
# Eine Thread-Pool basierte Netzwerkschnittstelle für einen verteilten In-Memory Storage

# A thread-pool-based network interface for a distributed in-memory storage

Masterarbeit

von

Marc Ewert

aus

Ratingen

vorgelegt am

Lehrstuhl für Betriebssysteme
Prof. Dr. Michael Schöttner
Heinrich-Heine-Universität Düsseldorf

9. Februar 2015

# Zusammenfassung

In dieser Masterarbeit wird das existierende Netzwerkmodul vom verteilten In-Memory Storage DXRAM umgestaltet, sodass eine beliebige Anzahl von Threads effektiv eingesetzt werden können.
Ausgehend vom vorhandenen Java Quellcode wird zunächst analysiert, was die Schwächen der bisherigen Implementierung sind. Dazu wird ermittelt, warum in der aktuellen Version kein sinnvolles Multithreading stattfinden kann. Ebenfalls werden langsame Programmabschnitte identifiziert. DXRAM arbeitet hauptsächlich mit sehr kleinen Objekten, weswegen bei einer aktiven Netzwerkkommunikation die einzelnen Programmabschnitte hochfrequentiert durchlaufen werden.
Durch den Einsatz von Java NIO bietet sich eine konsequente Umsetzung des Reactor Patterns an. Diese Umstrukturierung ermöglicht es die Netzwerkkommunikation, unter Zuhilfenahme eines Thread Pools, parallel abhandeln zu können. Um die vielen kleinen Nachrichten effektiv verarbeiten zu können, müssen mehrere Thread Pools eingesetzt werden. In Folge des Reactor Patterns ist die Verarbeitung der Nachrichten von der tatsächlichen Netzwerkkommunikation abgekoppelt. Aus diesem Grund wird eine eigene Flusskontrolle umgesetzt, um so den Empfänger nicht zu überlasten.
Wie vermutet, profitiert das System durch die Änderungen dieser Arbeit. Im Vergleich zur bisherigen Version konnte der Datendurchsatz deutlich gesteigert werden. Aufwändige Methoden wurden umgestaltet oder umgangen. Dadurch konnten auch die Antwortzeiten deutlich verkürzt, und das System insgesamt beschleunigt werden. Die Netzwerkkomponente ist dadurch in der Lage vorhandene Hardware auszulasten. Bei zu kleinen Nachrichten ergeben sich jedoch Einschränkungen. Werden diese in sehr großer Anzahl gesendet, ist das System durch den verwendeten Prozessor limitiert.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Listings

# Kapitel 1

# Einleitung

## 1.1 Motivation

Verteilte Systeme ermöglichen einen logischen Zusammenschluss von vielen autonomen Computern. Dadurch ist es möglich die zur Verfügung stehenden Ressourcen zu bündeln und als Einheit zu präsentieren. Somit ermöglichen Verteilte Systeme die Ausführung komplexer Anwendungen, welche nicht auf einzelnen Computern ausgeführt werden können. Die Grundlage für ein solches verteiltes System bildet die Netzwerkkommunikation. Die Computer müssen sich koordinieren und Daten schnell auf andere Computer verschoben werden können. Neben einer entsprechend leistungsstarken Netzwerkhardware wird dafür eine ebenso leistungsstarke Software benötigt. Diese muss bei Bedarf dazu in der Lage sein die zur Verfügung stehende Bandbreite auszureizen. Für einen effizienten Betrieb ist es ebenfalls wichtig die Antwortzeiten so gering wie möglich zu halten. Die Software muss also ebenso gewährleisten können, dass die Daten schnell verarbeitet werden.

Moderne Prozessoren sind durch ihre Mehrkernarchitektur darauf optimiert mehrere Aufgaben parallel auszuführen. Ebenso sind viele der Arbeitsschritte, welche zur Kommunikation notwendig sind, unabhängig voneinander. Insbesondere gilt das für die Verarbeitung unterschiedlicher Verbindungen. Folglich bietet es sich an diese Eigenschaften auszunutzen und anfallende Aufgaben zur Netzwerkkommunikation parallel von mehre-

ren Threads bearbeiten zu lassen.

