Verteilte Systeme

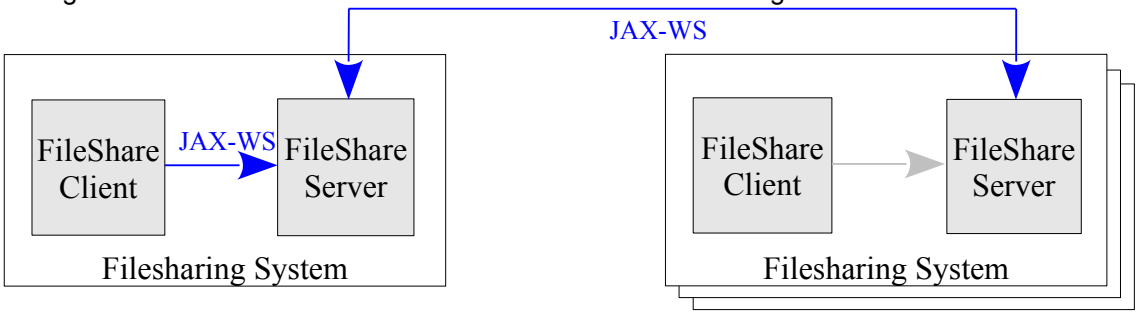
Belegarbeit Wintersemester 2012/13

Bearbeitungszeit 3 Wochen

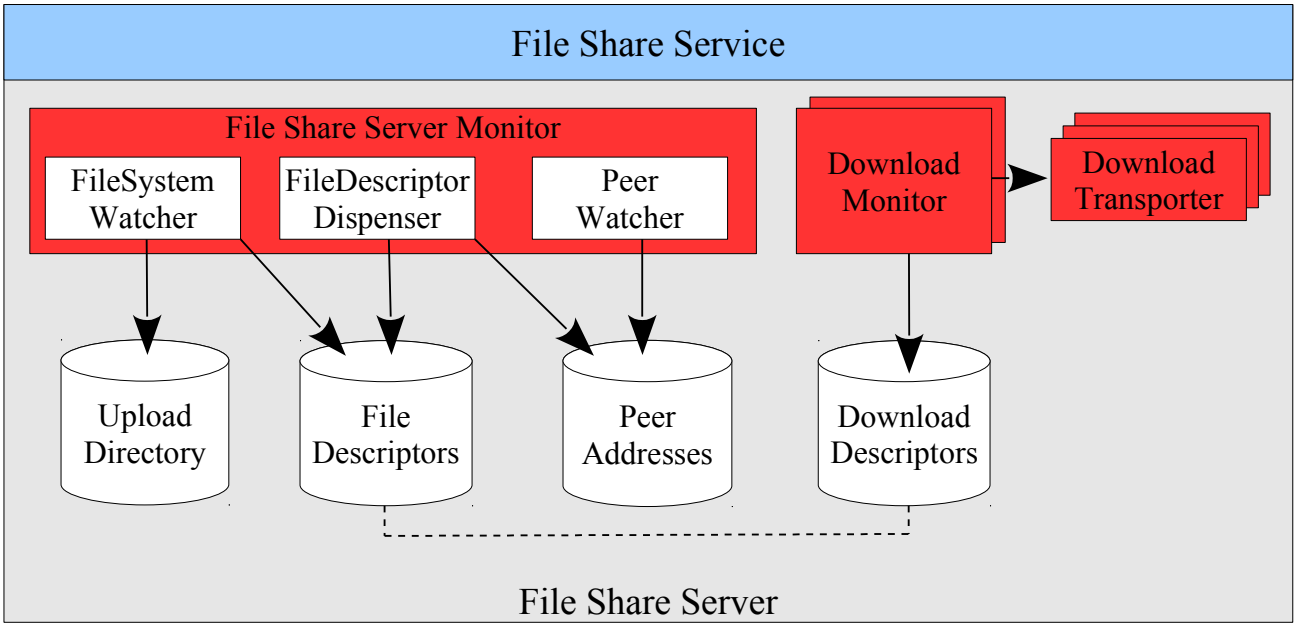
Grundlage: File Share Server & Client Design

Auf Basis einer verteilten Hashtabelle sowie des k-Bucket Auswahlverfahrens soll nun ein Filesharing-System implementiert werden. Der Grundgedanke dabei ist dass erstens alle Teilnehmer einen großen Teil der anderen Teilnehmer kennen, als auch dass jeder Teilnehmer auf seinem System mittels einer eine verteilten Hashtabelle Meta-Information über einige der zur Verfügung stehen Dateien beheimatet.

