then
    send requested file
else
    send GETGATEWAYSTATUS to other gateways in swarm
    wait for GATEWAYSTATUS replies or timeout
    if at least one gateway with at least one free slot replied then
        select one of the gateways with a free slot
        build the new URL
        send HTTP redirect to client (HTTP code 302 or 307)
    else
        reject request (HTTP code 503)
        //alternatively: put request in queue and send website with user's current queue status
    end if
end if
```

Algorithmus 6.1: Pseudoalgorithmus für ein typisches HTTP-Gateway

6.9 Das Protokoll

Dieses Kapitel stellt das von ByteStorm verwendete Nachrichtenprotokoll vor. Dieses Protokoll wird verwendet, um Datentransfers zwischen Peers durchzuführen sowie das anonymisierte Tunneln von Verbindungen zu gewährleisten. Die in vorhergehenden Kapiteln bereits besprochene Kommunikation über BubbleStorm ist *nicht* Teil dieses Protokolls, da BubbleStorm ein eigenes Protokoll implementiert, welches nicht Gegenstand dieser Arbeit ist. Das ByteStorm Protokoll kann sowohl TCP als auch CUSP als zuverlässiges, verbindungsorientiertes Transportprotokoll einsetzen. Falls eine Verschlüsselung des Nachrichtenstroms erforderlich ist, ist die Verwendung von CUSP notwendige Bedingung, denn dieses Transportprotokoll verschlüsselt auf Transportebene. Eine Verschlüsselung auf Anwendungsebene findet nicht statt, es sei denn, die Anonymisierungsfunktion wird genutzt.

Alle Protokollnachrichten haben das Nachrichtenformat <Message ID><Payload>. Die Message ID ist ein Byte, das die Nachricht identifiziert. Der Payload ist eine beliebige Kombination von Datenfeldern fester oder variable Länge. Feldern mit variabler Länge wird ein Feld fester Länge vorangestellt, das die Längeninformation für das nachfolgende Feld enthält. Bei sämtlichen Zahlenwerten des Protokolls handelt es sich um vorzeichenlose (unsigned) Ganzzahlen, die in Network Byte Order übertragen werden.

Einige Nachrichtenfelder werden in mehreren Nachrichten verwendet. Diese Felder und ihre Semantik werden im Folgenden aufgelistet:

- <id> Jede Protokollnachricht hat eine eigene ID und dient dazu, die Nachrichten zu unterscheiden (s.o.). Die ID ist eine 8 Bit Zahl.
- <len> Eine 8 Bit Zahl, die die Länge des darauffolgenden Feldes in Bytes angibt, sofern nicht anders angegeben. Das Feld ist vor Feldern mit variabler Länge zu finden.
- <len2> Eine 16 Bit Zahl, die die Länge des darauffolgenden Feldes in Bytes angibt, sofern nicht anders angegeben. Das Feld ist vor Feldern mit variabler Länge zu finden.
- <len4> Eine 32 Bit Zahl, die die Länge des darauffolgenden Feldes in Bytes angibt, sofern nicht anders angegeben. Das Feld ist vor Feldern mit variabler Länge zu finden.
- <treeCoord> Die 64 Bit Koordinate eines Baumknotens im Tempest Tree. Das Adressierungsschema für Baumknoten wurde im Unterkapitel 6.2.4 bereits vorgestellt. Das höchstwertige Byte enthält die Tiefe t, die verbleibenden 3 Bytes den Index i – die Position des Knotens auf der Ebene t. Die Werte von t und i beginnen bei 0, wobei $t = 0$ und $i = 0$ dem Wurzelknoten entspricht.

6.9.1 Protokoll-Nachrichten

Alle folgenden Protokoll-Nachrichten werden im Format *Name: <Feld1><Feld2><Feld3>...* dargestellt. Gesendet werden die Felder in der angegebenen Reihenfolge. Der Name ist nicht Teil der gesendeten Nachricht, sondern nur ein Bezeichner für die Nachricht.

KEEPALIVE: <id=0>

Clients können die Verbindung zu anderen Peers trennen, wenn diese längere Zeit keine Nachricht gesendet haben. Um dies zu vermeiden, kann nach einiger Zeit der Untätigkeit diese Nachricht gesendet werden.

HANDSHAKE: <id=1><ByteStormv1><TPID/RootHash><MaxTreeDepth><MaxInHaveDepth><MaxOutHaveDepth><MaxQueuedSegments><BubbleStormPort><ByteStormPort><HTTPGatewayPort><len><PeerId><Compression><len2><Extensions>

Die HANDSHAKE-Nachricht ist die erste Nachricht, die sowohl der Initiator als auch das Ziel einer Verbindung sendet. Da die im HANDSHAKE enthaltenen Parameter vom jeweiligen Tempest abhängen, muss das Ziel auf die HANDSHAKE-Nachricht warten. Somit ist mindestens die Dauer eines Roundtrips nötig, bevor Daten ausgetauscht werden können. Die einzelnen Felder haben die folgende Bedeutung:

- Auf die Nachrichten ID 1 folgt der String *ByteStormv1*, der das Protokoll identifiziert.
- Das Feld *TPID/RootHash* ist 192 Bit lang und enthält entweder die TPID oder den Wurzelhash des Tempests, der übertragen werden soll. Die TPID wird vom Initiator verwendet, solange er den korrekt signierten Wurzelhash nicht kennt. Sobald er den Wurzelhash kennt, verwendet er diesen. Dadurch wird das Feature ermöglicht, die Schwärme mehrerer Tempests mit den gleichen Dateien zu einem großen Schwarm zusammenzufassen. (vgl. Abschnitt 6.4; 24 Bytes)
- *MaxTreeDepth* gibt an, bis zu welcher Tiefe der Tempest Tree lokal gespeichert wird. So kann die Gegenseite vermeiden, Hashes aus dem Tempest Tree von einem Peer anzufragen, die dieser gar nicht haben kann. In Verbindung mit der Nutzung der Informationen der HAVE-Nachrichten kann verhindert werden, dass Anfragen nach Hashwerten unterhalb der MaxTreeDepth-Grenze nicht beantwortet werden können. (1 Byte)
- *MaxInHaveDepth* gibt die maximale Tiefe im Tempest Tree an, bis zu der der eigene Client wünscht, HAVE-Nachrichten zu empfangen und erlaubt somit, die Menge des empfangenen Overheads zu steuern. (1 Byte)
- *MaxOutHaveDepth* gibt die maximale Tiefe im Tempest Tree an, bis zu der der eigene Client HAVE-Nachrichten senden kann. Jeder Client darf HAVE-Nachrichten nur dann an einen anderen Peer senden, wenn der Tiefenwert t der Koordinate nicht größer ist als die Werte vom selbst gewählten MaxOutHaveDepth und der MaxInHaveDepth der Gegenseite. Es wird also immer das Minimum der von beiden Verbindungsendpunkten gewählten Werte verwendet, wobei für jede Übertragungsrichtung ein unabhängiger zu verwendender Wert ausgehandelt wird. Sendet Peer A also MaxInHaveDepth=5, MaxOutHaveDepth=4 und Peer B sendet MaxInHaveDepth=3, MaxOutHaveDepth=6, dann darf A nur dann eine HAVE-Nachricht an B senden, wenn die Koordinate maximal auf die Tiefe $min(MaxOutHaveDepth(A), MaxInHaveDepth(B)) = min(4, 3) = 3$ verweist und B darf nur HAVE-Nachrichten bis zur Tiefe $min(6, 5) = 5$ senden. Dieser Grenzwert wird im Folgenden auch als maximale HAVE-Tiefe bezeichnet. (1 Byte)