## 1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, das vorhandene Netzwerkmodul des verteilten Systems DXRAM dahingehend zu optimieren, dass Daten parallel durch mehrere Threads bearbeitet und versendet werden können. Die bisherige Netzwerkkomponente in DXRAM arbeitet bereits mit mehreren Threads, jedoch hat jeder Thread seinen eigenen Aufgabenbereich, welchen er nicht verlässt oder mit anderen teilt. Dadurch ergibt sich eine Abfolge von Aufgaben, die einander bedingen. Eingehende Nachrichten werden schlussendlich sequenziell abgearbeitet. Um eine parallele Verarbeitung zu ermöglichen muss der Netzwerkcode umgearbeitet und die Abhängigkeiten aufgelöst werden. So soll ermöglicht werden, dass jeder Verarbeitungsschritt parallel für beliebig viele Nachrichten ausgeführt werden kann.
Neben einer effizienteren Nutzung von Multithreading werden auch die Arbeitsschritte im Einzelnen optimiert. Einige Abschnitte greifen auf ineffiziente und teure Aufrufe zurück. Um die allgemeine Leistungsfähigkeit zu erhöhen sollen diese Aufrufe, soweit möglich, umgangen werden.

Aufgrund dieser Probleme ist es mit der aktuellen Version der Netzwerkkomponente nicht möglich vorhandene Netzwerkhardware auszunutzen. Zudem kann es unter Last zu starken Leistungseinbrüchen kommen. Durch die Änderungen in dieser Arbeit soll der Netzwerkcode nicht länger ein limitierender Faktor im Betrieb von DXRAM sein.

## 1.3 Struktur der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die Grundlagen dieser Arbeit erläutert. Dazu wird das DXRAM Projekt kurz vorgestellt. Weiterhin wird die Funktionsweise der verwendeten Java Netzwerkbibliothek erklärt. In Kapitel 3 wird das existierende Netzwerkmodul in seiner bisherigen Ausführung analysiert. Dazu werden kurz die Struktur

und die Arbeitsweise vorgestellt. Anschließend werden die Schwachpunkte der Implementierung aufgezeigt. Kapitel 4 befasst sich mit den Änderungen, welche im Rahmen dieser Arbeit an der Netzwerkkomponente vollzogen wurden. Auf dieser Basis werden in Kapitel 5 die Änderungen ausgewertet. Dabei werden die Versionen vor und nach den Änderungen miteinander verglichen. Abschließend werden die wichtigsten Punkte dieser Arbeit in Kapitel 6 noch einmal zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

# Kapitel 2

# Grundlagen

## 2.1 DXRAM

DXRAM ist ein persistenter In-Memory Storage mit dem Ziel Milliarden von sehr kleinen Objekten zu verwalten. Das Ziel von DXRAM ist es Zwischenlösungen, wie z.B. Memcached[1], zu umgehen. Solche Lösungen werden verwendet um gezielt Daten im Arbeitsspeicher zu halten und so schnellere Zugriffszeiten für langsamere Speichersysteme zu ermöglichen. Ein solches System hat jedoch den Nachteil, dass zwangsläufig nicht alle Daten vorgehalten werden können. Des Weiteren müssen sich die Entwickler selbstständig um die Synchronisierung von Cache und Speicher kümmern. Um diese Probleme anzugehen werden in DXRAM permanent alle Daten im Arbeitsspeicher gehalten. Ein ähnliches Ziel hat auch das RAMCloud[2] Projekt. Während RAMCloud jedoch auf ein Tabellen basiertes Datenmodell setzt, wird in DXRAM ein Key-Value Datenmodell umgesetzt, wobei der typische Eintrag zwischen 16 und 64 Byte groß ist. Dadurch zielt DXRAM insbesondere auf Graphenanwendungen, wie z.B. soziale Netzwerke, ab. Die Persistenz wird über ein transparentes Loggingverfahren realisiert, welches die Eigenschaften von SSDs berücksichtigt. Somit sollen Daten von ausgefallenen Knoten schnell wieder verfügbar sein [KS13].

[1] `http://www.memcached.org/`
[2] `https://ramcloud.stanford.edu/`

### 2.1.1 Speicherverwaltung

Die Daten werden in DXRAM in *Chunks* abgespeichert. Chunks bestehen aus einer 64 Bit ID (kurz *CID*) gefolgt von den binären Daten und bilden somit ein einzelnes Key-Value Paar. Die CID setzt sich wiederum aus der Node ID (kurz *NID*) und einer lokalen ID (kurz *LID*) zusammen. Die NID ist 16 Bit lang und bezieht sich auf den Knoten, welcher den Chunk erzeugt hat. Wichtig ist, dass die Chunks durch Migration auf andere Knoten verteilt werden können. Somit handelt es sich bei der NID nicht zwingend um den Knoten, welcher den Chunk speichert. Die LID ist ein einfacher Zähler, welcher die Anzahl der bisher erzeugten Chunks beinhaltet. Dieser Wert wird von den einzelnen Knoten intern verwaltet und ist somit nur lokal eindeutig. Die Länge der LID beträgt 48 Bit. Ein Knoten kann somit ungefähr 280 Billionen Chunks verwalten. Die Größe der Chunks wird beim Anlegen bestimmt, wobei jedoch eine Maximalgröße konfiguriert werden kann. Dieses Maximum ist für die eigentliche Anwendung nicht bindend. Bei Bedarf können mehrere Chunks in einem *Block* zusammengefasst werden. Somit werden auch große Objekte unterstützt.

