Projektarbeit: HBase auf DXRAM

Jochen Peters
Heinrich-Heine-Universität Düsseldorf
Arbeitsgruppe Betriebssysteme betreut durch Prof. Dr. Michael Schöttner

20. November 2018

Auf Github: no-go.github.io/HadoopDxramFS

1 Machbarkeit von HBase auf DXRAM

Ziel dieser Projektarbeit war es, heraus zu finden, ob sich die NO-SQL Datenbank HBase mit dem verteilten Key-Value-Storage DXRAM verbinden lässt. HBase nutzt Hadoop und dessen Dateisystem HDFS zur Speicherung, welches durch DXRAM abgelöst werden sollte. Ferner gab es als Ziel eine konkrete DXRAM-Anwendung zu entwickeln, die sich problemlos in populäre verteilte Entwicklungen auf Basis von HBase oder Hadoop einbinden lässt. Dies sollte Performance-Vorteile durch DXRAM verdeutlichen, welche durch Verwendung von Infiniband und einer Speicherung im Arbeitsspeicher (an der Java Heap-Verwaltung vorbei) zu erwarten sind. Die Popularität von DXRAM sollte damit gefördert werden. Zur Umsetzung wurden drei mögliche Konzepte ins Auge gefasst: 1) Nachbau von HBase mit DXRAM auf der Basis der Thrift Schnittstelle, die HBase einem Client zur Verfügung stellt. 2) Mit `libfuse` DXRAM unter Linux mount-fähig machen, und HBase mit Hadoop im lokalen Dateisystem laufen lassen. 3) Das Scheme-Konzept von Hadoop beim Dateizugriff nutzen, um bei Zugriffen auf `dxram://` anstelle von `hdfs://` eine Hadoop-Kompatible Dateisystem-Implementierung anbieten zu können. Die Entscheidung fiel auf das dritte Konzept, da es weder den Nachbau von HBase, noch Performance-Verluste durch `libfuse` zur Folge hatte. Nähere Details zur Software, Entscheidungsfindung, Konzept zur Umsetzung mit Details zu besonderen Herausforderungen sind in den kommenden Kapiteln beschrieben. Im letzten Kapitel findet man einen Vergleich zu anderen Projekten, den aktuellen Stand dieses

Abbildung 1: Logo Vorschlag für DxramFs

Projekts sowie Überlegungen zur zukünftigen Weiterentwicklung.

Vorab: Die Implementierung ist nicht ganz vollständig und wurde für erste Tests zur Machbarkeit sehr pragmatisch zusammen gestellt. Daher ist der Quellcode mit vielen kleinen Notizen oder `@todo` bestückt und eignete sich nicht für `javadoc`.

2 Beteiligte Software

In diesem Kapitel wird grob auf die genutzte Software beschreiben, ohne den kompletten Funktionsumfang dar zu legen. Die kommenden Unterkapitel beschreiben auch Probleme, die bei der Nutzung oder Dokumentation dieser Software auftraten. Bei wichtigen Komponenten, die in der Projektarbeit interessant waren, werden auch Details näher beschrieben.

2.1 DXRAM

DXRAM wird an der Heinrich-Heine Universität in Düsseldorf entwickelt und lebt von den Studenten, die diese Software unter anderem durch Projekt-, Bachelor- und Masterarbeiten befruchten. DXRAM ist sowohl in C als auch in Java (Version 8) geschrieben, wobei Beispiel Anwendungen und Benchmarks in Java geschrieben sind. DXRAM ist inzwischen nicht nur ein verteilter Key-Value-Storage, sondern umfasst ein ganzes „Ökosystem" von Modulen, mit bzw. in welches man eine verteilte Anwendung einfügen muss. Als Keys verwendet DXRAM Chunk-IDs, die auf „Chunks" – den allokierten Speicher – zeigen. Mehrere Peers, die durch Superpeers verwaltet werden, sind als Speicher- und Anwendungsknoten vernetzt. DXRAM übernimmt das Anlegen, Übertragen und ggf. Sperren von Chunks, wobei es bei Anwendungen weder eine Lastverteilung übernimmt, noch eine Entscheidung trifft, auf welchem Peer der Chunk angelegt wird. DXRAM stellt entsprechende Services in einer Anwendung zur Verfügung, um solche Entscheidungen dem Entwickler zu überlassen.

Bedauerlicherweise ist die Dokumentation auf viele Teilprojekte verteilt und es werden deployment Skripte empfohlen (github), die jedoch in ihrem Umfang den Start in die Entwicklung eher erschweren. Sollte man bereits einen Cluster passend konfiguriert zur Hand haben, mögen diese Skript nötig sein. Entwickelt man vorrangig lokal, ist der Workflow allerdings sehr übersichtlich: man kompiliert einmalig DXRAM, dann kompiliert man eine dxapp (siehe github), man kopiert diese dxapp in den passenden build-Ordner von DXRAM, trägt diese App in der Config von DXRAM einmalig ein, startet zookeeper, startet DXRAM als Superpeer, startet (ggf. mehrfach) DXRAM als Peer und ist dann fertig. Dies sind (auf lange Sicht) weniger als 6 Kommandozeilenbefehle und erfordert für einen ersten Start in DXRAM nicht zwingend ein komplettes Deployment-System.

Wie auch in anderen Software-Projekten gab es in DXRAM drei Arten von Beispielen:

- Hallo World (ohne z.B. wichtige Funktionen der Chunk-Speicherung zu zeigen)
- Ein Beispiel, was den kompletten Funktionsumfang abdeckt und für den Anfang zu komplex ist
- konkrete Anwendung, aber inzwischen veraltet

Eine konkrete Anwendung wie DXGraph wurde in Übungsblättern als Referenz genannt. Diese Wahl war zu diesem Zeitpunkt der Enwicklung jedoch nicht mehr gut gewählt da speziell DxGraph in einem alten Repository und Codezustand war. In dem Code gab es sehr praktische beispielhafte Chunks, die eine Speicherung von Objekten, welche nur aus Attributen unterschiedlichen Datentypens bestehen, beschrieben. Dort gab es diverse Methoden, um Strings und Arrays

in Chunks ab zu legen. Diese Methoden existieren auch weiterhin in DXRAM. Sie verleiten jedoch dazu, sich über die initiale Chunk-Größe im Speicher keine Gedanken zu machen und beim Ändern der Daten oder der Initialisierung den Aspekt der tatsächlichen Größe zu vergessen. Dies hat zum Ende der Projektarbeit die meiste Zeit gekostet, da Strings und Arrays in der Größe variierten, was bei der (De-)Serialisierung nach der Übertragung bei einem 2. Peer nicht abgefangen wurde.

2.2 DXNET

DXNET stellt in DXRAM die Übertragungsschicht dar. Es arbeitet unabhängig von DXRAM und kann in normalen Java-Anwendungen genutzt werden. DXNET ließ sich in Hadoop nutzen, um zwischen Hadoop und einer DXRAM Anwendung Daten austauschen zu können. Der Punkt der (De-)Serialisierung findet sich hier wieder: Während in DXRAM Chunks übertragen und (de-)serialisiert werden müssen, sind es in DXNET Messages. Da eine Message eher ein kurzweiliges Objekt ist, fielen die unterschiedlichen Längen bei Arrays nicht auf. Allerdings gab es dort ebenfalls Probleme bei DXNET (oder DXUTILS?) Methoden, die Strings betreffen. So wurden nun als alternative byte-Arrays mit fester Größe gefüllt, anstatt die fertigen Methoden von `dxutils` zu nehmen.

Zudem fehlte in DXNET ebenfalls ein griffiges Beispiel, um zwei Aspekte richtig zu verstehen:

- synchrone Datenübertragung
- Response Message

Dies führte im späteren Code zu einem sehr unschönen System, wenn z.B. zu einer ChunkId in Hadoop via DXNET nach einem Datei-Block gefragt wird: Nach dem Senden einer `AskBlockMessage` wartet in einer Schleife der Code darauf, dass der Message-Handler der `GetBlockMessage` ein statisches Objekt in der `AskBlockMessage` auf „nicht NULL" setzt. Dieser unfeine Code sollte einer weniger pragmatischen Lösung zum Austausch von Daten zwischen Hadoop und DXRAM weichen.