Das System ist dabei als Peer-to-Peer System ausgelegt: Auf jedem teilnehmenden System gibt es einen **`FileShareServer`**, der Metadaten über das lokale Dateisystem, sowie über nicht-lokale Dateien verwaltet (der lokale Teil einer verteilten Hashtabelle). Zur Kommunikation mit diesem lokalen Server dient ein Client, der alle Anfragen an den lokalen Server weiterreicht. Die Kommunikation geschieht dabei mittels JAX-WS:



Jeder `FileShareServer` implementiert daher ein Service-Interface **`FileShareService`**, welches sowohl Methoden zum Aufruf innerhalb der gleichen Betriebssystem-Instanz (Client->Server), als auch zum Aufruf über Betriebssystem-Instanzen hinweg (Server->Server) beinhaltet. Zudem pflegt jeder `FileShareServer` noch eine Reihe von Daten, sowie Monitor-Threads welche diese Daten kontinuierlich auf Veränderungen beobachten:



Die Peer-Adressen bestehen aus reinen Socket-Adressen. Dies ist ausreichend da nur über JAX-WS

kommuniziert wird, und bei allen `FileShareServer`-Instanzen das Protokoll und der Service-Pfad fest vorgegeben sind. Zudem wird eine verteilte Hashtabelle implementiert welche als Grundlage die Socket-Adressen der `FileShareServer` (der „Peers“) benötigt.

Jede Server-Instanz stellt dabei zum einen einen Web-Service bereit, und dient zum anderen als Runnable eines Monitor-Threads zur fortlaufenden Beobachtung der eigenen Ressourcen (FileShareServer-Monitor). Letzterer wird automatisch beim Starten des Servers bei einem Executor-Service für eine periodische Ausführung angemeldet, und bündelt mehrere komplexe Teilaufgaben zur Erleichterung des Debuggings.

Ein Teil des Monitor-Threads ist ein **`PeerWatcher`**. Dieser hat die Aufgabe ständig andere aktive `FileShareServer` zu kontaktieren und mit diesen die Peer-Adressen auszutauschen. Damit ist gewährleistet dass ein neu gestarteter `FileShareServer` relativ schnell an einen Großteil der gerade aktiven Socket-Adressen gelangt, solange er nur beim Bootstrap einen `FileShareServer` kennenlernt der relativ lange im Netz aktiv war. Dasselbe Verfahren wird z.B. beim Kademlia-Netz eingesetzt.

Die *Upload-Verzeichnisse* werden durch einen **FileSystemWatcher** ständig auf Veränderung beobachtet. Jedes mal wenn sich eine Datei ändert oder neu hinzukommt wird aus ihrem Inhalt ein Content-Hash berechnet, welcher letztlich aus den Bits 1-144 eines *SHA1*-Hashes besteht; das erste Bit wird dabei ignoriert damit der Content-Hash stets positiv ist, und die Länge ist auf 144bit begrenzt damit Vergleiche mit Socket-Adressen ohne Padding durchgeführt werden können. Dieser Content-Hash wird zusammen mit Meta-Information wie Datei-Länge, lokale Pfade, Dateinamen, und Quellen in einer **`FileDescriptor`**-Instanz modelliert. Beachtet dabei dass lokale Dateien mehrfach im lokalen Datei-System vorkommen, und dass Dateien generell lokal, remote, oder lokal&remote vorliegen können. Die Menge aller `FileDescriptor`-Instanzen über alle aktiven `FileShareServer` hinweg konstituiert letztlich die verteilte Hashtabelle, denn in ihr werden die File-Hashes auf die Quellinformation abgebildet.