Für die Verwaltung verwenden viele In-Memory Systeme Hashtabellen, um so die Daten mit einer Zugriffszeit von $\mathcal{O}(1)$ finden zu können. Hashtabellen haben jedoch den Nachteil, dass ein großer Teil des verwendeten Speichers ungenutzt bleibt, wenn es nur wenig Einträge gibt. Enthält die Hashtabelle zu viele Einträge, werden sie aufgrund von Kollisionen deutlich langsamer. In tabellen-basierten Systemen kommen alternativ häufig baumartig verwaltete Indizes zum Einsatz, welche einen Zugriff in $\mathcal{O}(\log n)$ ermöglichen. Hier kann es jedoch zu Speicherverschwendung kommen wenn Einträge häufig geändert werden. Aus diesem Grund verzichtet DXRAM auf die genannten Ansätze und implementiert stattdessen ein speichereffizientes, hierarchisches Verfahren. Dieses Verfahren ähnelt dem Paging, sodass in $\mathcal{O}(1)$ auf die Daten zugegriffen werden kann. Mittels diesen Verfahrens kann die LID auf die tatsächliche Speicheradresse abgebildet werden. Durch die sequenzielle Verteilung der LID müssen nur Seiten erzeugt und verwaltet werden, welche auch tatsächlich benötigt werden [KBS14]. Dieser Vorgang ist in Abbildung 2.1 zu sehen.

Die Verwaltung solcher Mengen von Objekten ist auch für Speicherallokatoren ein Problem. Abhängig vom verwendeten Speicherallokator und Laufzeitumgebung werden bis

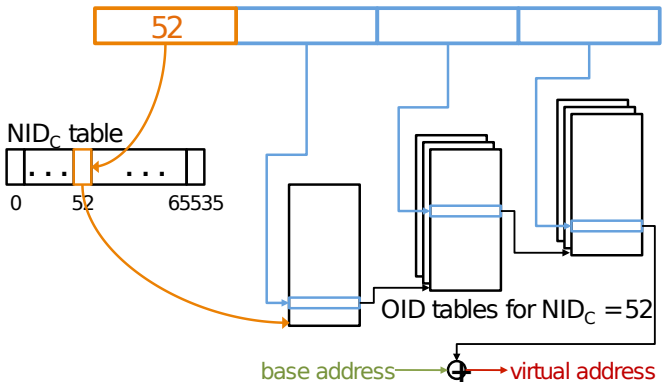



Abbildung 2.1: Paging der lokalen Speicherverwaltung (aus [KS13])

zu 64 Byte zusätzlich für Metadaten des entsprechenden Objekts benötigt. Zusätzlich wird der allozierte Speicher häufig an Adressen ausgerichtet, welche dem Vielfachen von vier oder acht entsprechen. Da dieser zusätzliche Speicherverbrauch bei der angestrebten Menge von sehr kleinen Objekten nicht tragbar ist, wird in DXRAM der Speicher manuell verwaltet. Mit Hilfe der Java *Unsafe* Klasse wird ein Speicherblock alloziert, welcher den Großteil des verfügbaren Arbeitsspeichers belegt. In diesem Speicherblock kann nun ein eigens entwickelter Speicherallokator die Daten ablegen. Dadurch lässt sich der zusätzliche Speicheraufwand auf zwei bis vier Byte pro Objekt reduzieren. Zudem wird nur genau so viel Speicher alloziert, wie auch tatsächlich verwendet wird.