2.3 HDFS, Hadoop und HBase

Die Software HDFS, Hadoop (2.8.2) und HBase (1.4.0) ist jeweils in Java geschrieben und ihr Code ist Online verfügbar. Bei HDFS (Hadoop Distributed File System) handelt es sich um ein Dateisystem, welches Dateien in große Blöcke von einigen Megabyte aufteilt. Gegenüber normalen Dateisystemen ist es für große Dateien ausgelegt und speichert zusätzliche Informationen zu jedem Block ab, wie Replikationsnummer, Netzwerk-Infrastruktur Daten, Speichermedium und Ort/Host. Rund um dieses Dateisystem hat sich das Hadoop „Ökosystem" gebildet, welches z.B. MapReduce in einer verteilten Form anbietet und die CPU Ressourcen eines Datanodes, welcher einen Block trägt, nutzt. Die Begrifflichkeiten der Komponenten und deren Zusammenspiel haben sich in Hadoop über die Jahre immer wieder etwas geändert. Das Grundprinzip ist jedoch gleich geblieben: Verarbeite einen Block dort, wo er liegt, denn eine Prozessmigration ist schneller, als eine Datenübertragung eines ganzen Blocks.

Hadoop hat sich vom simplen MapReduce hin zu einem komplexen RessourceManager entwickelt (YARN), der Entwicklern eine verteilte Job/Batch/Task Verarbeitung ermöglicht, welche nicht nur auf MapReduce beschränkt ist. Alle Beschreibungen zu dem Thema sind allerdings auf HDFS Komponenten (Namenode, Datanode) ausgelegt, obwohl jeder Verarbeitungsknoten mit einem NodeManager nicht zwingend auf HDFS als Speicherresource ausgelegt ist. Mit Ignite und Alloxio gibt es z.B. zwei Projekte, die ein Dateisystem im RAM für Hadoop zur Verfügung stellen.

Während das Erste mit eigenen (inzwischen in Hadoop veralteten) MapReduce Komponenten an HDFS „vorbei arbeitet" so kopiert Alloxio das Hadoop Ökosystem im Grunde komplett, und stellt seine eigenen Komponenten zur Ressourcenverteilung zur Verfügung. Alloxio mountet fremde Dateisysteme in sich hinein, anstatt wie im ursprünglichen Hadoop Ökosystem diese nur parallel (mit unterschiedlichem „Scheme") anbieten zu können. Gleich, ob ein Projekt nahe des Hadoop Ökosystems ein eigenes System zur Prozessverarbeitung anbietet oder nicht, das Dateisystem, welches die Daten trägt, muss der Prozessverwaltung Informationen liefern, die die Wahl des ausführenden Knotens ermöglicht. Da als Anwendung HBase im Projekt geplant war, war ein Blick in Quellcode von HBase nötig, um eine Entscheidung zu treffen, was für Informationen hinterlegt sein müssen.

Scheme: In Hadoop wird zur Pfadangabe ein Scheme vorangestelle, welches das betroffene Dateisystem – also eine passende Implemenierung davon in Hadoop – sowie einen möglichen *Namenode* (Infos über `Blocklocation` zu einer Datei) beinhaltet.

HBase ist die Opensource-Version von Google BigTable. Es ist eine No-SQL Datenbank, bei der das schnelle Speichern von kleinen Änderungen eines Datensatzes im Vordergrund steht. Das Konzept ist vergleichbar mit einem Logging von Änderungen mit einem Zeitstempel, die später bei einem Snapshot zusammengeführt werden. Bei Anfragen an diese Datenbank wird das schnelle Ausliefern eines Datensatzes bevorzugt, der nicht zwingend die neusten Änderungen enthält. Sehr alte Informationen werden nicht gelöscht, sondern komprimiert abgespeichert und bei Bedarf wieder entpackt.

Obwohl das Abspeichern vieler kleiner Änderungen und das Ausliefern von evtl. alten Daten eher einem verteilten Key-Value-Storage entspricht, setzt HBase auf Hadoop und HDFS als verteilten Massenspeicher. HDFS, welches für große Dateien ausgelegt ist, bietet diverse Vorteile gegenüber einem reinen Key-Value-Storage:

- Verteilte Prozessverarbeitung
- klassifizierung des Speichermediums (RAM, SSD, HD)
- ein umfangreiches Framework zur verteilten Auswertung der gespeicherten Daten

Eine weitere Stärke von HDFS ist die `append()` Operation bei Dateien, bei der nicht die gesamte Datei, sondern nur der letzte Block geladen werden muss.

Die verteilte Prozessverarbeitung ist für HBase in sofern nötig, da Jobs wie Kompression, Entpacken und Snapshots regelmäßig neben der Anfragenbearbeitung und Speicherung anfallen. Im Code von HBase finden sich für die Kompression und Dekompression auch die Begriffe *crypt* und *encryption* sowie *EncodedDataBlock*.

Die Verteilung der Daten einer „Tabelle" wird über deren Key-Einträge gesteuert. Die Knoten, welche die Daten zu verarbeiten und verwalten haben, sind für einen Bereich an Keys der Tabelle verantwortlich. In HBase und Hadoop wird dies als `RegionServer` bezeichnet, der als Anwendung (*AppMaster*) auf einem `NodeManager` läuft. Im derzeitigen Reference Guide von HBase 2.1 wird gesagt, es handle sich dabei um einen `Datanode` – ein HDFS Knoten also. Das ist aber so nicht ganz korrekt, da streng genommen HBase das Scheme-Konzept für unterschiedliche Dateisysteme übernommen hat. Schaut man sich den Code in HBase genauer an, so fällt die Wahl des `RegionServers` auf den *Host*, der zu den angefragten Daten die meisten Blöcke besitzt. Sollte das Dateisystem hier die Methode `getFileBlockLocations()` nicht überschreiben, wird nur `localhost` und als HDFS Datatransfer-Port `50010` zurück gegeben.

Wie der genaue Zugriff auf einzelne Blöcke in Hadoop und HBase abläuft, bliebt in der gesamten Zeit der Projektarbeit schwer nachvollziehbar. Ob ein Prozess initial auf einer Datei und somit auf einem NodeManager mit den meisten Blöcken ausgeführt wird, und dann beim Laden der Blöcke ein Prozess auf einem Block zu einem anderen NodeManager migriert wird, konnte nicht

heraus gefunden werden. Die default Portangabe `50010` für einen HDFS Datanode deutet darauf hin, dass hier KEINE Logik greift, die Blöcke jedes Dateisystems an einen ausführenden Knoten überträgt (AppMaster auf einem NodeManager) oder eine Prozessmigration stattfindet. Dieser Port wird nur für Daten eines HDFS Datanodes gebraucht. Dies legt nahe, dass bei einer Operation auf einer Datei in einem Nicht-HDFS Dateisystem auch spezielle Operation bei diesem Dateisystem zur Verfügung gestellt werden müssen, um eine Operation auf Blöcken nahe des Datenspeichers zu ermöglichen.

Dieser letzte Abschnitt legt nahe, dass man Hadoop und HBase mit anderen Dateisystemen als verteilte Anwendung nutzen kann. Will man jedoch Prozesse auf einzelnen Blöcken statt Dateien ermöglichen, kann man nicht mehr auf die Hadoop Routinen, welche auf HDFS ausgelegt sind, zählen. Interessant ist die Tatsache, dass in der abstrakten Hadoop Klasse `FileSystem` keine `getBlock()` Methode existiert. Hadoop arbeitet wohl primär mit `append()` auf dem letzten Dateiblock und erzeugt neue Dateien, anstatt in bestehenden die Blöcke zu verändern. Diese Vermutung stützt sich auch beim Einbau eines Loggings im original HDFS Code, wo bei MapReduce Beispielen zwischendurch ständig Dateien angelegt, gelöscht und umbenannt wurden.