Soll ein Download initiiert werden, so wird zuerst eine Kriterien-basierte Breiten-Suche (Query) nach passenden Dateien durchgeführt. Kriterien sind dabei z.B. passende Namensfragmente, Dateiendung, Größe, oder Änderungszeitpunkt. Dieser Query wird dabei im Prinzip an alle bekannten Peers gerichtet, jedoch maximal bis zu einem Timeout; sie resultiert in einer Menge von `FileDescriptor`-Instanzen. Da bei einer großer Zahl von bekannten Peers nicht mehr alle Peers bis zum Timeout kontaktiert werden können, kann diese Suche nur eine begrenzte Anzahl an Quellen für die Datei liefern!

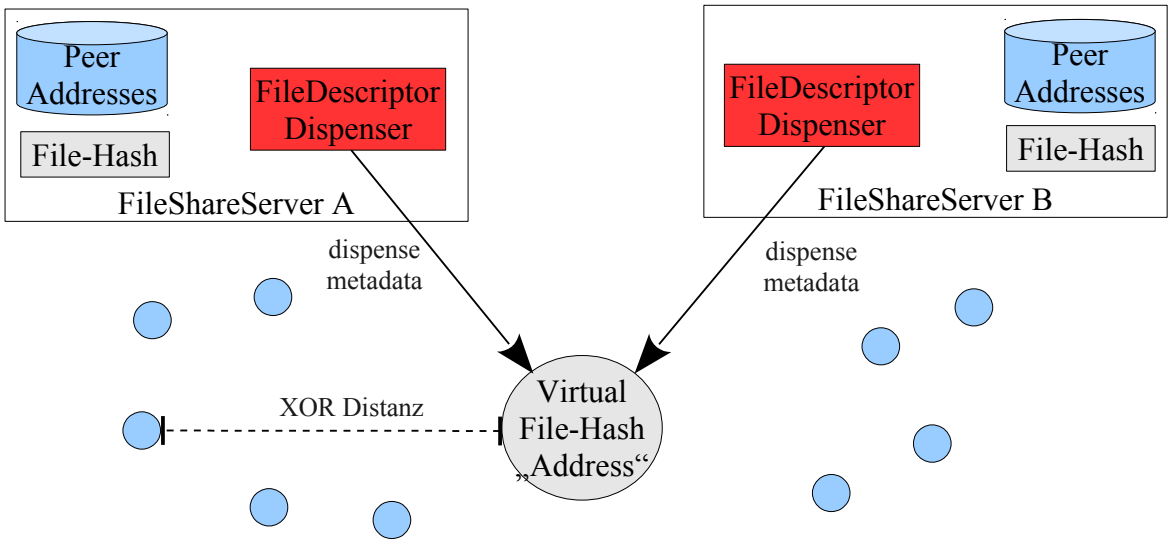
Hat sich ein Benutzer für eine Datei zum Download entschieden, ist im Prinzip nur noch der Content-Hash (die Quasi-Identität der Datei) sowie die Datei-Länge für den weiteren Verlauf relevant. Es wird für die herunter zu ladende Datei ein `DownloadDescriptor` erzeugt und ein **Download-Monitor** gestartet, welcher wiederum je einen **Transporter-Thread** für jede verfügbare Quelle startet. Die Datei wird dabei in Chunks fester Größe (1KB) heruntergeladen, und Quellen werden so lange verwendet wie sie verfügbar sind. Die Zieldatei wird beim Starten eines Downloads sofort angelegt, und um einen Download-Header erweitert welcher den Status der einzelnen Chunks, den Content-Hash der Datei, sowie die Datei-Länge umfasst. Beim Abschluss eines Downloads wird dieser Header einfach entfernt, wodurch die Zieldatei ihre endgültige Länge erhält.