- *MaxQueuedSegments* ist die maximale Anzahl an 16 KiB Datensegmenten, die durch die Gegenseite mittels REQUEST-Nachrichten in der Warteschlage zu jedem Zeitpunkt platziert werden dürfen. Bei Überschreitung dieser Grenze wird der Request per REJECT-Nachricht abgewiesen. (2 Bytes)
- *BubbleStormPort* ist der Port, auf dem lokal auf eingehende BubbleStorm-Verbindungen gewartet wird. Der Wert kann 0 sein, wenn eingehende Verbindungen nicht erwünscht sind. (2 Bytes)
- *ByteStormPort* ist der Port, auf dem lokal auf eingehende ByteStorm-Verbindungen gewartet wird. Der Wert kann 0 sein, wenn eingehende Verbindungen nicht erwünscht sind. (2 Bytes)
- *HTTPGatewayPort* ist der Port, auf dem lokal auf eingehende HTTP-Verbindungen gewartet wird, um lokale Tempests per HTTP auszuliefern. Der Wert ist 0, wenn das HTTP-Gateway lokal deaktiviert ist. (2 Bytes)
- *PeerId* identifiziert die verwendete ByteStorm-Implementierung und den Peer. Es handelt sich um einen UTF8-String im Format `_xxxxxx_yyyyyy`. `xxxxxx` steht für die Bezeichnung und Version der Implementierung. Da der Unterstrich (`_`) als Begrenzungszeichen vor und nach Bezeichnung/Version dient, ist er in dieser unzulässig. `yyyyyy` ist eine zufällig generierte Zeichenfolge der Länge 12. Die PeerId kann zur Wiedererkennung von Peers verwendet werden, ist jedoch kein zuverlässiges Kriterium, da es jeder Implementierung freisteht, die PeerId jederzeit zu ändern. (variabel, max. 100 Bytes)
- *Compression* gibt die unterstützten Kompressionsalgorithmen an. Unterstützte Kompressionsalgorithmen werden durch ein 1-Bit angezeigt. Derzeit sind 2 Kompressionsalgorithmen spezifiziert; Gzip auf Bit 0 (`compression & 1 == 1`) und Deflate auf Bit 1 (`compression & 2 == 2`). Kompression kann nur verwendet werden, wenn die Schnittmenge der unterstützten Kompressionsalgorithmen beider Seiten nicht leer ist. Besteht das Feld nur aus 0-Bits, ist Kompression deaktiviert. (1 Byte)
- Das Feld *Extensions* kann verwendet werden, um die Unterstützung von Erweiterungen zu kommunizieren. Jede Erweiterung kann eigene Protokollnachrichten definieren, die nicht Teil des ByteStorm-Protokolls sind. Das Feld verwendet Bencoding und besteht aus einer Liste von Dictionaries. Jede Erweiterung ist in einem eigenen Dictionary spezifiziert, welches mindestens einen Schlüssel *name* mit einem UTF-8 String als Wert enthält. Definiert die Erweiterung eigene Protokollnachrichten, so muss zudem ein Schlüssel *idlist* mit einer Liste aus Integern zwischen 32 und 255 vorhanden sein, für jede Erweiterungsnachricht eine. Die Reihenfolge der Nachrichten ist Teil der Spezifikation der Erweiterung. Es ist darauf zu achten, dass keine zwei Nachrichten dieselbe ID haben dürfen, auch wenn diese von unterschiedlichen Erweiterungen hinzugefügt werden. Falls die Gegenseite die jeweilige Erweiterung ebenfalls unterstützt, so muss diese die hier definierten IDs für das Senden der Erweiterungsnachrichten verwenden. Erweiterungsnachrichten werden immer

im Format <ID><Payload> gesendet, wie alle ByteStorm-Nachrichten. Werden keine Erweiterungen unterstützt, enthält das vorangestellte *len*-Feld den Wert 0.

MAX_QUEUED_SEGMENTS: <id=2><newValue>

Der im Handshake übermittelte Wert für *MaxQueuedSegments* kann während einer Verbindung jederzeit geändert werden, z.B. wenn der im Handshake übermittelte Wert zu klein war, sodass die Request-Warteschlange regelmäßig leer läuft. Der Sender teilt seinem Gegenüber also mit, dass letzterer ab sofort maximal <newValue> viele Datensegmente gleichzeitig in der Warteschlange des Senders platziert haben darf. Der ebenfalls 2 Bytes lange neue Wert ersetzt den vorhergehenden. Falls die neue Grenze kleiner ist als die Menge der zum Zeitpunkt der Änderung in der Request-Warteschlange befindlichen Segmente, so werden diese Requests trotzdem abgearbeitet und nicht verworfen. Nach der Änderung der maximalen Warteschlangenlänge eintreffende REQUEST-Nachricht werden, wie üblich, per REJECT-Nachricht abgewiesen, falls die Anzahl mit der REQUEST-Nachricht neu angefragten Segmente die neue Grenze überschreiten würde.

MAX_IN_HAVE_DEPTH: <id=3><newValue>

Der Wert von *MaxInHaveDepth* aus dem Handshake lässt sich über die Nachricht während einer Verbindung ändern. Durch Erhöhen des Wertes kann dem Gegenüber mitgeteilt werden, dass HAVE-Nachrichten ab sofort häufiger erwünscht sind, durch Verringern werden sie reduziert. Der Wert für die neue maximale Baumtiefe empfangener HAVE-Nachrichten ist 1 Byte lang.

MAX_OUT_HAVE_DEPTH: <id=4><newValue>

Analog kann auch der Wert von *MaxOutHaveDepth* jederzeit angepasst werden, um Bereitschaft zum Senden von weniger oder mehr HAVE-Nachrichten zu signalisieren. Die Wert für die neue maximale Baumtiefe gesendeter HAVE-Nachrichten ist 1 Byte lang. Die maximale HAVE-Tiefe wird weiterhin durch das Minimum des eigenen Werts von *MaxOutHaveDepth* und dem Wert von *MaxInHaveDepth* der Gegenseite bestimmt.

HAVE_VERIFIED: <id=5><treeCoord>

Die HAVE_VERIFIED-Nachricht unterrichtet andere Peers darüber, dass ein oder mehrere Segmente heruntergeladen und verifiziert wurden und signalisierisert damit, dass diese Segmente von anderen Peers heruntergeladen werden können. Als Parameter enthält die Nachricht eine Tempest Tree Koordinate. Alle Segmente, die vom an der Koordinate gespeicherten Hash abgedeckt sind, sind vollständig und mittels Tempest Tree verifiziert. Das bedeutet insbesondere

auch, dass der korrekte Hash für den (durch die Koordinate bezeichneten) Baumknoten und alle gespeicherten Nachfahren bekannt ist, sodass diese Hashwerte von anderen Peers angefragt werden können. Eine HAVE_VERIFIED-Nachricht mit der (Wurzel)Koordinate $(0, 0)$ bedeutet also, dass alle Datensegmente komplett sind. HAVE_VERIFIED-Nachrichten werden potenziell immer dann gesendet, wenn ein empfangenes Datensegment erfolgreich verifiziert wurde. Sie werden nach den in Abschnitt 6.2.7 beschriebenen Regeln gesendet. Ob tatsächlich eine Nachricht gesendet wird, hängt von der ausgehandelten maximalen HAVE-Tiefe ab, die für jede Verbindung individuell ist (s.o.). Wenn durch den Empfang eines Segments beispielsweise ein Knoten auf der Tiefe 4 vervollständigt wird, wird an alle verbundenen Peers eine HAVE-Nachricht gesendet, die bis zu dieser Tiefe gemäß Verbindungsparameter HAVE-Nachrichten erhalten.