2.4 Maven, Gradle, Versionierung und Codepflege

Das Hadoop Ökosystem mit HDFS (Version 2.8.2) lässt sich prima mit Maven kompilieren/bilden. Dies galt auch für den Google Cloud Storage Connector, der ab 8. Februar 2018 eine gute Orientierungshilfe war. Davor wurden Code-Anpassungen und Erweiterungen in Hadoop via IDE gemacht und gebildet. Trotz Maven war es ein erheblicher Zeitaufwand, alle Bibliotheken von DXNET und Hadoop in den passenden Versionen aufeinander ab zu stimmen, so dass der erstellte DXRAM Connector als jar-File inkl. weiterer Bibliotheken in Hadoop problemlos lief. Bei der Umstellung von Maven auf Gradle im August 2018 bei allen DXRAM Projekten konnte diese Umstellung nicht mehr adäquat in den Connector einpflegt werden (es bleib bei Maven). Da die Bibliotheken nun in die DXRAM Projekte eingebunden wurden, gab es nun eher weniger Probleme bei der Abstimmung mit Hadoop. Um in der IDE leichter arbeiten zu können und wegen kleinerer Änderungen an DXNET ist dieser Code händisch in den Code des Connectors eingepflegt. DXNET 0.5.0 ist also als Quellcode in den Connector eingepflegt.

Der DXRAM Part des Projekts ist hingegen problemlos auf gradle migriert worden. Dies gilt auch für einen DXNET Client, mit dem zum Ende der Projektarbeit Anfrage an die DXRAM Applikation getestet werden konnten, ohne über den DxramFs Connector in Hadoop gehen zu müssen.

Die Erfahrungen mit Maven und Gradle waren oft frustrierend. Zwar gab es Tutorials (speziell für Maven), aber das Erzeugen eines *.pom* Files, welche Plugins wie Lombok unterstützen, wie passende Bibliotheken für Hadoop herunter geladen werden, wie andere jar-Files für DXNET berücksichtigt werden, war am Ende eher Trial-And-Error- sowie Copy-Paste-Arbeit aus anderen Projekten oder aus „stackoverflow''. Das es bei den Plugins und Maven ebenfalls unterschiedliche Versionen gibt, dann noch eine „best practice" Empfehlung gibt oder Beispiele veraltet oder falsche sind, kommt noch hinzu. Speziell an dieser Stelle wurde nachvollziehbar, warum sich ein Programmierer wochenlang mit so etwas beschäftigt und als Leiter eines neuen Softwareprojekts keine Zeile Quellcode schreibt und sich nur damit auseinander setzt, dass das Build-System korrekt läuft. Zwei größere Frameworks in einem Projekt zusammen zu führen, ist nicht trivial.

2.5 Zookeeper und Kerberos

Zur Zeit benötigt DXRAM (0.5.0) noch Zookeeper. Dies gilt auch für HBase, welches zusätzlich Kerberos nutzt. Mit Kerberos und HBase traten leider immer Probleme auf. Die Java Beispiele für

HBase, als auch die HBase Console kränkelten, obwohl keine Änderungen in Hadoop eingebunden waren. Lediglich von HBase ausgelieferte jar Beispiele konnten getestet werden. Im März 2018 fiel daher die Entscheidung, sich erst mal auf die Hadoop MapReduce Beispiele zu beschränken, um weiter zu kommen.

3 Entscheidungsfindung

Ein Blick auf den Quellcode von HBase zeigte, dass es für einen Client, welcher auf HBase zugreifen will, für mehrere Programmiersprachen eine API gibt. Schaut man sich diese jedoch genauer an, verbirgt sich dahinter Apache Thrift. Thrift war durch die Bachelorarbeit Entwurf und Implementierung verteilter Lösungsansätze für Capital Budgeting Probleme mit GLPK und Apache thrift bereits bekannt. Es ist vergleichbar mit RPC von Sun (bzw. ONC+ von Oracle) und erzeugt Code-Gerüste (Server wie Client) für unterschiedliche Sprachen auf der Basis einer IDL. Diese Thrift Datei, in der alle nötigen Datenstrukturen und Calls beschrieben sind, ließe sich leicht verwenden, um das gesammten Request-Handling von HBase durch eine eigene Implementierung aus zu tauschen. Wenn man keinen Wert auf Snapshots und (De)Komprimierung legt, ist dies auf den ersten Blick die beste Lösung, um HBase durch eine DXRAM Anwendung aus zu tauschen – für die Zukunft wäre das ein interessantes Projekt. Ein anderes Projekt wäre, wenn man DXNET oder Infiniband auf der Basis des Template Konzepts von Thrift in Apache Thrift zur Verfügung stellt. Zur Zeit der o.g. Bachelorarbeit schien es in Thrift nur Sockets und für Javascript-in-Browser-Clients Websockets zu geben.

Eine tiefere Recherche zum Thema HBase zeigte, dass dessen Anwendungen vorrangig in Java geschrieben werden. Die HBase API bietet wohl dort mehr Möglichkeiten. Da HBase auf Hadoop aufbaut, drängt sich das Hadoop Ökosystem zur Verarbeitung von Daten in HBase als Framework auf. DXRAM sollte mit dieser Projektarbeit in einem populären Anwendungszweig eintauchen, wo ein komplettes HBase- und Hadoop-Replacement weniger ratsam wäre, als eine Integration von DXRAM in Hadoop. Hadoop (und HBase) wirbt damit, dass es nicht auf HDFS als Dateisystem angewiesen ist. In Hadoop findet sich unter anderem eine FTP-Dateisystem Anbindung oder eine auf das lokale Dateisystem. Die Idee war, hier nun ein DxramFs zu hinterlegen, welches anstelle von HDFS unser DXRAM als Massenspeicher nutzt. DXRAM als Dateisystem in Hadoop zu implementieren, würde allen Hadoop und HBase Anwendungen ohne Programmieraufwand DXRAM als Massenspeicher zur Verfügung stellen.

Für die Idee *DXRAM als Dateisystem in Hadoop* gab es zwei Möglichkeiten, die man in Betracht ziehen konnte:

1. DxramFs mountfähig unter Linux mit libfuse.
2. DxramFs auf die selbe Art wie HDFS oder FTP als Dateisystem in Hadoop implementieren.

Die erste Möglichkeit baut auf der Idee auf, man könne doch in HBase und Hadoop als Speicherort die Angabe `file:///tmp/dxrammount/` machen. Auf diese Weise würde man DXRAM als Massenspeicher auch noch einer Vielzahl anderer Systeme zur Verfügung stellen. Es wäre dann vergleichbar mit NFS. Da NFS auf RPC aufbaut und nicht libfuse nutzt, könnte man sich dies als zukünftiges weiteres Projekt vorstellen, bei dem man in den Code von NFS schaut, sofern dieser von Sun bereitgestellt wird. Die Nutzung des `file://` Scheme in Hadoop lässt den Code von Hadoop unberührt und man verliert die Möglichkeit, dem Hadoop Ökosystem Informationen zum Standort der Daten zu geben. Die Idee, in einer Ordnerstruktur die Lokalisation der Daten zu hinterlegen, wäre nicht so gut. Auf diese Art würde man das Prinzip aufgeben, dass eine Anwendung beim Zugriff auf ein Verteiltes System dieses als ein „Ganzes" sieht. Die Verteilung der Daten wäre nicht mehr transparent/versteckt und die Anwendung müsste sich beim Dateizugriff

mit den Knoten und dem Speicherort auseinander setzen.[1] Zusätzlich wurde von `libfuse` abgeraten, da innerhalb der Betriebssystem Gruppe von Performance-Problemen berichtet wurde, die jeden Speedup von DXRAM zu Nichte machen würde.