Der **`FileDescriptorDispenser`** schließlich hat die Aufgabe immer wieder Peers über lokal bekannte Dateien zu informieren, sowie gleichzeitig von diesen erweiterte/aktualisierte Quellinformationen über diese Dateien zu erhalten (was unter Anderem eigenen Downloads zugute kommt). Dazu werden für jeden lokal bekannten `FileDescriptor` die Peers nach ihrer Distanz zum Content-Hash der Datei gefiltert, und die übrig bleibenden Peers für den Informationsaustausch kontaktiert.

Der `FileDescriptorDispenser` vereinigt damit die Aufgaben der Quellinformatons-Verteilung und der Quell-Suche innerhalb der verteilten Hashtabelle. Dies hat den Vorteil dass nicht nur Quellinformation zu aktuellen Downloads beständig aktiv gesucht wird, sondern auch externe Quellen zu lokal verfügbaren Dateien, sowie solchen Dateien deren Content-Hash nahe an der eigenen Socket-Adresse liegen. Die statische Methode `SocketAddressFilter.newFileHashBasedFilter(int, int)` liefert dabei die Implementierung des für das k-Bucket Filterverfahren benötigten Filters auf Basis eines BucketSet.

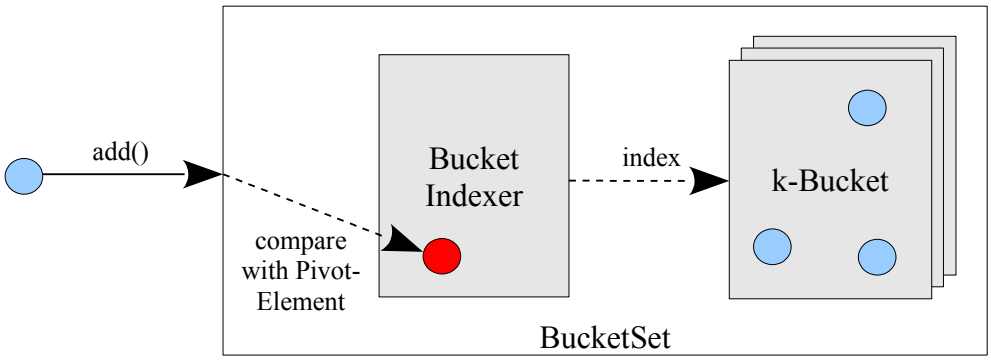
Beim Filtern selbst wird ein Content-Hash (eine 144-Bit lange Ganzzahl) einer Datei als IPv6 Socket-Adresse (eine 128-Bit lange Ganzzahl als IP-Adresse plus 16 Bit Port) interpretiert, und diese zum Pivot-Element eines Bucket-Indexer gemacht. Dieser Indexer verteilt alle bekannten Peer-Adressen eines FileShare-Servers anhand ihrer binären Distanz zum Pivotelement (dem Content-Hash) auf die k-Buckets eines BucketSets, womit bevorzugt dem Content-Hash binär „nahe“ IP-Adressen in das BucketSet gelangen (da

die übrigen wegen Platzmangel in überfüllten k-Buckets nicht aufgenommen werden). Alle ins BucketSet (das Filter-Ergebnis) gelangten Peer-Adressen werden dann kontaktiert.



Grundlage: k-Buckets

Die Klasse **`BucketSet`** implementiert eine k-Bucket Menge mit einstellbarer Bucket-Maximalgröße (k) sowie Bucket-Anzahl, welche durch Implementierung einer Instanz von `BucketSet.BucketIndexer` festgelegt werden. Der Indizierer beinhaltet ein Pivot-Element und entscheidet durch Vergleich mit diesem in welchen k-Bucket jedes neu hinzuzufügende Element wandert. Jeder k-Bucket kann dabei maximal k Elemente aufnehmen – daher der Name. Ein BucketSet wiederum kann maximal k*N Elemente aufnehmen, und wird primär zum Filtern von Socketadress-Mengen verwendet.