### 2.1.2 Replikation

Dadurch, dass der Datenbestand permanent im flüchtigen Arbeitsspeicher gehalten wird, ist es notwendig, die Daten zu replizieren. Zu diesem Zweck werden in DXRAM SSDs verwendet. Im Vergleich zu traditionellen Festplatten können SSDs einen deutlich höheren Datendurchsatz gewährleisten. Fällt ein Knoten aus, können die Daten dadurch sehr viel schneller wiederhergestellt werden und stehen früher wieder zur Verfügung. Jedoch braucht auch eine SSD mit 500 MB/s für das Auslesen von 32 GB Daten länger als eine Minute. Die Backups eines Knotens sollten also auf mehreren verschiedenen Knoten gespeichert werden, sodass der Wiederherstellungsprozess verteilt ausgeführt werden kann. Die Standardkonfiguration sieht vor, dass jeder Chunk sowohl auf dem lokalen, als

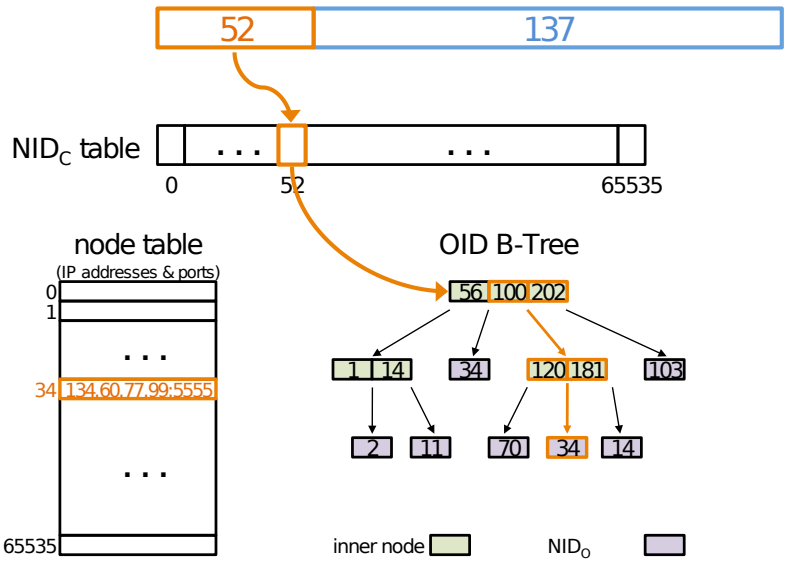



Abbildung 2.2: Metadatenverwaltung im Super-Peer (aus [KS13])

auch auf drei entfernten Knoten repliziert wird. Um ein geordnetes Herunterfahren von DXRAM zu ermöglichen wird der gesamte virtuelle Speicherblock lokal auf der SSD gespeichert. Dadurch, dass in DXRAM keine Speicheradressen gespeichert werden, sondern nur Offset-Pointer, kann der Speicherblock beim Hochfahren direkt wieder in den Arbeitsspeicher geladen werden.

### 2.1.3 Super-Peer Overlay

Zur Verwaltung der Metadaten kommt in DXRAM ein Super-Peer Overlay zum Einsatz. Bis zu 10% der eingesetzten Knoten übernehmen die Rolle eines Super-Peers. Diese ausgewählten Knoten speichern selber keine Objekte, sondern haben eine Übersicht darüber, welche der ihnen zugeteilten Knoten welche Objekte beheimatet. Die Zuordnung der IDs auf die Knoten erfolgt dabei über ID Bereiche. Eine direkte Abbildung jeder einzelnen Chunk ID auf die Knoten ID wäre zu speicherintensiv. Diese Bereiche werden innerhalb eines B-Baums verwaltet, da dieser bereits mit Hilfe von Wertebereichen arbeitet und für eine effiziente Suche optimiert ist.

Um zu bestimmen, welche Knoten welchem Super-Peer zugeordnet sind, ist das Super-

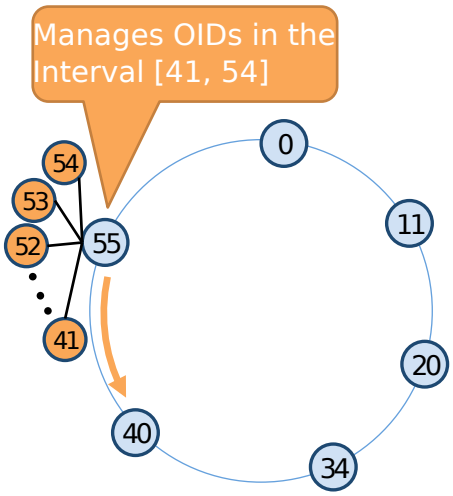



Abbildung 2.3: Super-Peer Overlay (aus [KS13])

Peer Overlay ähnlich wie Chord aufgebaut. Das bedeutet, dass ein Super-Peer alle Knoten verwaltet, deren ID sich zwischen seiner, und der ID seines Vorgängers liegen. Die Super-Peers kennen sich untereinander und können bei Bedarf eine Verbindung aufbauen.