Die zweite Möglichkeit DXRAM als Dateizugriff in Hadoop ein zu bauen, ist eine Implementierung vergleichbar mit anderen Konnektoren wie FTP, viewfs, HDFS usw. Hierzu gibt es sowohl Beispiele und einen Contract, der das verteilte Verhalten des neuen Dateisystems beschreibt, welches unter anderem nicht allein durch Unittests geprüft werden kann. An dieser Stelle muss einem klar werden, das Hadoop einem nicht das verteilte Sperren auf Dateien abnimmt, sondern dass man dies selbst implementieren muss. Würde das Interface von Hadoop einem dies automatisch abnehmen, müsste in dem *Contract* kein Hinweis bzgl. atomarer Operationen stehen. Diese Form der Einbindung von DXRAM in Hadoop (als „Connector") kann genutzt werden, um weitere Information über den Speicherort der Daten zu hinterlegen. Dafür Verantwortlich ist die Methode `getFileBlockLocations()` , welche im gesamten Projektzeitraum nur beim *RegionServer* von HBase verwendet wurde, wo es um die Wahl des Hosts geht, welcher zu den angefragten Dateien die meisten Dateiblöcke beheimatet. Im Code des RegionServers könnte man evtl. in Zukunft auch zwischen HBase und DXRAM eine Verbindung herstellen.

Zwischendurch kam auch die Idee, man könne doch von Ignite oder anderen Projekten, die mit einem Key-Value-Storage arbeiten und eine Hadoop-Anbindung ermöglichen, diesen Code übernehmen und entsprechend an DXRAM anpassen. Nach ca. 2 bis 3 Tagen wurde diese Idee jedoch verworfen. Nach dem Einlesen in den Quellcode von HBase, Hadoop, den Google Cloud Storage Connector, sowie dem Hadoop DFS Client (HDFS) war speziell in Ignite kein Einstieg erkennbar, wo sich der Key-Value Zugriff durch eine Alternative auf Basis von DXRAM austauschen ließe.

Die Wahl, wie sich DXRAM als Massenspeicher in HBase Anwendungen zur Verfügung stellt, fiel auf das Implementieren eines Hadoop-kompatiblen Dateisystems, wie es z.B. beim Hadoop eigenen FTP-Connector der Fall ist. Die Vorteile dieser Lösung gegenüber den Anderen sind:

- Gewinnung von Interesse an DXRAM in der HBase- und Hadoop-Community (weil kein Replacement)
- keine Performance-Einbußen bei verteilter Ausführung durch den Verlust von Informationen bzgl. des Speicherorts
- weiterhin Nutzung des Hadoop Ökosystems bei der Prozessverteilung
- minimaler Anpassungs-Aufwand bei HBase Anwendungen (Verwendung `dxram://` statt `hdfs://` in der Konfiguration)

4 Konzept und Umsetzung

Die Grundidee war sehr einfach: Kopiere z.B. den `fs/ftp/` Ordner mit dem FTP-Connector im Hadoop Quellcode, nenne alles, was FTP heißt um auf DXRAM, entferne den FTP-spezifischen Code, und tausche alles bei `connect()`, `disconnect()` sowie den grundlegenden Dateioperationen, wo `@Override` darüber stand, durch eigenen Code aus. Hinterher kompiliere Hadoop in der Console, und schon kann man mit folgenden Befehlen die Methoden testen:

```
bin/hadoop fs -mkdir -p dxram://dummy:9999/test/abc
bin/hadoop fs -put README.txt dxram://dummy:9999/
bin/hadoop fs -ls dxram://dummy:9999/
... und andere Operationen.
```

[1] Dies ist auch ein Kritikpunkt an DXRAM, da in der Version 0.5.0 der Peer als ID angegeben werden muss, wo ein Chunk abgelegt werden soll. Das Handling, ob dieser Peer online ist und ob dort überhaupt noch Speicher zur Verfügung steht, ist somit in die Verteilte Anwendung übertragen worden und nicht transparent.

Der Aufbau von diesem FTP Connector war recht simple und wurde übernommen, anstatt sich am Code des *Google Cloud Storage Connectors* zu orientieren. So ist `DxramFs` bis heute eine Erweiterung von `DelegateToFileSystem` und trägt in einer privaten Variable des Typs `DxramFileSystem` ein Implementierung des Dateisystems (`extends FileSystem`). Später wurde vom *Google Cloud Storage Connectors* nur übernommen, wie sich dieser Code in Hadoop als jar-File einbinden lässt, ohne Hadoop (bzw. Teilprojekte davon) neu bilden zu müssen. Wichtig ist es, in der Hadoop Konfiguration `core-site.xml` diese Implementierungen mit an zu geben:

```
1     <property>
2         <name>fs.dxram.impl</name>
3         <value>de.hhu.bsinfo.dxramfs.connector.DxramFileSystem</value>
4         <description>The FileSystem for dxram.</description>
5     </property>
6     <property>
7         <name>fs.AbstractFileSystem.dxram.impl</name>
8         <value>de.hhu.bsinfo.dxramfs.connector.DxramFs</value>
9         <description>
10            The AbstractFileSystem for dxram
11        </description>
12    </property>
```

Um von DXNET und DXRAM ungestört und losgelöst einen Connector anfänglich testen zu können, wurde zuerst alles so implementiert, dass Dateien im Ordner `/tmp/myfs/` abgelegt werden. Dieser Connector war also zuerst ein Dummy- oder „Fake-Dateisystem". Bei dessen Implementierung wurde schnell klar, dass sich fast alle FileSystem Operationen auf eine Datei bezogen. Außer private Methoden zur Pfad-Korrektur, Anfragen an ein Home- oder Working- Verzeichnis oder connect/disconnect Methoden, wurde daher alles in einer `DxramFile` Klasse implementiert. Ein Blick in den Hadoop FS Compatibility Contract – kurz HCFS – scheint sowas auch vor zu sehen. Ab Hadoop v2 solle man `FileContext` implemetieren statt FileSystem. Da es in Hadoop eine gewisse Rückwärtskompatibilität gibt, habe ich das ignoriert. Zumal Hadoop in ihrem eigenen FTP-Connector dies so nicht macht. Wie historisch dieser Code ist, merk man auch an diversen Deprecated Warnungen beim Bilden und Ausführen. Allerdings gibt es in HBase eine Klasse `HFileContext`, welche in dem Zusammenhang hätte interessant sein können. Diese entpuppt sich jedoch nicht als hilfreich, da sie mit der `FileContext` Klasse von Hadoop nichts zu tun hat.

Um ein Gefühl für die Verwendung der FileSystem Methoden bei Mapreduce oder kleinen HBase Beispielen zu bekommen, wurden im Rahmen der Projektarbeit in das original HDFS zahlreiche Log Infos eingebaut. Falls man sich die Suche des HDFS Connectors im Hadoop Code ersparen will, es ist in `hadoop-hdfs-project/hadoop-hdfs-client/` und dort in `src/main/java/org/apache/hadoop/hd` und der Datei `DistributedFileSystem.java` zu finden.

Bis 9. März 2018 lief das Fake-Dateisystem mit Mapreduce, jedoch wollten HBase Beispiele damit nicht laufen. Man erhielt immer wieder die Meldung ...

```
2018-03-05 16:56:09,817 ERROR [main] zookeeper.RecoverableZooKeeper: ZooKeeper exists
  failed after 4 attempts
2018-03-05 16:56:09,818 WARN  [main] zookeeper.ZKUtil: hconnection-0x37c366080x0,
  quorum=localhost:2181, baseZNode=/hbase Unable to set watcher on znode (/hbase/hbaseid)
org.apache.zookeeper.KeeperException$ConnectionLossException: KeeperErrorCode =
  ConnectionLoss for /hbase/hbaseid
  at org.apache.zookeeper.KeeperException.create(KeeperException.java:99)
```

... wenn ich die `hbase shell` aufrufen wollte. Das zookepper, was von DXRAM als zip mitgeliefert wurde, funktionierte aber. Auch wenn dieser Fehler nicht darauf hindeutet, waren zwei Dinge bis dahin noch offen:

- das Fake-Dateisystem hatte bis dahin kein `append()` implementiert, welches HBase braucht.
- es ist unklar, ob das Fake-Dateisystem ähnlich wie bei Hadoop in der Config und als jar-File bei HBase hinterlegt werden muss.