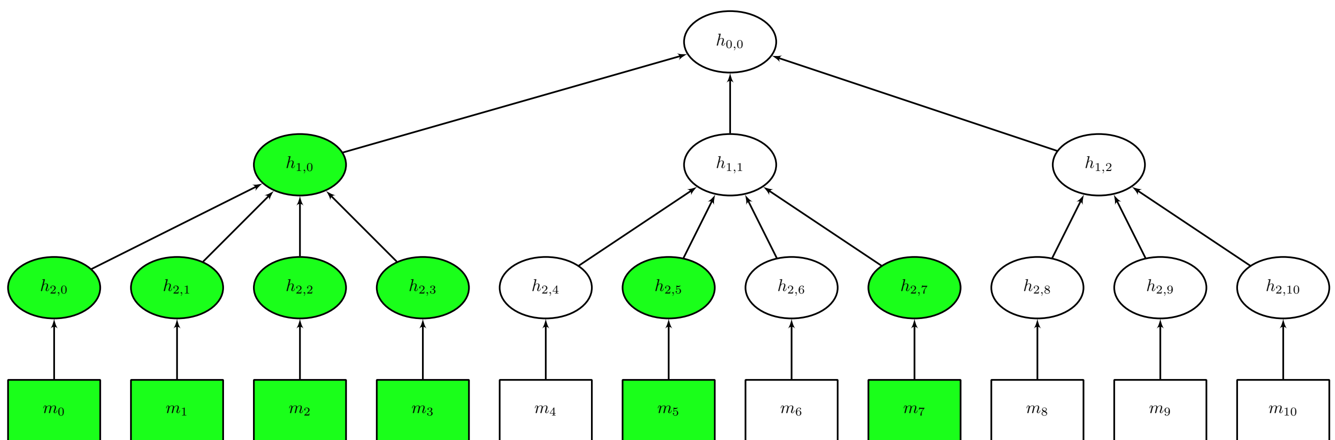

Abbildung 6.8: Vollständige Datensegmente von Peer B

Zur besseren Illustration ein detailiertes Beispiel: Ein Peer A ist mit einem Peer B verbunden. Abbildung 6.8 zeigt den aktuellen Zustand des Tempests von Peer B aus dessen Sicht. Durch den Handshake wurde ausgehandelt, dass Peer B nur HAVE-Nachrichten bis zur Tiefe 1 an A sendet. Bis zum gegenwärtigen Zeitpunkt hat Peer A nur eine HAVE-Nachricht mit der Koordinate $h_{1,0}$ erhalten. Von den ebenfalls bereits durch B fertiggestellten Knoten $h_{2,5}$ und $h_{2,7}$ weiß A nichts, da A auf Tiefe 2 keine HAVE-Nachrichten erhält. Peer B lädt nun weitere Datensegmente von einem dritten Peer herunter. Er empfängt das Segment m_4 und verifiziert es erfolgreich, sodass der Knoten $h_{2,4}$ vollständig ist. A erhält aber wiederum keine HAVE-Nachricht, da diese die Koordinate $(2, 4)$ enthalten würde, die Tiefe also abermals 2 ist. Nun erhält B auch das Segment m_6 und verifiziert es, sodass der Knoten $h_{2,6}$ vollständig ist. Da nun alle Kinder des Elternknotens $h_{1,1}$ vollständig sind, ist dieser Knoten ebenfalls vollständig. Weil nach den Regeln in Abschnitt 6.2.7 eine HAVE-Nachricht immer die Koordinate des höchstmöglichen fertigen Knotens auf dem Wurzelpfad enthält, lautet die Koordinate in diesem Fall $(1, 1)$. Diese liegt innerhalb der Tiefenbegrenzung, sodass A eine HAVE-Nachricht mit dieser Koordinate erhält. Peer A weiß nun also, dass Peer B die Segmente m_4 bis m_7 soeben fertiggestellt hat.

HAVE_DOWNLOADED: <id=6><treeCoord>

HAVE_DOWNLOADED benachrichtigt ebenfalls andere Peers über das Fertigstellen von Datensegmenten über eine Baumkoordinate. Sie hat jedoch lediglich die Semantik, dass alle Datensegmente im betreffenden Teilbaum heruntergeladen wurden, jedoch nicht, dass sie auch verifiziert worden sind. Somit kann bei Empfang von HAVE_DOWNLOADED auch nicht davon ausgegangen werden, dass der Sender der Nachricht den korrekten Hashwert kennt. HAVE_DOWNLOADED kann alternativ zur Nachricht HAVE_VERIFIED gesendet werden. Dies ist inbesondere dann sinnvoll, wenn lokal nur wenige Ebenen des Tempest Trees gespeichert werden oder sogar nur der Wurzelhash. Auf diese Weise können auch unverifizierte Segmente zum Download angeboten und somit Uploadkapazititäten von Peers genutzt werden, die andernfalls erst sehr spät oder gar erst nach vollständigem Download des Tempests mit dem Upload beginnen können. Da unverifizierte Segmente potenziell fehlerhaft sein können, ist es jeder Implementierung des Protokolls selbst überlassen, ob sie unverifizierte Segmente von anderen Peers herunterlädt oder nicht. Das Senden von HAVE_DOWNLOADED-Nachrichten ist immer dann unzulässig, wenn stattdessen eine HAVE_VERIFIED-Nachricht gesendet werden kann, sprich wenn ein korrekter Hash bekannt ist, mit dem der betreffenden Baumknoten verifiziert werden kann. Nach dem Senden einer HAVE_DOWNLOADED-Nachricht ist das Senden von HAVE_VERIFIED mit derselben Koordinate zulässig, umgekehrt jedoch nicht.

BITMASK_VERIFIED: <id=7><level><len4><bitmask>

Die BITMASK_VERIFIED-Nachricht wird nach dem Handshake von beiden Endpunkten gesendet, um den jeweils anderen Peer über den lokalen Tempest-Zustand in Kenntnis zu setzen. Der Parameter `bitmask` enthält jeweils ein Bit für jeden Knoten von links nach rechts im Tempest Tree auf der Baumebene, die vom Parameter *level* (1 Byte) vorgegeben wird. Ein 1-Bit hat dieselbe Semantik wie eine HAVE_VERIFIED-Nachricht für den jeweiligen Knoten, steht also für einen vollständigen und verifizierten Knoten. Die Baumebene sollte möglichst sinnvoll gewählt werden. Beispielsweise kann der Parameter *MaxOutHaveDepth* aus dem gesendeten Handshake verwendet werden. Unvollständige Bytes in der Bitmaske werden am Ende aufgefüllt. Alternativ zur BITMASK_VERIFIED-Nachricht können nach dem Handshake auch HAVE_VERIFIED-Nachrichten gesendet werden, wenn die dadurch entstehende Datenmenge kleiner ist, als beim Versenden der BITMASK_VERIFIED-Nachricht. Das ist beispielsweise immer dann der Fall, wenn der Tempest vollständig ist. In diesem Fall reicht eine einzelne HAVE_VERIFIED-Nachricht mit der Koordinate $(0, 0)$ aus. Neue Peers ohne fertige Segmente können auf beides verzichten.

BITMASK_DOWNLOADED: <id=8><level><len4><bitmask>

BITMASK_DOWNLOADED hat dieselbe Funktion wie BITMASK_VERIFIED, zeigt aber analog zu HAVE_DOWNLOADED nur an, dass die entsprechenden Baumknoten vollständig sind, aber nicht verifiziert. Sie sollte von Peers mit sehr niedriger Baumtiefe verwendet werden.