Das Super-Peer Overlay dient jedoch nicht nur allein zur Suche von Objekten. Sollte ein Knoten ausfallen wird der zuständige Super-Peer benachrichtigt. Dieser verwaltet anschließend den Wiederherstellungsprozess der Daten, welche der ausgefallene Knoten gespeichert hatte. Dazu werden alle Knoten kontaktiert, die ein Backup der verlorenen Daten gespeichert haben. Diese Backups werden von den angesprochen Knoten von ihrer SSD in den Arbeitsspeicher geladen und stehen damit wieder zur Verfügung. Bei Bedarf können diese Daten anschließend auf andere Knoten migriert werden, um so eine bessere Lastverteilung zu gewährleisten. Um einen Ausfall eines Super-Peers abfangen zu können, werden die Metadaten eines einzelnen Super-Peers auf seinen direkten Nachbarn im Super-Peer Overlay gespeichert.

## 2.2 Java NIO

Die *Java New I/O* (kurz: Java NIO) Bibliothek wurde als neuartiger Weg für I/O Operationen entwickelt. Das Ziel war es, eine schnelle Alternative zu den traditionellen APIs aus den Java IO und Java Networking Bibliotheken zu bieten. Eingeführt wurde Java NIO in der Java Version 1.4. Eine der Hauptneuheiten ist dabei die Möglichkeit Operationen nicht-blockierend auszuführen. Dadurch ist es nicht mehr länger notwendig, dass jede Verbindung von einem eigenen Thread überwacht werden muss. Somit kann ein einzelner Thread eine Vielzahl von Verbindungen bedienen, womit vorhandene Ressourcen deutlich effizienter genutzt werden können.

### 2.2.1 Buffer und Channel

Das grundlegende Objekt für den Datenaustausch mittels Java NIO stellt der *Channel*[3] dar. Ein Channel repräsentiert eine Verbindung zu einem beliebigen Kommunikationspartner. Der Channel tritt in folgenden Ausprägungen auf:

- SocketChannel
- ServerSocketChannel
- DatagramChannel
- FileChannel

Wie zu sehen ist, gibt es sowohl Channels für die Netzwerkkommunikation mittels TCP, als auch mittels UDP. Zusätzlich können Daten über einen Channel auch an Dateien übertragen werden. Der Channel erinnert an die Klassen `InputStream` und `OutputStream` von Java. Die Übertragung funktioniert beim Channel jedoch mit Hilfe von textitBuffer Objekten. Channels können Daten aus einem Buffer lesen oder in einen Buffer reinschreiben. Buffer gibt es ebenfalls in Ausprägungen für jeden Basisdatentyp, welcher

[3] `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/package-summary.html`

```
RandomAccessFile aFile = new RandomAccessFile("nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);

while (bytesRead != -1) {
    buf.flip();

    while (buf.hasRemaining()){
        System.out.print((char) buf.get());
    }

    buf.clear();
    bytesRead = inChannel.read(buf);
}
```

Listing 2.1: Beispiel zu Buffer und Channels (aus [Jen13])

in Java verfügbar ist. Für die allgemeine Verwendung bietet sich dabei der `ByteBuffer`[4] an. Dieser bietet ebenfalls Methoden zum Lesen und Schreiben der verschiedenen Datentypen an.

Ein Buffer besitzt im Kern ein Array in dem die Daten gespeichert werden. In Java NIO wird der Buffer jedoch nicht wie ein Array verwendet, sondern ist zustandsbasiert. Der Zustand wird durch die Felder `position`, `limit` und `capacity` bestimmt. Das Feld `capacity` gibt die maximale Größe des Buffers an. Im Falle einer `ByteBuffer` Instanz entspricht der Wert der maximalen Anzahl an Bytes, welche im Buffer abgelegt werden können. Mit dem Feld `limit` wird angegeben, wie viel Platz der Buffer noch besitzt. Hier muss unterschieden werden, ob vom Buffer gelesen wird, oder ob der Buffer schreibend verwendet wird. Wird der Buffer gelesen entspricht der `limit` Wert der Anzahl an Daten die noch gelesen werden können. Beim Schreiben spiegelt der Wert wider, wie viele Daten noch aufgenommen werden können. Das `position` Feld gibt schlussendlich an, welcher Eintrag im Array als nächstes gelesen oder beschrieben wird [Tra03].