An dieser Stelle wurde im Projekt ein Schnitt gemacht und sich nur noch auf Mapreduce als Mindest-Applikation konzentriert Es lag die Vermutung nahe, dass mit HBase etwas nicht stimme und nicht zwingend am Fake-Dateisystem. So wurde nun begonnen, die Operationen des Fake-Dateisystems stückweise mit DXNET an eine DXRAM Applikation weiter zu leiten, da sich DXRAM nicht direkt in Hadoop einbauen ließ. [2]

Im HCFS gab es noch einen weiteren Punkt, der leider bis heute nicht geklärt ist. Dort steht als Anforderung:

> The filesystem looks like a "native" filesystem, and is accessed as a local FS...

Schaut man sich die Tutorials an, so wird viel mit ssh gearbeitet, was auf einen gewissen lokalen Dateizugriff hindeuten kann. Schaut man in die Readme des *Google Cloud Storage Connectors*, welcher auf den ersten Blick ein Mounten ins lokale Dateisystem nicht aufführt, so ist diese Aussage im HCFS vermutlich historisch bedingt, als YARN und ein NodeManager in Hadoop noch nicht existierten. In der Readme des *Google Cloud Storage Connectors* heißt es als Feature:

> **Quick startup**: In HDFS, a MapReduce job can't start until the NameNode is out of safe mode – a process that can take from a few seconds to many minutes depending on the size and state of your data. With Google Cloud Storage, you can start your job as soon as the task nodes start, leading to significant cost savings over time.

Ein NameNode, welcher den DataNode zu den angefragten HDFS-Daten kennt, ist also auch hier nicht nötig und wie gesagt: ein Mounten ins lokale Dateisystem gibt es bei diesem Connector nicht. Wie solle dies auch gehen? Die Knoten des Google Cloud Storage sind auch weder mit ssh erreichbar, noch werden sie als ausführende Knoten bei Hadoop gelistet.

Prinzipiell ist im Scheme Konzept von Hadoop dieser Teil `system://[host]:[port]/` für den Hostname und Port des Namenodes, welcher die Metadaten wie die BlockLocations enthält, vorgesehen. Der DxramFs Connector nutzt dies nicht, da sich hier via DXNET und DXRAM Informationen über den Standort der BlockChunks (das Analogon der HDFS File-Blocks) erfragen lassen. Das DXRAM Superpeer-Peer Konzept sieht hier keinen einzelnen Knoten für ein Lookup vor.

4.1 Aufbau des DxramFs

Als Idee für den Aufbau des Dateisystems wurde eine Mischung aus Verkettung (doppelt verkettete Liste) und Arrays genommen. Ein `FsNode` ist zur Speicherung des Ordner oder Dateinamens gedacht, sowie diveres Größenangaben. Mit einem Array `refIDs` aus Referenznummern wird auf

[2] Die Konzepte von Ignite und Alluxio, welche ein anderes, nicht RPC-ähnliches Konzept verfolgen, um einen Zugriff auf ihr Dateisystem aus einer konkreten verteilten Anwendung heraus ermöglichen, werden erst später in diesem Bericht betrachtet. Diese Konzepte entfernen sich jedoch von der Idee, dass nur mit einem File-System Connector in Hadoop und Infos zum Speicherort eine verteilte Prozessverwaltung allein durch Hadoop ermöglicht wird.

weitere Ordner oder Dateien (ebenfalls `FsNode`) gezeigt. Ist der `FsNode` eine Datei, so ist das Array mit Referenznummern auf `Blockinfo` gefüllt. Diese Datenstruktur enthält dann in etwa die Informationen, welche in Hadoop beim Aufruf von `getFileBlocklocations()` gebraucht werden. Darunter ist auch eine Referenznummer auf den `Block`, der den Teil der Daten einer Datei trägt. Sollte das Dateisystem mit dem Array nicht mehr auskommen, so gibt es für `FsNode` noch den Typ `EXT` (neben den Typen `FILE` und `FOLDER`). Die Referenznummer eines solchen `FsNode` wird dann in `forwardId` beim vollen `FsNode` eingetragen (Verkettung). Die Grafik *Daten-Management von DxramFs* stellt ein Beispiel dar, wie Ordner, Metadaten und Dateien abgelegt werden können. Die Größe der Referenz-Arrays lässt sich einmalig in der Config Datei von Hadoop hinterlegen bzw. bei der `DxramFs` Applikation in DXRAM. Dort sind auch Angaben bzgl. maximale Pfadlänge oder maximale Länge des Dateinamens hinterlegt. In der Klasse `DxramFsConfig` werden diese Daten beim Programmstart (DXRAM App und Connector) abgelegt. Auf die gleiche Weise wird ein Node zu Peer Mapping (Hadoop zu DXNET und DXRAM) aus den Config-Dateien gelesen und in der `NodePeerConfig` Klasse abgelegt.

Auf der Hadoop Code Seite, also dem `connector/`, sollten nur die drei Klassen `FsNode`, `Blockinfo` & `Block` arbeiten sowie ein paar DXNET Komponenten, die sowohl im Connector als auch der DXRAM Applikation verwendet werden. Dieser Codezweig ist in `de.hhu.bsinfo.dxapp.dxramfscore.*` hinterlegt, in dessen Sub-Package `rpc` die DXNET Nachrichtentypen enthalten sind. Die `MessageHandler` dieser Nachrichten Typen sowie TAGS und andere Konstanten, befinden sich ebenfalls in den Klassen dieser Nachrichtentypen.

Im `dxram_part/` Ordner des Projekts ist der Code für die DXRAM Applikation hinterlegt. Im Package `de.hhu.bsinfo.dxapp.dxramfspeer` sind die Klassen für die Chunks hinterlegt, welche die Daten von `FsNode`, `Blockinfo` & `Block` tragen sollten. Wie weiter oben bereits erwähnt, war es mehr als frustrierend, dass sich nicht so problemlos wie erwartet die drei „Attribut-Klassen" initialisieren, mit DXNET übertragen, gefüllt und in DXRAM als Chunk abspeichern ließen. Ein Minimalbeispiel als DXRAM Projekt, welches exemplarisch in Github angelegt wurde und die Situation beschreibt, ist bis November 2018 von der Projektgruppe unkommentiert geblieben. Sollte man DXRAM in Zukunft in bestehende Projekte einbauen, ist ein Satz unterschiedlicher Attribute in einer Klasse, welche sowohl in DXRAM zur Speicherung (und DXNET zur Übertragung) als auch in einer anderen Anwendung eingesetzt wird, nichts ungewöhnliches. Für jeden elementaren Datentyp einen eigenen Chunk definieren zu müssen und dann nur mit deren Chunk-IDs ein Chunk Objekt zusammen zu setzen, erscheint nicht programmierer- und perfomance-freundlich. Ist diese „Attribut-Referenzierung" in DXRAM so gewollt?

4.2 Konfiguration der Projekt-Komponenten

Die nachfolgende Grafik stellt den eingeplanten Aufbau des Projekts dar. Hierbei tritt der `RegionServer` einmal als konzeptionelle Komponente in der *HBase Wolke* auf, als auch als `App Master` in einem NodeManager, welcher Teil von YARN (Hadoop) ist. In der Grafik zeigen Nummern wichtige Ports, die als Beispiel im Projekt verwendet wurden. HDFS mit Namenode und Datanode ist in Grün angedeutet, um das Replacement von HDFS gegenüber DxramFs zu verdeutlichen und ggf. Probleme mit diesem Aufbau besser ansprechen zu können.