REQUEST: <id=9><treeCoord>

Mit dieser Nachricht können ein oder mehrere Datensegmente angefragt werden. Für die Anfrage wird dazu eine Baumkoordinate benutzt. Dadurch können alle Segmente unterhalb des entsprechenden Knotens auf einmal angefragt werden. Es ist darauf zu achten, dass die Anfrage den Verbindungsparameter *MaxQueuedSegments* der Gegenseite nicht überschreitet.

CANCEL: <id=10><treeCoord>

Zuvor gesendete Anfragen können zurückgezogen werden. Als Parameter wird die zuvor in einer REQUEST-Nachricht gesendete Koordinate oder die Koordinate eines der Kinder verwendet. Damit können einzelne Requests auch teilweise zurückgezogen werden, z.B. weil einige Datensegmente, die vom ursprünglichen Request abgedeckt sind, bereits anderweitig heruntergeladen wurden.

REJECT: <id=11><treeCoord>

Erhaltene Anfragen können ganz oder teilweise abgewiesen werden, indem die zuvor in einer REQUEST-Nachricht empfangene Koordinate oder die Koordinate eines der Kinder als Parameter verwendet wird. Segmentanfragen dürfen niemals einfach ignoriert werden. Dies wäre eine Protokoll-Verletzung. Angefragte Segmente müssen entweder per SEGMENT-Nachricht übermittelt oder zurückgewiesen werden.

SUGGEST: <id=12><treeCoord>

Ein Peer kann einem anderen Peer unverbindlich Segmente zum Herunterladen vorschlagen. Das kann nützlich sein, um die Anzahl von Festplattenzugriffen beim Hochladen von Segmenten zu minimieren und somit die maximale Upload-Rate zu optimieren, indem solche Segmente vorgeschlagen werden, die sich noch im Dateicache befinden.

SEGMENT: <id=13><treeCoord><len2><data>

Zur Übertragen der eigentlichen Daten eines Tempests dient die SEGMENT-Nachricht. Als Parameter enthält sie die Tempest Tree Koordinate genau eines einzelnen Segments. Die Koordinate bezeichnet also immer einen Knoten auf der untersten Ebene des Baums. Es folgen die

Daten des Segments. Ein Segment ist immer 16 KiB groß. Die einzige Ausnahme stellt das letzte Segment von einem Tempest dar, welches kürzer sein kann.

GETSINGLEHASH: <id=14><treeCoord>

Um den lokalen Tempest Tree während des Downloads mit Hashwerten zu füllen, die zum Verifizieren heruntergeladener Daten dienen, können mit dieser Nachricht einzelne Hashwerte angefragt werden. Mit der Anfrage wird der Hashwert an genau der angegebenen Baumkoordinate erfragt. Der Sender hat sicherzustellen, dass der Empfänger den jeweiligen Hashwert auch tatsächlich kennt. Dies ist immer genau dann der Fall, wenn der Empfänger zuvor eine HAVE_VERIFIED- oder BITMASK_VERIFIED-Nachricht gesendet hat, die die Komplettierung des Knotens an der Baumkoordinate (oder einem seiner direkten oder indirekten Eltern) signalisiert, und die Tiefe t nicht größer ist, als der Parameter *MaxTreeDepth* aus der erhaltenen Handshake-Nachricht.

SINGLEHASH: <id=15><treeCoord><hash>

Der Empfänger hat auf jede GETSINGLEHASH-Nachricht mit dieser Nachricht zu antworten. Der übermittelte Hashwert ist 192 Bit (24 Bytes) lang und entspricht dem lokal im Tempest Tree an der betreffenden Koordinate gespeicherten Hashwert.

GETMULTIPLEHASHES: <id=16><treeCoord><count>

Es lassen sich auch auf einmal mehrere Hashes einer Baumebene erfragen. Die Baumkoordinate gibt in diesem Fall einen Bezugsknoten an, der Parameter *count* (4 Bytes) die Anzahl zurückzuliefernder Hashes. Beginnend ab der Bezugskoordinate werden *count* Hashwerte mit aufsteigendem Index angefragt, sprich alle Hashwerte beginnend ab dem Knoten `(t,i)` bis einschießlich dem Knoten `(t,i+count-1)`. Ist $count = 1$, so hat diese Nachricht dieselbe Semantik wie GETSINGLEHASH. Für das Versenden der Nachricht gelten die gleichen Regeln, wie für GETSINGLEHASH. Der Sender muss also darauf achten, dass der Empfänger den Knoten an der Bezugskoordinate komplettiert hat.

MULTIPLEHASHES: <id=17><treeCoord><count><hashes>

Der Empfänger einer GETMULTIPLEHASHES-Nachricht kann selbst entscheiden, ob er die Anfrage beantwortet oder nicht, da Anfragen je nach Parameter potenziell zu sehr langen Antworten führen können. Die Antwort enthält – neben der Bezugskoordinate und dem Zähler *count* (4 Bytes) aus der Anfrage – alle Hashes der Knoten `(t,i)` bis einschießlich `(t,i+count-1)`. Da jeder Hash 24 Bytes lang ist, ist das Feld *hashes* stets $count \cdot 24$ Bytes lang.

GETROOTHASH: <id=18>

GETROOTHASH wird von einem neuen Peer nach dem Handshake gesendet, wenn der korrekte Root Hash noch nicht bekannt ist. Die in der ROOTHASH-Antwort enthaltene Signatur wird überprüft. Im Falle ihrer Korrektheit wird der Root Hash und die Signatur lokal gespeichert, um für die Prüfung der Nutzdaten, empfangener Hashes und bei künftigen Handshakes verwendet zu werden und um selbst GETROOTHASH-Nachrichten beantworten zu können. Ist der Empfänger der Erzeuger des Tempests und noch nicht in der Lage, die Anfrage zu beantworten, weil der initiale Hashvorgang noch nicht abgeschlossen ist, dann speichert er alle erhaltenen Anfragen und beantwortet sie zeitnah nach Abschluss des Hashvorgangs.

ROOTHASH: <id=19><rootHash><len2><rootSig>

Als Antwort auf Anfragen nach dem Root Hash wird dieser (24 Bytes), zusammen mit der vom Erzeuger des Tempests generierten Signatur *RootSig* (wie in 6.2.6 definiert), gesendet.

ENABLE_COMPRESSION: <id=20><algorithm>

Mit dieser Nachricht wird die Kompression des Nachrichtenstroms aktiviert. Alle nachfolgenden Nachrichten sind mit dem als Parameter angegebenen Algorithmus komprimiert. Der *algorithm* Parameter ist ein Byte, bei dem immer genau ein Bit auf 1 gesetzt ist. Eine Kompression ist nur möglich, wenn beide Seiten mindestens einen gemeinsamen Algorithmus unterstützen. Das gesetzte Bit muss sich daher an einer Position befinden, an der sich beim Schnitt der *compression*-Parameter aus den Handshake-Nachrichten beider Seiten ebenfalls ein 1-Bit befindet.

DISABLE_COMPRESSION: <id=21>

Die Kompression wird abgeschaltet. Nachfolgende Nachrichten werden nicht mehr komprimiert. Dies ist die letzte komprimierte Nachricht.

GETGATEWAYSTATUS: <id=22><TPID>

Um lokal nicht erfüllbare Anfragen über das HTTP Gateway an andere Peers sinnvoll zu delegieren, muss ihr aktueller Status bekannt sein, insbesondere die Auslastung ihrer Kapazitäten. Der Status des HTTP-Gateways im Bezug auf ein Tempest kann mit dieser Nachricht abgerufen werden. Dazu wird die TPID als einziger Parameter gesendet. Diese Nachricht darf gesendet werden, ohne dass zuvor ein Handshake stattfand. Sie kann auch die einzige Nachricht sein, die während einer Verbindung gesendet wird, sodass der Statusabruf auch in Echtzeit bei einer eingehenden HTTP Downloadanfrage erfolgen kann, ohne nennenswerte Verzögerungen zu erzeugen.