Nach der Erzeugung ist der Buffer im schreibenden Zustand und muss mit Daten befüllt werden. Möchte man in den lesenden Zustand wechseln muss man die `flip()` Methode des Buffers aufrufen. Durch diese Methode wird zum einen das Limit auf die aktuelle

[4] `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html`

Position gesetzt, zum anderen wird die Position im Anschluss auf 0 gesetzt. Nun kann der Buffer sequenziell ausgelesen werden. Wichtig ist, dass die `flip()` Methode manuell aufgerufen werden muss, bevor ein Buffer an einen Channel übergeben werden kann. Andernfalls werden keine Daten aus dem Buffer gelesen.

Listing 2.1 zeigt exemplarisch die Verwendung von Channels in Verbindung mit Buffern. Zuerst wird mit Hilfe der Klasse `RandomAccessFile` auf eine Datei zugegriffen. Der Datenaustausch findet über einen `FileChannel` statt, welcher beim Erzeugen ebenfalls mit angelegt wird. Anschließend wird ein `ByteBuffer` alloziert. Dieser `ByteBuffer` wird an die `read()` Methode des Channels übergeben, welcher die Daten aus der Datei in den Buffer schreibt. Der Buffer wird solange befüllt, bis entweder die Datei vollständig ausgelesen wurde, oder der Buffer voll ist. Nach dem Befüllen wird der Buffer über die `flip()` Methode in den lesenden Zustand versetzt und Stückweise ausgelesen. Mit Hilfe der `clear()` Methode wird der Buffer in seinen Ausgangszustand zurückgesetzt. Das heißt, dass die Position auf 0 und das Limit auf den Wert der Maximalkapazität gesetzt wird. Der Buffer ist somit wiederverwendbar. Der Inhalt wird durch die `clear()` Methode jedoch nicht gelöscht. Dieser wird durch folgende Operationen lediglich überschrieben. Die Prozedur setzt sich so lange fort, bis die Datei vollständig ausgelesen wurde.

Dieses Beispiel lässt sich genau so auf alle anderen Channel und Buffer Implementierungen anwenden. Die für diese Arbeit relevanten SocketChannel lassen sich auf die gleiche Weise ansprechen wie der gezeigte FileChannel. Unterschiedlich ist lediglich die Erzeugung. Das Schreiben in einen Channel funktioniert analog zum Lesen. Anstelle der `read()` Methode wird dem Channel über die `write()` Methode ein Buffer übergeben. Dieser wird im Voraus mit Hilfe der `put()` Operation mit Daten befüllt.

### 2.2.2 Selector

Um mehrere Channels mit einem einzelnen Thread überwachen zu können stellt Java NIO die Klasse *Selector*[5] bereit. Ein Selector lässt sich bei verschiedenen Channels re-

[5] `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/Selector.html`

```
Selector selector = Selector.open();

ServerSocketChannel channel = ServerSocketChannel.open();
channel.configureBlocking(false);
ServerSocket socket = channel.socket();
socket.bind(new InetSocketAddress(port));

SelectionKey key = channel.register(selector, SelectionKey.OP_ACCEPT);

while (true) {
    int readyChannels = selector.select();

    if (readyChannels == 0) continue;

    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

    while (keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();

        if(key.isAcceptable()) {
            // a connection was accepted by a ServerSocketChannel.
        } else if (key.isConnectable()) {
            // a connection was established with a remote server.
        } else if (key.isReadable()) {
            // a channel is ready for reading
        } else if (key.isWritable()) {
            // a channel is ready for writing
        }
        keyIterator.remove();
    }
}
```

Listing 2.2: Beispiel zum Selector (aus [Jen13])

gistrieren, sodass dieser bei bestimmten Ereignissen informiert wird. Diese Ereignisse lassen sich anschließend gebündelt von einem Thread abrufen, welcher diese dann abarbeiten kann. Der Selector ist hauptsächlich für die Verwendung in Kombination mit den verschiedenen Channels für die Netzwerkkommunikation vorgesehen.

In Listing 2.2 ist der grundlegende Ablauf für die Verwendung eines Selectors dargestellt. Zuerst wird ein `ServerSocketChannel` erzeugt, welcher auf einem Port auf eingehende Verbindungen wartet. Anschließend wird der Selector bei dem Channel registriert. Der Channel muss dabei zwingend im nicht-blockierenden Modus arbeiten. Weiterhin

muss bei der Registrierung angegeben werden für welche Ereignisse der Selector zuständig ist. Mögliche Ereignisse sind *accept*, *connect*, *read* und *write*. Es lassen sich auch mehrere Ereignisse gleichzeitig registrieren. Die Werte für die möglichen Ereignisse sind als Bitfeld gespeichert. Sie lassen sich also mit der bitweisen `ODER` Operation verknüpfen.