Die Idee bei der Elementauswahl durch k-Buckets ist nun den Indexer so zu wählen dass „wertvolle“ Elemente in wenig frequentierte k-Buckets wandern, und damit für diese dort meistens Platz ist. Weniger „wertvolle“ Elemente dagegen werden in höher frequentierte k-Buckets indiziert, so dass nur wenige (k) von ihnen dort Platz finden werden, und die anderen verloren gehen. Sobald versucht wurde alle zu filternden Elemente einzufügen beinhaltet das BucketSet dann statistisch eine relativ hohe Anzahl an „wertvollen“ Elementen, aber auch eine gewisse Zahl von weniger „wertvollen“.

Aufgabe 1 (5 Punkte): FileShareService Interface & Halo

Das Interface **`FileShareService`** soll als Service-Interface eines „bottom-up“ generierten JAX-WS Web-Service dienen. Dazu sind die Service-Methoden des Interfaces, sowie die Halo-Klassen **`DownloadDescriptor`** und **`FileDescriptor`** geeignet zu annotieren:

- Parameter sollen in der generierten WSDL sprechende Namen erhalten
- Pfade sollen in der WSDL als Strings erscheinen
- Felder der Halo-Klassen sollen wenn möglich platzsparend als Attribute übertragen werden (außer ihr Datentyp stellt z.B. eine Menge dar)
- Transiente Felder sollen wie immer nicht gemarshaled werden, und natürlich auch nicht in der WSDL erscheinen

Beachtet dass die Klasse **`de.sb.javase.io.SocketAddress`** aus `javase-util.jar` anstelle von `java.net.InetSocketAddress` verwendet wird, da erstere es erlaubt auch IPv4-Adressen wie IPv6-Adressen zu behandeln, und zudem die notwendigen Annotationen beinhaltet um mittels JAX-B gemarshaled werden zu können. Das benötigte JAR-File findet sich wie immer unter `/share/lehrende/Baumeister/Verteilte Systeme/lib/java-se/javase-util.jar`. Eine Kopie dieses JAR-Files muss im Class-Path Eures Projektes referenziert werden.

Beachtet auch dass die `FileShareServer-Instanzen` keinen netzbasierten Adress-Resolver verwenden um herauszubekommen unter welcher IP-Adresse sie im Internet sichtbar sind; ohne diese Funktionalität kann das FileShare-Netzwerk daher nur innerhalb eines Intranets/Providers betrieben werden.

Annotiert zudem die Klasse `FileShareServer` so dass `FileShareService` von ihr als Service-Interface genutzt wird.

Aufgabe 2 (10 Punkte): FileShareServer

Implementiert die Methode **`FileShareServer.close()`** indem ihr ALLE schließbaren Ressourcen der Instanz schließt. Eure Implementierung muss dabei garantieren dass auch im Fehlerfall wenigstens versucht wird jede dieser Ressourcen zu schließen.

Startet sodann eine „lokale“ `FileShareServer`-Instanz mit folgenden Argumenten:

- Log-Level (INFO, FINE, FINER, FINEST, etc), dient zur Einstellung der „Gesprächigkeit“ des Server-Logs
- Service-Port
- Upload-Directory
- optional weitere Bootstrapping-Socketadressen von Peers im Format `host:port`

Verwendet den vorhandenen **`FileShareClient`** sowie einen eigenes Testprogramm (wird nicht bewertet) um sicherzustellen dass beim Aufruf der Service-Methoden die Parameter und Resultate korrekt übertragen werden. Für alle weiteren Tests solltet ihr einen Setup mit einem „lokalen“ und zwei „remote“ Servern verwenden, die jedoch alle auf derselben Maschine laufen. Achtet dabei darauf dass jeder der Server ein separates Upload-Directory bekommt, in dem einige Dateien zum Upload bereitstehen. Durch Bootstrapping mit einer oder mehrerer Peer-Adressen sollte sich dann das wissen um die aktiven Peers innerhalb einer Minute auf alle Peers verteilen.