Das Zusammenspiel der Komponenten ist leicht erklärt: HBase nutzt Hadoop und gibt beim Pfad der Daten ein Scheme mit an, welches Hadoop anweist, auf DxramFs anstellte von HDFS zuzugreifen. Sobald z.B. nach dem Speicherort der Blöcke einer Datei gefragt wird, leitet der DxramFs (Connector) diese Anfrage an seinen zugewiesen (lokalen) DXNET Peer. Diese Verbindung läuft dann z.B. auf den Ports 65220 und 65221 ab. Dieser DXNET Peer ist auch ein DXRAM Peer (z.B. Port 22222) und nutzt nun den `LookupService` von DXRAM, um zu einer ChunkId (welche den BlockChunk repräsentiert) den passenden Host zu finden. Auf diese Weise

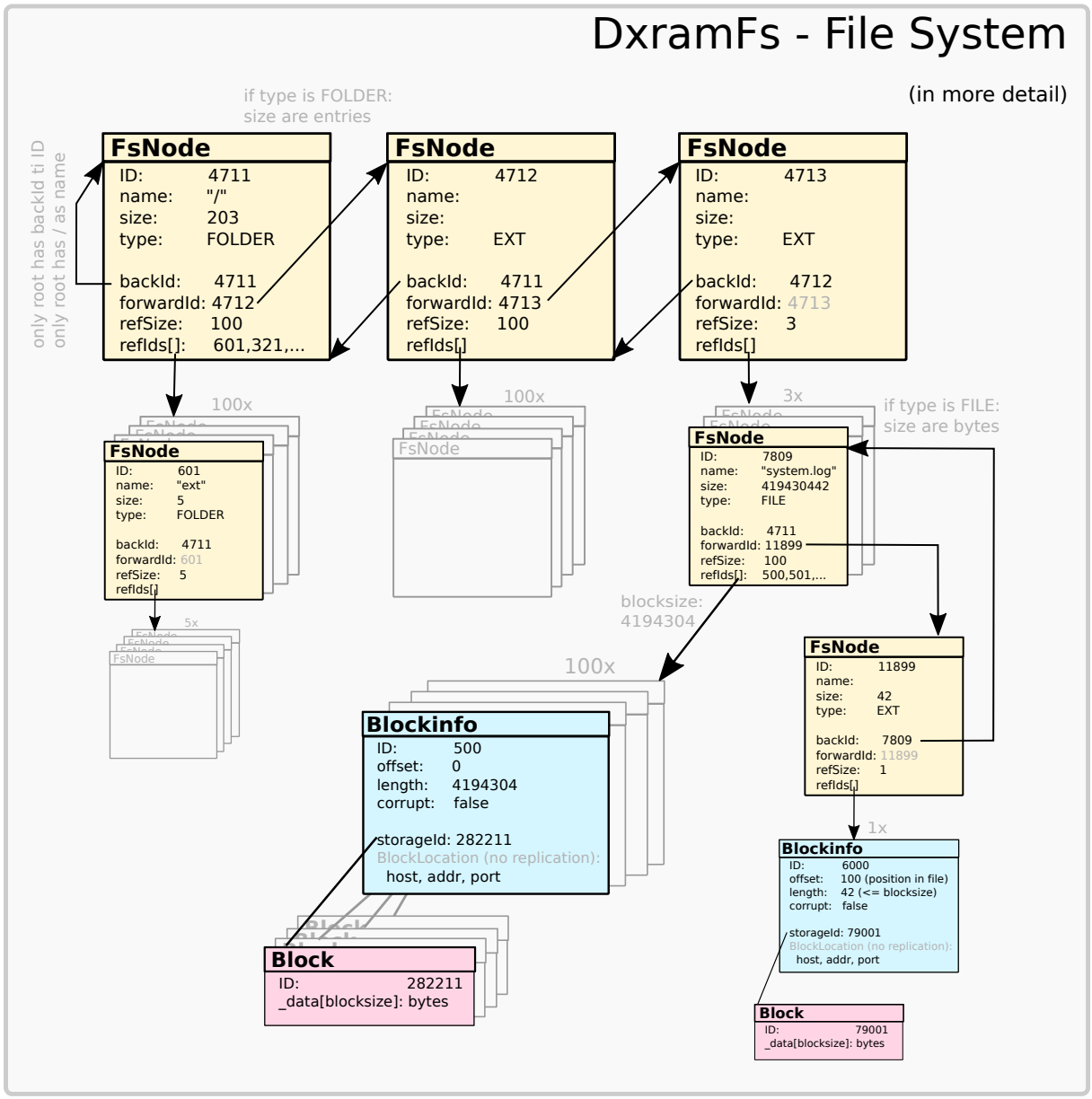

Abbildung 2: Daten-Management von DxramFs

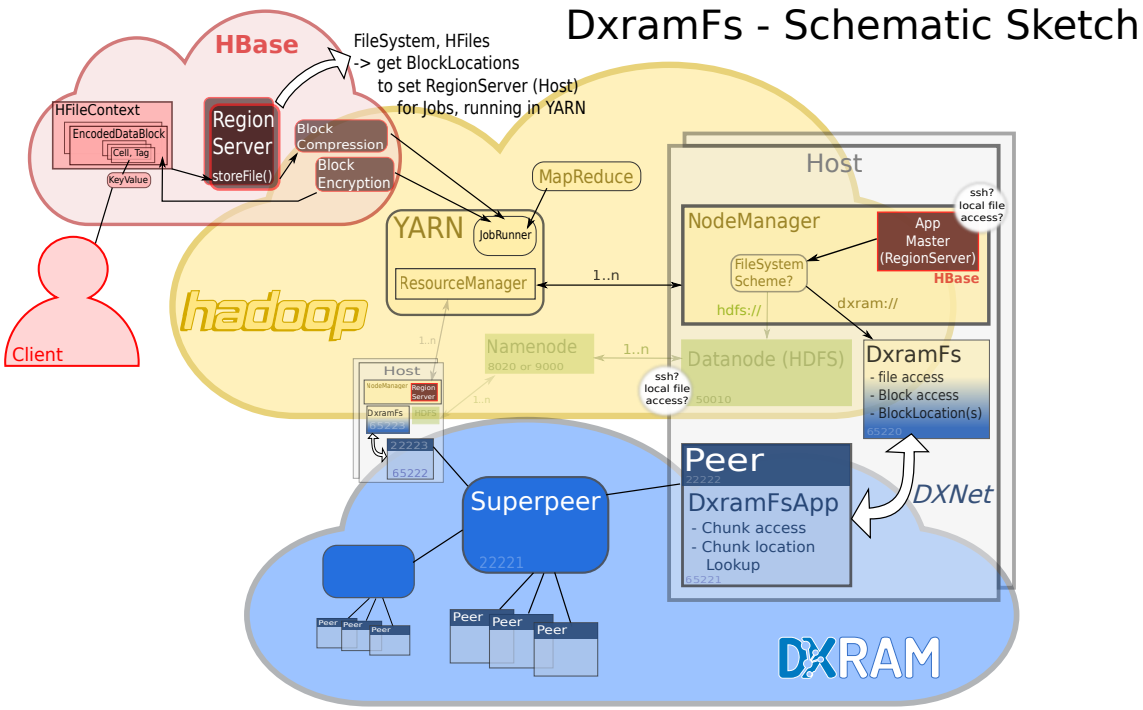

Abbildung 3: DXramFs: Zusammenspiel von HBase, Hadoop und DXRAM

kann YARN eine Entscheidung treffen, welcher `RegionServer` zur Verarbeitung gewählt werden muss.