GATEWAYSTATUS:<id=23><TPID><gwPort><numFree><flags><len><urlPrefix>

Ein HTTP-Gateway antwortet mit dieser Nachricht auf GETGATEWAYSTATUS-Anfragen. Die Antwort enthält die TPID aus der Anfrage, den aktuellen Port des HTTP-Gateways, die Anzahl freier Download-Plätze (4 Bytes), ein Flags-Feld (1 Byte) und ein optionales URL-Präfix. Für das Flags-Feld sind 3 Werte zulässig: 0 signalisiert ausschließliche Unterstützung von HTTP, 1 ausschließliche Unterstützung von HTTPS und 2 die Unterstützung beider Protokolle. Aus diesen Daten erzeugt der Empfänger URLs für Antworten auf HTTP-Anfragen wie in Abschnitt 6.8 beschrieben.

TUNNEL: <id=24><len2><payload>

Die Tunnel-Nachricht wird für sämtliche Übertragungen der Anonymisierungsfunktion verwendet (siehe Kapitel 6.6). Sie kann gesendet werden, ohne dass zuvor ein Handshake stattfand.

7 Kurzevaluierung & Fazit

7.1 Kurzevaluierung

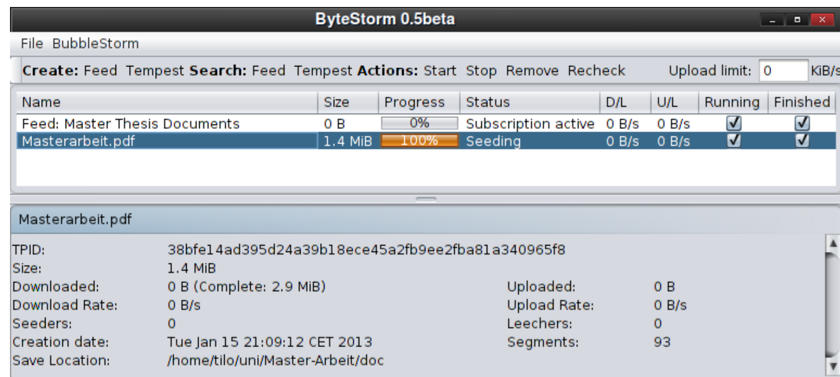

Abbildung 7.1: ByteStorm GUI mit Übersicht über Feeds und Tempests

Um die Konzepte von ByteStorm auf Fehlkonzeptionen und Implementierbarkeit zu untersuchen, wurde ein Prototyp auf der Basis des BitTorrent-Clients aus der Bachelorarbeit [VSB10] entwickelt. Aus dem Java-basierten Client wurden sämtliche Bestandteile des BitTorrent Protokolls entfernt und durch das ByteStorm Protokoll ersetzt. Zudem wurden die Funktionalitäten für das Erzeugen und Abonnieren von Feeds, dem integrierten HTTP-Gateway mit Load Balancing sowie die Such- und Veröffentlichungsfunktionen hinzugefügt bzw. erweitert. Einige Funktionen und Details wurden für den Prototypen jedoch auch weggelassen oder vereinfacht. So sind beispielsweise Übertragungen über anonyme Verbindungen nicht möglich und die Segmentauswahl beim Download folgt nicht der Rarest-First-Strategie, sondern einer einfachen Zufallsstrategie.

Da das Protokoll nicht vollständig implementiert wurde, lassen sich noch keine quantitativen Aussagen zur Performanz der P2P-Datenübertragungen machen. Versuche mit einem kleinen Netz aus einigen wenigen Clients lassen jedoch eine ähnliche Effizienz vermuten, wie die des BitTorrent-Protokolls. Da die Performanz von ByteStorm von einigen Parametern abhängt (z.B. Baumhöhe, Handshake-Parameter, Download-Strategie, Netzwerkbandbreite und -auslastung), ist eine nähere Untersuchung erforderlich, um Werte und Regeln zu finden, die zu einem optimalen Protokollverhalten führen. Die sekundären Funktionen wie Suchfunktion, Feeds und HTTP-Gateway zeigten sich bereits äußerst zuverlässig. Bei Versuchen mit dem Prototyp wurde jede HTTP-Anfrage befriedigt, sofern es im Schwarm des entsprechenden Tempests freie Kapazitäten gab.

7.2 Zusammenfassung

In den vorangegangenen Kapiteln wurde das Protokoll ByteStorm beschrieben, das das Verteilen von Content via Peer-to-Peer Technologien ermöglicht und gleichzeitig Lösungen für viele Einsatzszenarien integriert, für die andernfalls separate Software nötig wäre.

ByteStorm basiert auf Squalls und Tempests, die eng miteinander zusammenhängen. Tempests fassen Content in Form einer Menge von Dateien zusammen. Squalls beschreiben die Details zu einem Tempest. Gleichzeitig sind Squalls der Anker für eine Vertrauenskette, anhand der überprüfbar ist, ob der Content vom erwarteten Urheber stammt. Diese Überprüfung ist für beliebige Teile eines Tempests möglich, bis hinunter zu einzelnen Segmenten. Dadurch sind sämtliche Daten vor Manipulation geschützt, wie eingangs gefordert. Ermöglicht wird dies durch einen quaternären Hashbaum über den Content, dessen Wurzelknoten durch eine digitale Signatur mit dem Squall verbunden ist. Squalls können leicht gefunden werden, da jeder Squall mittels BubbleStorm in einer verteilten Datenbank gespeichert wird, welche von Benutzern anhand zahlreicher Suchkriterien durchsucht werden kann.

Herausgebern und Nutzern von regelmäßig neu veröffentlichtem Content erleichtern Feeds die Verbreitung bzw. das Beziehen des Contents. Herausgeber fügen dafür ihren Content beim Erzeugen eines neuen Tempests zu einem Feed hinzu und die Nutzer, welche den entsprechenden Feed abonniert haben, laden neue Veröffentlichungen automatisch herunter. Die Authentizität aller Feed-Einträge wird sichergestellt, indem die Vertrauenskette nicht erst beim Squall, sondern bereits beim Feed beginnt.

Über HTTP-Gateways können selbst Nutzer, die nicht Teil des ByteStorm-Netzes sind (und ByteStorm eventuell nicht einmal kennen), Dateien aus Tempests herunterladen. Jeder Seeder im Schwarm eines Tempests kann sich selbst als HTTP-Gateway anbieten. Die Gateways im Schwarm kümmern sich selbstständig um Load Balancing, also das Verteilen aller HTTP-Anfragen auf die HTTP-Gateways. Dazu tauschen sich die HTTP-Gateways im Schwarm untereinander über ihre aktuelle Auslastung aus. HTTP-Anfragen können an ein beliebiges Gateway gesendet werden. Kann es die Anfrage aktuell nicht selbst erfüllen, wird der Anfragende an ein anderes Gateway weitergeleitet, das noch freie Kapazitäten besitzt.

Anonymisierte Verbindungen sind auf Basis von Onion Routing möglich. Der die Verbindung herstellende Peer wählt dabei den Pfad des verschlüsselten Tunnels. Der Pfad besteht aus anderen ByteStorm-Knoten. Damit beide Endpunkte einer Verbindung anonym bleiben können, wurden Proxy-Tunnel eingeführt, bei denen ein anderer Peer Verbindungen anstelle des anonymen Peers annimmt und diese durch den Tunnel weiterleitet.