Implementiert die Methode **`FileShareServer.queryFileDescriptors(...)`** so dass sie bei potentiell jedem bekannten Peer einschließlich des Server's selbst die korrespondierende Nachricht `FileShareServer.getFileDescriptors(...)` aufruft, und die Ergebnisse konsolidiert zurückgegeben werden. Beachtet dabei folgendes:

- Da die Menge der Peers relativ groß werden kann, müssen die Web-Service Anfragen an diese möglichst parallel ausgeführt werden. Spezifiziert jede Anfrage an einen Peer daher als Future welches ein Array von File-Descriptoren als Ergebnis liefert; für die Peer-Adressen ist der zuletzt berechnete Snapshot der Peer-Adress Menge ausreichend, diese kann vom PeerWatcher abgefragt werden und hat den Vorteil

dass beim Iterieren die Peer-Adress Menge nicht synchronisiert werden muss. Die zur Konstruktion eines Peer-Proxy's benötigte WSDL-URL erhaltet ihr dann mittels der statischen Methode `FileShareResources.serviceWsdlLocator(SocketAddress)`.

- Verwendet den Task-Scheduler des Servers um die Futures zur Ausführung zu registrieren. Nachdem alle Peer-Anfragen registriert sind, berechnet die Teil-Antwort des eigenen Servers und wartet dann bis entweder der Timeout erreicht wird, oder alle Futures fertig sind; dies muss hier mittels geeignetem Polling geschehen, und die dadurch notwendige Latenz soll maximal 10ms betragen! Falls der als Parameter spezifizierte Timeout überschritten wird können die Futures mittels ihrer Methode `cancel(false)` aus der Queue des Task-Schedulers entfernt werden; dies hat keine Wirkung auf bereits fertig berechnete oder gerade laufende Exemplare.
- Resynchronisiert dann gegen jedes Future, und integriert wenn möglich das Future-Teilergebnis in das Gesamtergebnis der Methode. Jeder Content-Hash darf im Gesamtergebnis in maximal einer `FileDescriptor`-Instanz auftauchen; sollte der gleiche Content-Hash von mehreren Peers zurückgeliefert werden, sind die Mengen der Source-Adressen sowie die Dateinamen in der bereits vorhandenen `FileDescriptor`-Instanz zu vereinigen, welche dann Teil der Ergebnismenge wird.

Implementiert die Methode **`FileShareServer.getFileContent(...)`** so dass die mittels Content-Hash adressierte Datei geöffnet, und ein Chunk mit der angegebenen Länge und dem gegebenen Offset zurückgeliefert wird; die Datei ist danach wieder zu schließen. Die Abbildung von Content-Hash auf den lokalen Dateipfad findet sich dabei in den File-Descriptoren. Sollte dort kein passender Eintrag vorhanden sein, oder die Datei lokal nicht verfügbar sein, werft eine `NoSuchFileException`. Falls die Datei dagegen lokal mehrfach vorhanden ist, wählt einfach eine Ausgabe zum Öffnen aus. Ist die Datei zu klein um zum gegebenen Offset zu springen, werft eine `EOFException`. Ist die gegebene Länge dagegen zu groß, so stutzt das Ergebnis auf die Anzahl der bis EOF verfügbaren Bytes.

Implementiert die Methode **`FileShareServer.addFileDescriptor(FileDescriptor)`**. Dort soll der gegebene File-Descriptor zur fileDescriptors-Map des FileSystemWatcher (`this.fileSystemWatcher.getFileDescriptors()`) hinzugefügt werden. Falls dort noch kein Eintrag unter dem Content-Hash (BigInteger) des File-Descriptors exisitiert soll der File-Descriptor einfach in der Map unter seinem Content-Hash registriert werden. Andernfalls sollen nur die Mengen der Source-Adressen sowie die Dateinamen des bestehenden Descriptors mit den entsprechenden Mengen des gegebenen Descriptors erweitert werden. Der bestehende bzw. neue File-Descriptor soll dann zurückgegeben werden damit auch der anfragende Peer seine Descriptor-Information auf den gemeinsamen Stand bringen kann. Achtet bei der Implementierung darauf dass diese Map durch andere Service-Methoden des Servers modifiziert werden kann!