Für diesen Aufbau existiert ein Node-Peer Mapping, mit dem z.B. eine gestartete DXRAM Anwendung weiß, auf welchem Port sie einen DXNET Peer zu starten hat. Ebenso mussten alle DXNET IDs der Peers bekannt sein. Da zu Beginn des strukturellen Aufbaus nicht ganz klar war, welche Informationen (speziell: Port) bei der BlockLocation in Hadoop erwartet wurde, und ob/wie eine Prozessmigration statt findet, war der DXNET Client im Connector so ausgelegt, dass prinzipiell mit jedem Host/NodeManager/Peer kommunizieren kann. DXNET nur auf `localhost` zu nutzen, erschien zu eingeschränkt. Für die Zukunft könnte man hier eine andere Lösung suchen, die via DXNET direkt mit DXRAM kommuniziert oder – da es sich um die selbe physische Hardware handelt – ein Netzwerk freier Weg wählen, um auf Chunks des lokalen DXRAM Peers von einer anderen Anwendung aus zu zu greifen (JNI, Messagebus, ... ?).

Das Mapping der PeerIds, Ports und Hosts bzw. IP-Adressen ist in der Datei `DxramFsApp.conf` unter `dxnet_to_dxram_peers` abgelegt. In Hadoop kann dies in der `core-site.xml` gepflegt werden. Dort sind analog zur `DxramFsApp.conf` auch weitere Angaben zu pflegen:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3  <configuration>
4      <property>
5          <name>fs.dxram.impl</name>
6          <value>de.hhu.bsinfo.dxramfs.connector.DxramFileSystem</value>
7          <description>The FileSystem for dxram.</description>
8      </property>
9      <property>
10         <name>fs.AbstractFileSystem.dxram.impl</name>
```

```
11          <value>de.hhu.bsinfo.dxramfs.connector.DxramFs</value>
12          <description>
13              The AbstractFileSystem for dxram
14          </description>
15      </property>
16      <property>
17          <name>fs.defaultFS</name>
18          <value>dxram://localhost:9000</value>
19      </property>
20
21      <property>
22          <name>dxram.file_blocksize</name>
23          <!-- blocksize shuld be smaller than chunksize -->
24          <value>4194304</value>
25      </property>
26
27      <property>
28          <name>dxram.ref_ids_each_fsnode</name>
29          <value>128</value>
30      </property>
31
32      <property>
33          <name>dxram.max_pathlength_chars</name>
34          <value>512</value>
35      </property>
36
37      <property>
38          <name>dxram.max_filenamelength_chars</name>
39          <value>128</value>
40      </property>
41      <property>
42          <name>dxram.max_hostlength_chars</name>
43          <value>80</value>
44      </property>
45      <property>
46          <name>dxram.max_addrlength_chars</name>
47          <value>48</value>
48      </property>
49
50      <property>
51          <name>dxnet.me</name>
52          <value>0</value>
53      </property>
54
55      <property>
56          <name>dxnet.to_dxram_peers</name>
57          <!-- me is talking to localhost:65221 or localhost:65222, and
58          them are talking to localhost:22222 or 22223.
59          The dxnet-dxram peer mapping localhost:65221 at localhost:22222 is
60          good, to identify the location of a block. -->
61          <value>0@127.0.0.1:65220@,1@127.0.0.1:65221@127.0.0.1:22222,
            ↪  2@127.0.0.1:65222@127.0.0.1:22223,3@127.0.0.1:65223@</value>
62      </property>
63
64  </configuration>
```

Hier sei `dxnet.me` mit dem Wert `0` besonders erwähnt, mit dem der Connector aus `dxnet.to_dxram_peers`

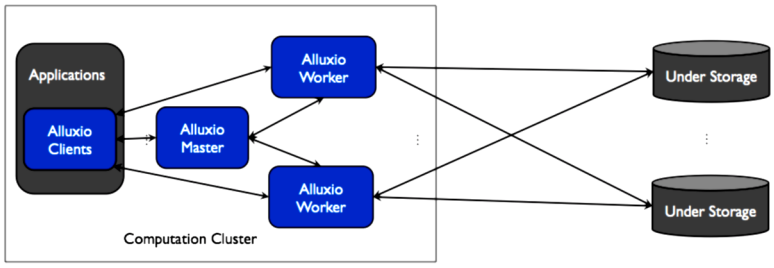

Abbildung 4: Alluxio Master-Worker

sich den passenden Port als DXNET peer heraus sucht. Diese Angabe müsste bei einem zweiten Hadoop Node geändert werden (z.B. 3).

Damit die DXRAM Applikation `DxramFsApp` ihre Konfigurationsdatei einlesen kann, muss man deren Pfad als Parameter in der `config/dxram.json` Datei mit angeben:

```
    "m_autoStart": [
        {
            "m_className": "de.hhu.bsinfo.dxapp.DxramFsApp",
            "m_args": "dxapp/DxramFsApp.conf",
            "m_startOrderId": 0
        }
    ],
```

Start man z.B. so einen DXRAM Peer ...