Durch Kombination der integrierten Funktionen lassen sich alle in der Einleitung genannten Szenarien mittels ByteStorm umsetzen. Die integrierte Suchfunktion sowie die Feeds helfen dabei, dass Benutzer ihren gewünschten Content leicht finden und beziehen können, tragen also zur Förderung der Benutzbarkeit von Endprodukten bei.

Anhand der Implementierung eines Prototyps konnte eruiert werden, dass die entwickelten Konzepte in der vorliegenden Form praktisch einsetzbar sind.

7.3 Ausblick

Wie aus den vorhergehenden Kapiteln dieser Arbeit bereits hervorgeht, ist die Performanz von ByteStorm von einigen Protokollparametern abhängig. Um diese zu optimieren sind Messreihen unter unterschiedlichen, praxisnahen Bedingungen nötig. Aufgrund der verwandten Funktionsweise mit BitTorrent ist eine Performanz zu erwarten, die mindestens der des BitTorrent-Protokolls entspricht.

Ein Problem der Anonymisierungsfunktion ergibt sich in dem Falle, dass zwei anonyme Peers miteinander in Verbindung treten wollen. Jeder Peers betreibt in diesem Szenario einen eigenen Tunnel. Beide treffen beim Proxy-Peer aufeinander. Die effektive Pfadlänge ist damit die Summe der Längen der beiden Tunnel. Ein langer Pfad hat jedoch negative Konsequenzen für die Übertragungsbandbreite. Wünschenswerter wäre, wenn im Anonym-Anonym-Fall ein Tunnel aufgebaut werden würde, der kürzer ist, aber den Sicherheitsanforderungen beider Peers entspricht, also die Anonymität von keinem der Peers eingeschränkt wird. Dem entgegen steht jedoch das Designziel, dass es Peers, die miteinander verbunden sind, möglichst schwer gemacht werden soll, herauszufinden, ob der andere Peer ein anonymer Peer ist, um die plausible Abstreitbarkeit aufrecht zu erhalten.

ByteStorm eignet sich sehr gut als Basis für die Entwicklung weiterer Funktionen. Denkbar sind beispielsweise der Ausbau zu einem dezentralen Paket- und Updatemanager ähnlich denen in Linuxdistributionen, einem von Nutzern betriebenen, automatisierten CDN oder die Erweiterung des HTTP-Gateways, sodass das Gateway den gewünschten Tempest nicht bereits kennen muss, um HTTP-Anfragen weiterzuvermitteln.

8 Glossar

Bubble

Ein Bubble ist ein Teilgraph über Knoten eines Bubblestorm-Netzwerks. Zu sendende Daten werden auf allen Knoten eines Bubbles repliziert.

Bubblecast

Den Vorgang der Replikation von Daten in einem Bubble wird als Bubblecast bezeichnet.

Churn

Churn bezeichnet die Tatsache, dass Knoten in einem P2P-Netzwerk zu jedem beliebigen Zeitpunkt das Netzwerk betreten und verlassen können und ist bei der Entwicklung neuer P2P-Verfahren zu berücksichtigen.

Gossip

Gossip (deutsch: Tratsch) ist der Austausch neuester Informationen zwischen Peers eines P2P-Netzwerks. Von anderen Peers erfahrene Neuigkeiten werden an zufällig ausgewählte andere Knoten weiterverbreitet. Kommunikationsinhalt kann beispielsweise der Zustand der umliegenden Knoten sein.

Cover Traffic

Um Nutzdaten bei der Übertragung vor der Identifizierung zu schützen, können sie in einen Strom aus (an und für sich nutzlosen) Daten integriert werden. Auf diese Weise kann ein Angreifer nicht unterscheiden, ob tatsächlich ein Kommunikationsinhalt übermittelt oder nur Rauschen übertragen wurde.

HAVE-Message

Eine HAVE-Message signalisiert anderen Peers in ByteStorm, welche Teile eines Tempests ein Peer bereits fertiggestellt hat. Es wird zwischen zwei Arten von HAVE-Messages unterschieden: HAVE_DOWNLOADED signalisiert, dass die betreffenden Daten heruntergeladen, aber

noch nicht verifiziert wurden. Bei HAVE_VERIFIED wurden die Daten zusätzlich als korrekt verifiziert.

Infohash

Der Infohash identifiziert in BitTorrent ein Torrent eindeutig. Er ist ein SHA1-Hash über bestimmte Teile einer .torrent-Datei.

Leecher

Ein Leecher bezeichnet einen Knoten in einem P2P Content Distribution System, der Daten von anderen Peers herunterlädt und diesen Download noch nicht abgeschlossen hat.

Overlay-Netzwerk

Ein Overlay-Netzwerk ist ein logisches Netzwerk, das auf einem bestehenden Netzwerk (wie dem Internet) aufbaut und eine eigene Topologie besitzt.

Peer-to-Peer

Die direkte Kommunikation zwischen zwei Netzwerkendpunkten ohne einen Umweg über einen zentralen Server wird als Peer-to-Peer, Peer2Peer oder kurz P2P bezeichnet. Jeder Peer ist den anderen gleichgestellt, d.h. jeder Peer übernimmt gleichzeitig die Funktionalitäten von Client und Server.

Seeder

Ein Seeder ist ein Knoten in einem P2P Content Distribution System, der Daten ausschließlich an andere Peers verteilt, der also keine Daten (mehr) herunterlädt.

Source Routing

Beim Source Routing gibt der Sender eines Datenpakets vor, welche Route das Datenpaket durch ein Netzwerk zu nehmen hat. Router, die auf dem Pfad liegen, ist es also nicht gestattet, autonome Entscheidungen über die Weitergabe des Paketes zu treffen.

Squall

In ByteStorm gehört zu jedem Tempest mindestens ein Satz von Metadaten (Name, Beschreibung, Kategorien, Dateiliste, etc.), die durch ein digitales Signaturverfahren an den Tempest gebunden sind. Diese Metadaten heißen Squall und dienen der Suche von Tempests, Peers und

Feeds, informieren über den Inhalt eines Tempests und enthalten ggf. Informationen, wie mit dem fertigen Download umzugehen ist.

Symbol (Coding Theorie)

Ein Symbol ist die kleinste von Kodierverfahren verwendete Einheit. Sie ist das vom Kodierverfahren verwendete Alphabet. Ein oder mehrere Symbole können zu sog. Codeworten zusammengefasst werden. Ein Symbol kann beispielsweise einem Bit entsprechen, also den Alphabet bestehend aus 0 und 1.

Tempest

Ein Tempest fasst in ByteStorm eine oder mehrere Dateien zu einer Einheit zusammen, um diese Dateien zusammen zwischen den Peers zu verteilen. Die Dateien werden durch den Tempest Tree, einem Hashbaum, vor Übertragungsfehlern abgesichert.

TPID

TPID steht für Tempest ID und ist ein Verfahren zur Bestimmung einer eindeutigen, manipulationssicheren Identifikation eines Tempests. Eine TPID wird berechnet, indem der Tiger Hashalgorithmus auf die Konkatenation einer UUID und einem öffentlichen Schlüssel angewendet wird. Beide befinden sich im Squall.