Als Rückgabewert der `register()` Methode wird ein `SelectionKey`[6] zurück gegeben. Der `SelectionKey` ist ein Token, welches die Registrierung symbolisiert, und lässt sich für die eigene Verwendung ablegen. Mit dem `SelectionKey` erhält man Zugriff auf den entsprechenden Channel und Selector. Weiterhin lässt sich am `SelectionKey` ablesen, ob die Verbindung lesebereit ist, oder andere Ereignisse aufgetreten sind. Die `SelectionKey` Referenzen bleiben für den restlichen Verbindungsablauf erhalten und können über die `attach()` Methode noch ein individuelles Objekt zugewiesen bekommen. Darüber lassen sich weitere Informationen der Verbindung zuordnen, die für die eigene Verarbeitung hilfreich sind.

Für die Handhabung der Channels wird auf dem Selector die `select()` Methode aufgerufen, welche die Ereignisse der Channels bündelt. Die `select()` Methode gibt die Anzahl der aufgetretenen Ereignisse zurück. Die `selectedKeys()` Methode hingegen liefert eine Auflistung aller `SelectionKeys` zurück, welche bereit zur Verarbeitung sind. Mit Hilfe der `SelectionKey` Objekte erhält man die dazu gehörigen Channels und kann über diese entsprechend lesen oder schreiben.

Das Abarbeiten der `SelectionKey` Objekte ist jedoch nicht frei von Fallstricken. Zum einen ist der `select()` Aufruf der hier verwendeten Variante blockierend. Der Selector wartet also, bis mindestens ein `SelectionKey` bereit zur Verarbeitung ist, bevor der Aufruf abgeschlossen wird. Wichtig ist hierbei, dass während dieser Zeit nur eingeschränkt mit den Channels interagiert werden kann, welche beim Selector registriert sind. Alle Operationen am Channel, die ein Ereignis an den Selector melden würden, werden ebenfalls blockiert, bis der `select()` Befehl wieder freigegeben wird. Dadurch kann es auch zu einem Deadlock kommen. Ein Thread wartet im `select()` Befehl auf neue `SelectionKey` Objekte während ein weiterer Thread Daten über die Verbindung

[6] `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/SelectionKey.html`

senden möchte. Um zu Senden muss jedoch der entsprechende Channel zum Schreiben markiert werden. Dieser Aufruf wird jedoch durch den `select()` Befehl blockiert. Um eine solche Situation zu vermeiden gibt es auch einen nicht-blockierenden `select()` Aufruf, welcher sofort zurückkehrt. Verwendet man dennoch den blockierenden Aufruf und muss auf einem Channel arbeiten, lässt sich die `wakeUp()` Methode am Selector aufrufen, um den wartenden Thread manuell freizugeben.

Weiterhin bemerkenswert ist, dass ein `SelectionKey` mehrfach in der Auflistung vom Selector vorkommen kann. Ist eine Verbindung gleichzeitig bereit Daten zu lesen und zu schreiben, kommt dieselbe Referenz zweimal in der Auflistung vor. Dieser Fall muss entsprechend bedacht werden. Weiterhin muss der `SelectionKey` nach abgeschlossener Bearbeitung manuell aus der Liste entfernt werden. Wird dies nicht gemacht, wächst die Liste mit der Zeit immer weiter an, weil vom Selector stets dieselbe Referenz wiederverwendet wird.

# Kapitel 3

# Analyse

Diese Masterarbeit baut auf dem bereits vorhandenen Netzwerkmodul von DXRAM auf, welches durch Umstrukturieren und Erweitern verbessert werden soll. Aus diesem Grund ist es sinnvoll zuerst die Basis vorzustellen und zu analysieren. In diesem Kapitel wird die bisherige Implementierung vorgestellt und vorhandene Schwächen aufgezeigt.

## 3.1 Bisherige Implementierung

Die derzeitige Implementierung setzt bereits eine Netzwerkkommunikation mittels *Java NIO* um. Um die Arbeitsweise des Moduls zu verstehen werden die wichtigsten Klassen folgend vorgestellt und deren Zusammenhang erläutert. Der prinzipielle Aufbau des Netzwerkmoduls wird in Abbildung 3.1 dargestellt.

### 3.1.1 NetworkHandler

Einstiegspunkt der Netzwerkkomponente ist der `NetworkHandler`, welcher die zentralen Funktionalitäten zur Kommunikation über das Netzwerk nach außen hin bereitstellt. Der `NetworkHandler` kann über das `NetworkInterface` angesprochen werden, welches in Listing 3.1 zu sehen ist.