```
DXRAM_OPTS="-Dlog4j.configurationFile=config/log4j2.xml \
 -Ddxram.config=config/dxram.json -Ddxram.m_config.m_engineConfig.m_role=Peer \
 -Ddxram.m_config.m_engineConfig.m_address.m_ip=127.0.0.1 \
 -Ddxram.m_config.m_engineConfig.m_address.m_port=22222" ./bin/dxram
```

... kann die Applikation im Ordner `dxapp/` die Config-Datei finden und erkennen, dass sie als DXRAM Peer auf Port 22222 auch einen DXNET Peer auf Port 65221 zu starten hat.

5 Vergleich und Aussicht auf Weiterentwicklung

5.1 Vergleich mit anderen Projekten

Alluxio (früher: Tachyon) hat einen Master-Worker-Aufbau und ist dem ResourceManager-NodeManager-Aufbau von YARN sehr ähnlich. Alluxio ist im Grunde ein Branch des Hadoop Ökosystems und daher schlecht mit dem Aufbau von DxramFs vergleichbar. Interessant an dem Projekt ist, wie sich das Alluxio Dateisystem im Hadoop Ökosystem (speziell YARN) nutzen lässt, da es einen eigenen Connector zur Verfügung stellt, der mit dem Connector von DxramFs vergleichbar ist. Da Alluxio mit dem Master-Worker-Aufbau ein eigenes Konzept für verteilte Anwendungen hat, wird es nicht empfohlen, YARN von Hadoop zu benutzen. DxramFs fehlt bisher eine eigene Struktur, um YARN in vergleichbarer weise zu umgehen. Die Projektentscheidung war ja, nur mit Informationen aus der `BlockLocation` des DxramFs-Connectors die Hadoop Prozessverwaltung YARN effektiv mit nutzen zu können und keine eigene Prozessverwaltung zu erstellen.

Der Google Cloud Storage Connector hat anders als Alluxio kaum einen Eingriff ins Hadoop Ökosystem gemacht. Genau wie der DxramFs Connector muss YARN die Prozesse verwalten. Er

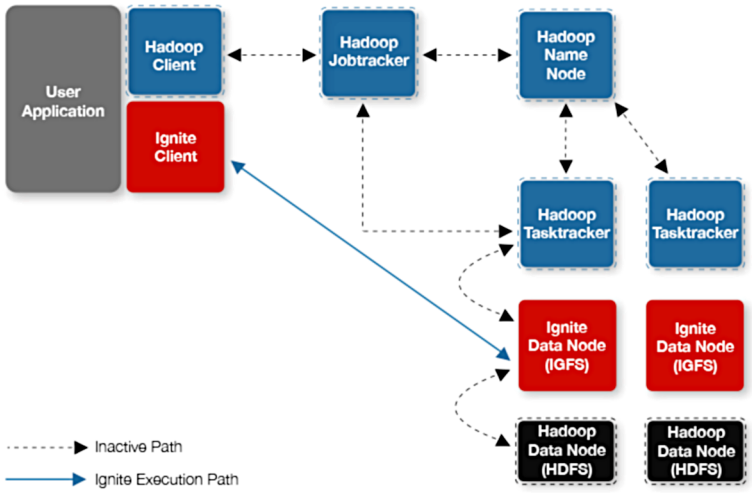

Abbildung 5: Ignite Client und Ignite Data Node

ist im Grunde fast mit dem FTP Connector von Hadoop vergleichbar. Der Code dieses Connectors sowie dessen Anleitungen sind interessant, da sie eine gute Basis für DxramFs waren, um vom Hadoop Quellcode losgelöst ein Hadoop Filesystem Projekt zu erstellen.

Ignite ist als weiteres Projekt zu nennen, welches wie Alluxio die Daten bzw. das Dateisystem im Arbeitsspeicher ablegt. Es stellt ebenfalls einen Connector für Hadoop zur Verfügung, arbeitet aber mit dem Task-System des Hadoop Ökosystems bevor es dort YARN gab. Ähnlich wie Alluxio stellt es ein eigenes System für Jobs zur Verfügung. Anstelle eines *Hadoop Client* (Job?) der mit JobTracker, Tasktracker und Namenode die „alte" Hadoop Prozessverwaltung darstellt, gibt es einen *Ignite Client*, der direkt mit einem *Ignite Data Node* kommuniziert. Vergleicht man dieses Konzept mit DxramFs und HBase, so müsste man den App Master *RegionServer* im Code von HBase anpassen, und direkt eine Kommunikation mit einem DXRAM Peer machen. In dem Fall könne man vermutlich sowohl das Scheme-Konzept von Hadoop sowie die Idee, für DXRAM ein Dateisystem zu erstellen, aufgeben. Ob jedoch nur der RegionServer von HBase von dieser Änderung betroffen wäre, ist an zu zweifeln.

5.2 Aktueller Stand

Wer an dem Projekt weiter arbeiten möchte, findet alles auf `https://github.com/no-go/HadoopDxramFS`, wo auch dieses Dokument und Notizen sowie Grafiken abgelegt sind. Derzeit ist `HadoopDxramFS` nicht produktiv einsetzbar. Die letzten Versuche im September 2018 DXRAM mit einem 2. Peer laufen zu lassen und eine `FsNode` Datenstruktur auf diesen zu übertragen, scheiterten. Mit einem einzelnen DXRAM Peer war es möglich, zumindest eine leeren Datei in einem Ordner zu hinterlegen. `rename` und `delete` auf Ordnern(!) hatte ebenfalls funktioniert, bis dann ein Upgrade auf DXRAM 0.5.0 kam[3].

Operationen in `DxramOutputStream.java` konnten wegen der Problematik der korrekten Speicherung eines `FsNode` in DXRAM nicht komplette getestet und ausgeführt werden. Dies gilt auch für das Laden eines `Blocks` in den Buffer von `DxramOutputStream` sowie deren `flush()` Methode und der nötigen Änderungen, die durch die DxramFsApp an den Chunks gemacht werden müssen. unmittelbar offen sind daher die Punkte:

[3]Not supporting remove operation if chunk locks are disabled

- korrekte Chunk Speicherung (speziell Arrays und String)
- `create()` und `DxramOutputStream`
- `open()` und `DxramInputStream`
- `DxramFile.getFileBlockLocations()`

5.3 Weitere offene Punkte

Wie bereits (sehr weit oben) erwähnt, gibt es noch Bugs beim Löschen von Chunks und somit auch bei `mv`. Ein Upgrade von dxmem auf 0.5.0 (?) sollten das Problem lösen. Anlegen eines `FsNode` vom Typ `FILE` im Root-Verzeichnis ist zur Zeit nicht möglich. Hier befindet sich wohl noch ein Bug der `DxramFsApp.java` Datei.

Was ebenfalls fehlt:

- Den Typ `EXT` wurde noch nicht überall umgesetzt. Es sind also nicht mehr als `ref_ids_each_fsnode` Dateiblöcke möglich oder mehr Ordnereinträge.
- UTF-8 Datei- bzw. Ordnernamen werden von Hadoop verlangt. Ggf. muss man dazu einfach in DXRAM `StandardCharsets.US_ASCII` in Strings anpassen und dies in `DxramFsConfig.java` angleichen.
- Konkrete Tests mit MapReduce, einer Multinode Konfiguration, HBase Beispiele
- Zu klären: Muss das Scheme auch HBase bekannt gemacht werden?
- YARN und dessen Nutzung des `Statistics` Objekt im FileSystem: Gibt es noch andere Entscheidungen bei der Wahl des `NodeManagers` ?
- Locks (dxram, hadoop, ...) wer übernimmt das, um inkonsistente Zustände auf Blöcken aus zu schließen?
- Mehrere Requests an einen DXNET Peer: Der Code in dem Punkt nicht sicher!
- Hadoop FS Unit Tests

5.4 Fazit und Aussicht

Im Nachhinein wäre es wohl besser gewesen, die FTP Dateisystem Anbindung von Hadoop im ersten Schritt auf eine Multi-Node (hilbert?) System mit `getFileBlocklocations()` um zu bauen und damit konkrete Tests der verteilten Prozessverarbeitung zu machen. Bisher konnte noch keine solche HDFS und Datanode freie Konfiguration getestet werden. Das Erstellen eines eigenen FileSystem Connectors für Hadoop ist nicht so schwer, wenn man bereits ein Dateisystem hat, an das man es zu binden kann. Ausfälle eines verteilten Dateisystems korrekt abfangen und behandeln ist jedoch ein weiterhin offener Punkt. Inwiefern die Prozessverarbeitung von Hadoop den Wegfall von Knoten oder das Blockieren von Dateien oder Blöcken abfedern kann, ist eine weitere offene Frage. Man muss bedenken, dass eine Verarbeitung nur auf Daten stattfindet, die dem zu verarbeitenden Knoten gehören[4], was Blockieren von Chunks/Blöcken überflüssig machen sollte, denn deren Daten gehören dem Prozess exklusiv und nicht mehreren Prozessen (anderer Verarbeitungsknoten = andere Daten).

Sollte eine konkrete HBase Anwendung mit dessen Thrift API arbeiten, so ist für die Zukunft hier ein Entwicklung möglich. Im ersten Schritt könnte man zuerst eine Datenbank mit DXRAM bauen und später Snapshots und Daten (de)kompression beifügen. In dem Fall könne man Hadoop und HBase komplett ersetzen und eine Implementierung eines DXRAM Connectors an Hadoop entfällt. Es gibt bestimmt spezielle Szenarien, in denen der Wegfall von Hadoop und dem original HBase erwünscht ist.

[4] es sei denn, man nutzt auch Replikationen

Eine weitere Lösung wäre, bei der man um die Implementierung eines Dateisystems in Hadoop herum kommen könnte, wenn man im Code des RegionServers jeden Datei oder Block-Zugriff auf DXRAM umleiten würde. Dies setzt jedoch ein tiefes Verständnis des Codes von HBase und dessen Verbindung mit YARN voraus.

6 Anhang

ab	Thema	Stunden
19.10.2017	Einarbeitung HBase & Hadoop, Konzeptsuche zur Einbindung DXRAM in HBase oder Hadoop, Implemetierung eines HDFS- kompatiblen Fake-Dateisystems mit umfangreichen Logging	155
16.3.2018	Konzept zur Einbindung von DXRAM anstelle des Fake-Dateisystems, erste Einbindung von DXNET in Hadoop-Code, Umgang mit Maven	50
17.5.2018	Verbindung Hadoop DxramFs Connector via DXNET mit DXRAM, Entwicklung eines eigenen Deployment-Skripts; DxramFs single Peer kann mit Ordnern bereits: exists, mkdir, list, isdir, size, delete, rename	127
7.9.2018	Anlegen leerer Dateien (create, append) und Recherche bzgl. Daten in BlockLocation sowie dessen Verarbeitung beim RegionServer (HBase, YARN)	40
28.9.2018	Switch auf neues DXRAM, Debugging mit 2. Peer und Problemen bei Chunk Speicherung	57
22.10.2018 bis 31.	Anfertigung Abschlussbericht, Korrekturlesen	30
20.11.2018	Korrekturen im Abschlussbericht	5
	Summe	464