9 Abbildungsverzeichnis

4.1 Dandelion Nachrichtenaustausch beim Download zwischen Clients [DAN07, Seite 2] 31
4.2 Aufbau eines Tokens in STEP [STE06, Seite 2] 38

5.1 Anonymes Routing mit Information Slicing [STO05, Seite 2] 49

6.1 Aufeinandertreffen von Query Bubble und Data Bubble [BBS07, Seite 2] 53
6.2 Veröffentlichung von Torrents im BubbleStorm-Netzwerk [VSB10, Seite 22] 56
6.3 Suche nach Torrents im BubbleStorm-Netzwerk [VSB10, Seite 23] 57
6.4 Beispiel eines Hash Trees 64
6.5 Verifikation von Knoten h_2 (gelb) über Wurzelpfad (rot) mithilfe von zusätzlichen Knoten (grün) 65
6.6 Beispiel eines nicht voll besetzten Tempest Trees 68
6.7 Schalenverschlüsselung beim Onion Routing (Harrison Neal, Lizenz: CC BY-SA)[1] 84
6.8 Vollständige Datensegmente von Peer B 97

7.1 ByteStorm GUI mit Übersicht über Feeds und Tempests 103

10 Algorithmenverzeichnis

6.1 Pseudoalgorithmus für ein typisches HTTP-Gateway 92

11 Literaturverzeichnis

[AEC04] R. L. Collins, J. S. Plank. Assessing the performance of Erasure Codes in the Wide Area. *International Conference on Dependable Systems and Networks (DSN 2005)*, June 2005.
`http://www.cs.utk.edu/~library/TechReports/2004/ut-cs-04-536.pdf`

[BAR06] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, M. Dahlin. BAR Gossip. *Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI)*, November 2006.
`http://www.cs.utexas.edu/~elwong/research/publications/bar-gossip.pdf`

[BBS07] W. W. Terpstra, J. Kangasharju, C. Leng, A. P. Buchmann. BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. *SIGCOMM'07*, 2007.

[BIT07] A. Ramachandran, A. d. Sarma, N. Feamster. BitStore: An Incentive-Compatible Solution for Blocked Downloads in BitTorrent. *In 8th Conference on Electronic Commerce (EC)*, 2007.

[BRU12] C. Leng. BubbleStorm: Replication, Updates, and Consistency in Rendezvous Information Systems. *Dissertation des Fachbereichs Informatik, TU Darmstadt*, August 2012.

[BT03] B. Cohen. Incentives Build Robustness in BitTorrent. 2003.
`http://www.bittorrent.org/bittorrentecon.pdf`

[BTC08] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
`http://www.bitcoin.org/bitcoin.pdf`

[CTA10] P. Manils, A. Chaabane, S. Le Blond, M. A. Kaafar, C. Castelluccia, A. Legout, W. Dabbous. Compromising Tor Anonymity Exploiting P2P Information Leakage. *HotPETs, Berlin*, 2010.

[CUS10] W. W. Terpstra, C. Leng, M. Lehn, A. P. Buchmann. Channel-based Unidirectional Stream Protocol (CUSP). *Proceedings of the IEEE INFOCOM Mini Conference (INFOCOM'10)*, March 2010.
`http://www.dvs.tu-darmstadt.de/publications/pdf/cusp-short.pdf`

[CVL06] C. Gkantsidis, J. Miller, P. Rodriguez. Comprehensive View of a Live Network Coding P2P System. *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC '06)*, 2006.
http://research.microsoft.com/pubs/69452/imc06.pdf

[DAN07] M. Sirivianos, X. Yang, S. Jarecki. Dandelion: Cooperative Content Distribution with Robust Incentives. *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.
http://www.cs.duke.edu/~msirivia/publications/dandelion-netecon.pdf

[DFA98] J. W. Byers, M. Luby, M. Mitzenmacher, A. Rege. Digital Fountain Approach to Reliable Distribution of Bulk Data. *Proceedings of ACM SIGCOMM '98*, 1998.
http://www.cs.rutgers.edu/~rmartin/teaching/fall04/cs552/readings/by98.pdf

[DSB87] R. C. Merkle. A digital signature based on a conventional encryption function. *Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*, 1987.
http://www.cse.msstate.edu/~ramkumar/merkle2.pdf

[FEC02] J. Lacan, L. Lancérica, L. Dairaine. When FEC Speed up Data Access in P2P Networks. *Proceedings of the Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS 2002)*, 2002.
http://dmi.ensica.fr/IMG/pdf/doc-1.pdf

[FLO09] M. E. Dick, E. Pacitti, B. Kemme. Flower-CDN: a hybrid P2P overlay for efficient query processing in CDN. *Proceedings of the 12th ACM International Conference on Extending Database Technology (EDBT)*, 2009.
http://hal.inria.fr/inria-00331231/PDF/RR-6689.pdf

[FLO11] M. E. Dicka, E. Pacittib, R. Akbariniac, B. Kemme. Building a Peer-to-Peer Content Distribution Network with High Performance, Scalability and Robustness. *Information Systems 36, 2 (2011) 222-247*, 2011.
http://hal-lirmm.ccsd.cnrs.fr/docs/00/60/78/98/PDF/2011_-_InfoSys_-_Building_a_Peer-to-Peer_Content_Distribution_Network_with_High_Performance_Scalability_and_Robustness.pdf

[FOU05] D.J.C. MacKay. Fountain codes. *IEE Proceedings Communications, Volume 152, Issue 6*, Dezember 2005.
http://www.eecs.harvard.edu/~michaelm/E210/fountain.pdf

[FRB07] M. Sirivianos, J. H. Park, R. Chen. Free-riding in BitTorrent Networks with the Large View Exploit. *IPTPS '07*, 2007.
http://research.microsoft.com/en-us/um/redmond/events/iptps2007/papers/sirivianosparkchenyang.pdf

[FRC06] T. Locher, P. Moor, S. Schmid, R. Wattenhofer. Free Riding in BitTorrent is Cheap. *5th Workshop on Hot Topics in Networks (HotNets)*, 2006. (BitThief)
http://www.disco.ethz.ch/publications/hotnets06.pdf

[FTP09] A. Sherman, J. Nieh, C. Stein. FairTorrent: Bringing Fairness to Peer-to-Peer Systems. *Proceedings of the 5th international conference on Emerging networking experiments and technologies (CoNEXT '09)*, December 2009.

[FTT06] R. Aringhieri, E. Damiani, S. De Capitani Di Vimercati, S. Paraboschi, P. Samarati. Fuzzy Techniques for Trust and Reputation Management in Anonymous Peer-to-Peer Systems. *Journal of the American Society for Information Science and Technology archive Volume 57 Issue 4*, February 2006.
http://sieci.pjwstk.edu.pl/media/bibl/%5BAringhieri%5D_%5BFuzzy%20Techniques%5D_%5BASIST%5D_%5B2006%5D.pdf

[IBR07] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, A. Venkataramani. Do incentives build robustness in BitTorrent? *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07*, 2007.
http://www.cs.washington.edu/homes/arvind/papers/bittyrant.pdf

[IIE12] D. Carra, P. Michiardi, H. Salah, T. Strufe. On the Impact of Incentives in eMule: Analysis and Measurements of a Popular File-Sharing Application. Submitted to *IEEE JSAC*, 2012.
http://profs.sci.univr.it/~carra/downloads/carra_ICDCS-12_sub.pdf

[IIM07] M. Zghaibeh, K. G. Anagnostakis. On the Impact of P2P Incentive Mechanisms on User Behavior. *Proceedings of the Joint Workshop on The Economics of Networked Systems and Incentive-Based Computing*, 2007.
http://netecon.seas.harvard.edu/NetEcon07/Papers/zghaibeh_07.pdf

[INC10] P. Gasti, A. Merlo, G. Ciaccio, G. Chiola. On the Integrity of Network Coding-based Anonymous P2P File Sharing Networks. *9th IEEE International Symposium on Network Computing and Applications (IEEE NCA10)*, 2010.
http://www.6nelweb.com/bio/papers/netcoding-nca10.pdf

[IPS07] M. Babaioff, J. Chuang, M. Feldman. Incentives in Peer-to-Peer Systems. In N. Nisan, T. Roughgarden, E. Tardos, V. Vazirani, eds., *Algorithmic Game Theory*, Cambridge University Press, 2007.