Implementiert die Methode **`FileShareServer.getDownloadDescriptors()`**. Dort soll der Inhalt der assoziierten Objekte („values“) der `downloadDescriptors`-Map des Servers als Array zurückgegeben werden. Nachdem das Ergebnis ermittelt wurde sollen zudem alle geschlossenen Download-Descriptoren aus der Map entfernt werden, und das Ergebnis dann zurückgegeben werden. Achtet bei der Implementierung darauf dass diese Map durch andere Service-Methoden des Servers modifiziert werden kann!

Aufgabe 3 (5 Punkte): SocketAddressFilter

Implementiert die Methode **SocketAddressFilter.newTimestampBasedFilter(int)** so dass ein Filter (d.h. eine Instanz einer Subklasse von `SocketAddressFilter`) zurückgeliefert wird, deren Methode **`filter(SocketAddress[] peerAddresses, Long timestamp)`** folgende Operationen ausführt:

- Berechnet für jede der gegebenen Peer-Adressen die „Distanz“ zum gegebenen Zeitstempel; da die Zeitstempel dabei 64bit-Zahlen sind sollen nur die Bytes 8-16 der binären Socket-Adressen (erhaltet ihr mittels `SocketAddress.toBinarySocketAddress(ConversionMode.ForceIP6)`) Verwendung finden. Verwandelt zum Vergleich den Zeitstempel und den Ausschnitt des Byte-Arrays jeweils in eine BitInteger-Instanz. Die Distanz könnte ihr dann entweder in XOR-Metrik (a XOR b) oder in klassischer Metrik mittels Betrag(a – b) berechnen.
- Gebt vom Ergebnis maximal „capacity“ Socket-Adressen zurück die die kleinste Distanz zum gegebenen Zeitstempel aufweisen, z.B. mittels eines TreeSet zum Sortieren der Socket-Adressen nach Distanz.

Der neue Filter sollte dazu dienen dass der Peer-Watcher gezielter Peers zur Aktualisierung der eigenen Peer-Menge ansprechen kann. Da die Peer-Watcher aller `FileShareServer` die Peers dazu nach denselben Kriterien (i.e. Mittels des gleichen Filters) auswählen, und gleichzeitig jeder seine eigene Socket-Adresse bei der Anfrage auf dem angesprochenen Peer hinterlässt, entsteht so ein Netzwerk bei dem einige Peers sehr schnell alle aktiven Peers kennen lernen, und dieses Wissen dann an alle Peers weiter geben.

Aufgabe 4 (10 Punkte): Content-Multicast

Neue Downloads (also solche für die noch keine Chunks empfangen wurden) sollen die Möglichkeit erhalten ihre Chunks (oder einen guten Teil davon) mittels UDP-Multicast zu erhalten. Dazu muss das Service-Interface der FileShareServer um eine Methode getMulticastAddress(BigInteger contentHash) → SocketAddress erweitert werden. Diese Methode soll eine NoSuchFileException werfen wenn der Content-Hash lokal nicht bekannt ist, oder keine passende lokale Datei zur Verfügung steht.

Danach soll der Server nach einer Verzögerung von 5 Sekunden anfangen in einem neuen Thread den gesamten Inhalt der Datei mittels UDP zu übertragen; falls während der Übertragung eine weitere `getMulticastAddress()`-Anfragen nach der gleichen Datei eingeht, soll die gleiche Multicast-Adresse zurückgegeben werden; der anfordernde Peer klinkt sich dann in eine laufende Übertragung ein. Die Idee ist dabei dass der Multicast spezifisch für das zu übertragende Datei sein soll. Daher muss vom Sender per Zufall eine Socket-Adresse aus dem Bereich möglicher IPv4 oder IPv6 Multicast-Adressen ausgewählt werden, welche dann spezifisch für die Übertragung genutzt wird; verwendet dabei stets Port 8010. Da Kollisionen nicht ausgeschlossen werden können soll mit jedes UDP-Paket folgende Informationen beinhalten - verwendet dabei einen `DataInputStream` bzw. `DataOutputStream` zum Lesen und Schreiben:

- Content-Hash
- Chunk-Index
- Chunk-Inhalt (immer 1024 Bytes, außer beim letzten Chunk der kleiner sein darf)