`http://www.eecs.harvard.edu/cs285/Nisan_Non-printable.pdf`

[ISW12] J. Xu, X. Wang, J. Zhao, A. O. Lim. I-Swifter: Improving chunked network coding for peer-to-peer content distribution. *Peer-to-Peer Networking and Applications, 2012, Volume 5, Number 1*. 2012.
`http://www.springerlink.com/content/r142h070112pqx1u/?MUD=MP`

[KAR03] V. Vishnumurthy, S. Chandrakumar, E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing.*In Proceedings of the Workshop on the Economics of Peer-to-Peer Systems, Berkeley, California*, June 2003.
`http://www.cs.cornell.edu/people/egs/papers/karma.pdf`

[KAR05] F. D. Garcia, J.-H. Hoepman. Off-line Karma: A Decentralized Currency for Peer-to-peer and Grid Applications. *In 3rd Applied Cryptography and Network Security conference (ACNS)*, June 2005.
`http://www.cs.ru.nl/~jhh/publications/karma-full.pdf`

[LTC02] M. Luby. LT Codes. *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.
urlhttp://www.inference.phy.cam.ac.uk/mackay/dfountain/LT.pdf

[MPA07] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, E. Rachlin. Making P2P Accountable without Losing Privacy. *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, 2007.
`http://www.cs.brown.edu/~mchase/papers/bittorrent.pdf`

[MPR07] Y. Zhu, Y. Hu. Making Peer-to-Peer Anonymous Routing Resilient to Failures. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007.
`http://fac-staff.seattleu.edu/zhuy/web/papers/zhu_ipdps07_final.pdf`

[NCD07] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. O. Wainwright, K. Ramchandran. Network Coding for Distributed Storage Systems. *26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, May 2007.
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.6892&rep=rep1&type=pdf`

[NWC05] C. Gkantsidis, P. R. Rodriguez. Network Coding for Large Scale Content Distribution. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2005.
http://www.eecs.harvard.edu/~michaelm/CS222/contentdist.pdf

[OHR08] M. Piatek, T. Isdal, A. Krishnamurthy, T. Anderson. One hop Reputations for Peer to Peer File Sharing Workloads. *Proceedings of the 5th USENIX Symposium on Networked Systems Design & Implementation (NSDI '08)*, April 2008.
http://www.cs.washington.edu/homes/arvind/papers/onehop.pdf

[PAC00] C. Shields and B. N. Levine. A Protocol for Anonymous Communication Over the Internet. *In ACM Conference on Computer and Communications Security*, 2000.

[PAP11] S. Kaune. Performance and Availability in Peer-to-Peer Content Distribution Systems: A Case for a Multilateral Incentive Approach. Technische Universität Darmstadt, 2011. (URN: urn:nbn:de:tuda-tuprints-24924)
http://tuprints.ulb.tu-darmstadt.de/2492

[PPA03] B. Yang, H. Garcia-Molina. PPay: Micropayments for Peer-to-Peer Systems. *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, 2003.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7457&rep=rep1&type=pdf

[PPP10] T. Isdal, M. Piatek, A. Krishnamurthy, T. Anderson. Privacy-preserving P2P data sharing with OneSwarm. *SIGCOMM 2010*, September 2010.
http://www.michaelpiatek.com/papers/oneswarm_SIGCOMM.pdf

[RAP06] A. Shokrollahi. Raptor Codes. *IEEE/ACM Transactions on Networking (TON), Volume 14, Issue SI*, Juni 2006.
http://www.inference.phy.cam.ac.uk/mackay/dfountain/RaptorPaper.pdf

[SAM03] L. P. Cox, B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *Proceedings of the 19th ACM SOSP*, October 2003.
http://www.cs.rochester.edu/meetings/sosp2003/papers/p135-cox.pdf

[SBP10] R. L. Xia, J. K. Muppala. A Survey of BitTorrent Performance. *IEEE Communications Surveys and Tutorials, Vol. 12, No. 2*, 2010.
http://repository.ust.hk/dspace/bitstream/1783.1/6529/1/SurveyofBitTorrent.pdf

[SCD07] F. Zhao, T. Kalker, M. Médard, K. J. Han. Signatures for Content Distribution with Network Coding. *Information Theory 2007 (ISIT 2007)*, June 2007.

[SFS05] T. Chothia, Konstantinos Chatzikokolakis. A Survey of Anonymous Peer-to-Peer File-Sharing. *IFIP International Symposium on Network-Centric Ubiquitous Systems (NCUS 2005)*, 2005.
http://www.lix.polytechnique.fr/~tomc/Papers/AnonyP2PSurvey.pdf

[SOC12] Z. Wang, C. Wu, L. Sun, S. Yang. Strategies of Collaboration in Multi-Swarm Peer-to-Peer Content Distribution.
textitTsinghua Science and Technology Vol. 17. February 2012.
http://media.cs.tsinghua.edu.cn/~wangzhi/zhi-tsinghua-science-technology.pdf

[STE06] I. Martinovic, C. Leng, F. A. Zdarsky, A. Mauthe, R. Steinmetz, J. B. Schmitt. Self-protection in P2P Networks: Choosing the Right Neighbourhood. *International Workshop on Self-Organizing Systems (IWSOS)*, September 2006.

[STO05] S. Katti, D. Katabi, K. Puchala. Slicing the Onion: Anonymous Routing without PKI. *4th ACM Workshop on Hot Topics in Networks (HotNets)*, November 2005.
http://dspace.mit.edu/bitstream/handle/1721.1/30564/MIT-CSAIL-TR-2005-053.pdf

[SWI08] J. Xu, J. Zhao, X. Wang, X. Xue. Swifter: Chunked Network Coding for Peer-to-Peer Content Distribution. *Proceedings of IEEE International Conference on Communications (ICC 2008)*, 2008.
ftp://lenst.det.unifi.it/pub/LenLar/proceedings/2008/icc08/DATA/S11S02P03.PDF

[TAR02] M. J. Freedman, R. Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. *Proceedings of the 9th ACM conference on Computer and communications security (CCS'02)*, 2002.
http://pdos.csail.mit.edu/tarzan/docs/tarzan-iptps.ps

[THF96] R. Anderson, E. Biham. Tiger: A Fast New Hash Function. *Fast Software Encryption, Third International Workshop Proceedings*, 1996.
http://www.cs.technion.ac.il/~biham/Reports/Tiger/

[TRB07] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, H. J. Sips1. TRIBLER: a social-based peer-to-peer system. Delft University of Technology, Vrije Universiteit, 2007
`https://www.tribler.org/attachment/wiki/File/CPE-Tribler-final.pdf?format=raw` (Collaborative Downloader)

[UCP06] D. Stutzbach, R. Rejaie. Understanding churn in peer-to-peer networks. *Proceedings of the 6th ACM Conference on Internet measurement (IMC'06).*, 2006.

[UTB10] D. Marks, F. Tschorsch, B. Scheuermann. Unleashing Tor, BitTorrent & Co.: How to Relieve TCP Deficiencies in Overlays (Extended Version). *Proceedings of the 35th IEEE Conference on Local Computer Networks (LCN 2010)*, 2010.
`http://www.cn.uni-duesseldorf.de/publications/library/Marks2010b.pdf`

[VSB10] T. Eckert. Verteilte Suche für BitTorrent-Netzwerke. Technische Universität Darmstadt, 2010.
`http://www.dvs.tu-darmstadt.de/publications/BScs/Eckert-Tilo-BSc.pdf`