Pakete die nicht den richtigen Content-Hash aufweisen (oder zu klein sind um einen Content-Hash, Chunk-Index sowie mindestens ein Byte Chunk-Inhalt zu beinhalten) können so von den horchenden Peers leicht ignoriert werden. Sollte beim Lesen der Datei in Fehler passieren, so soll die Übertragung einfach abgebrochen, und die geöffneten Ressourcen sauber geschlossen werden. Beachtet dass die Chunk-Größe von 1KB gewählt wurde damit alle

Der Empfangs-Teil des Verfahrens ist in der Methode `DownloadMonitor.run()` als Erweiterung zu implementieren. Dazu soll zuerst überprüft werden ob noch alle Chunks im Zustand AQUIRABLE sind, nur dann soll das Multicast-Verfahren zum Einsatz kommen; andernfalls ist mit dem bestehenden Verfahren fortzufahren. In ersterem Fall soll dann aus allen verfügbaren Quellen („sources“-Feld) ein Peer zufällig ausgewählt, und diesem mittels JAX-WS die neue Nachricht `getMulticastAddress(BigInteger)` gesendet werden. Falls diese ohne Exception beantwortet wird soll so lange auf Pakete gehorcht werden bis entweder eine 10-sekündige Pause eintritt, oder ein Paket mit dem für diese Datei höchstmöglichen Chunk-Index (`chunkCount(contentLength) - 1`) eintrifft. Bis dahin sollen die eintreffenden Pakete empfangen, auf korrekten Content-Hash untersucht, die Chunks mittels `putChunk()` in die Ziel-Datei geschrieben, und der Chunk-Status mittels `putChunkStatus()` von AQUIRABLE auf COMITTED gebucht werden. Tritt der Abbruchfall ein, so sollen die mit dem Multicast in Zusammenhang stehenden Ressourcen sauber geschlossen, und die run()-Methode verlassen werden; beim nächsten Aufruf sollte dann das bestehende Web-Service basierte Übertragungsverfahren genutzt werden um noch fehlende Chunks zu ergänzen, und den Download dann abzuschließen.

Bearbeitungsmodalitäten:

Gruppenstärke: 3-4, Abweichungen davon nur nach Rücksprache mit mir

Bearbeitungszeit: 3 Wochen bis Sonntag 24.02.2013, 24:00 Uhr (einige Stunden später ist auch noch ok)

Hilfsmittel: Alle außer Code anderer Gruppen ; dies umfasst auch die Weitergabe des eigenen Codes und das Einsehen von Code anderer Gruppen. Code anderer Gruppen ist schlicht in jeder Form tabu, und alle Gruppenmitglieder haften dafür dass diese Regel nicht verletzt wird!

Ports: 8001 - 8010 im Rechnerraum, sonst frei wählbar

Kommentare: Keine außer wenn explizit verlangt

Hotline: Bei Verständnisfragen täglich zwischen 19:00 und 22:00 unter 0174/3345975 oder 030/859 729 44. Bitte keine Fragen per Email!

Abgabeformat: 1 JAR-File mit dem Quellcode (!) der Lösung, inklusive der ausgegebene Klassen innerhalb eines Packages. Zusätzlich muss eine Datei „**`/META_INF/authors.txt`**“ vorhanden sein welche Eure Matrikelnummern, Email- Adressen und Namen beinhaltet (möglichst in einer Zeile pro Gruppenmitglied). Bitte beides unter keinen Umständen vergessen!

Abgabe: Übergabe per Email an sascha.baumeister@gmail.com

Bewertung: 30 Punkte maximal bei vollständiger und fehlerfreier Implementierung. Abzüge betragen 0.5 Punkte für kleinere Fehler/Auslassungen, 1 Punkt für „normal“ schwere Fehler/Auslassungen oder grob umständliche Lösungsansätze, sowie 2 Punkte für grobe Fehler. Themaverfehlungen werden individuell geahndet, typischerweise mit 0 Punkten für die betroffene Aufgabe; dies solltet ihr daher unter allen Umständen vermeiden.