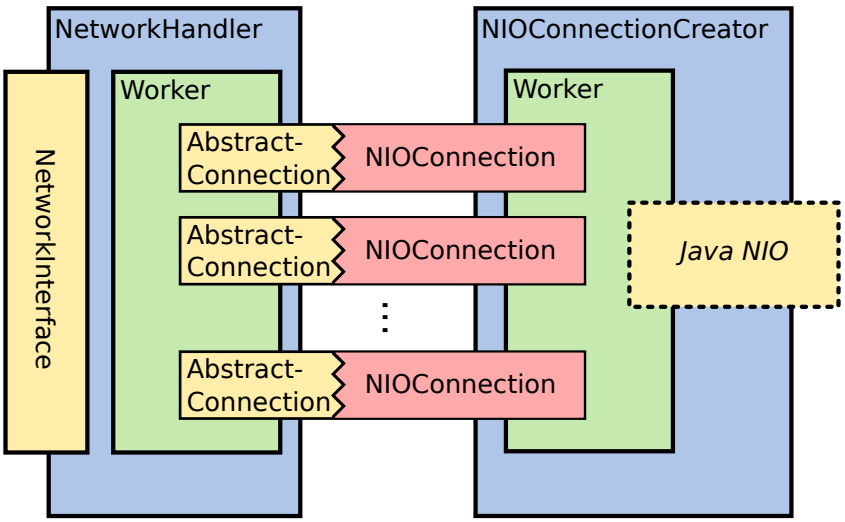



Abbildung 3.1: Struktur der bisherigen Implementierung

Der erste Teil des Funktionsumfangs stellt das Senden von Nachrichtenobjekten dar. Der zweite Teil ist die Möglichkeit, eine Klasse als Empfänger für bestimmte Nachrichtentypen zu registrieren, oder entsprechend die Registrierung rückgängig zu machen. Weiterhin ist es möglich den `ConnectionManager` zu aktivieren, bzw. zu deaktivieren. Ist der `ConnectionManager` aktiv, wird nur eine bestimmte Anzahl von Verbindungen gleichzeitig aufrecht erhalten. Wird dieser Schwellenwert überschritten initiiert der `ConnectionManager` die Schließung der am wenigsten genutzten Verbindung. Im deaktivierten Zustand gibt es keine Beschränkung der Anzahl aktiver Verbindungen durch die Software. Durch das `CoreComponent` Interface erbt der `NetworkHandler` noch die Aufrufe zur Initialisierung und Schließung der Netzwerkkomponente.

Der `NetworkHandler` stellt jedoch nicht nur die Grundfunktionen zur Verfügung. Weiterhin ist er für die Deserialisierung der Nachrichtenobjekte, sowie deren Auslieferung verantwortlich. Hierzu hat der `NetworkHandler` einen *Worker* implementiert, welcher in einem eigenen Thread läuft und darauf wartet, dass für eine Verbindung neue Daten zur Verfügung stehen. Sobald neue Daten gelesen wurden, werden diese an den entsprechenden `IncomingMessageCreator` weitergeleitet und dort verarbeitet. Liegt ein vollständiges Nachrichtenobjekt vor, wird dieses an die dafür registrierten Klassen ausgeliefert.

```
public interface NetworkInterface extends CoreComponent {

    void activateConnectionManager();

    void deactivateConnectionManager();

    void sendMessage(AbstractMessage p_message) throws
       NetworkException;

    void register(Class<? extends AbstractMessage> p_message,
       MessageReceiver p_receiver);

    void unregister(Class<? extends AbstractMessage> p_message,
       MessageReceiver p_receiver);

    public interface MessageReceiver {
        void onIncomingMessage(AbstractMessage p_message);
    }
}
```

Listing 3.1: NetworkInterface

#### IncomingMessageCreator

Innerhalb der `NetworkHandler` Klasse ist der `IncomingMessageCreator` implementiert. Dieser wird verwendet um den eingehenden Datenstrom einzulesen und daraus Nachrichtenobjekte zu erzeugen. Jede Verbindung hat dabei ihre eigene Instanz.

Der `IncomingMessageCreator` ist im Kern ein Zustandsautomat, welcher mit Byte Daten versorgt wird. Das Ziel dabei ist es, den Nachrichtenstrom in zusammenhängende Daten zu unterteilen, sodass sich ein einzelnes Nachrichtenobjekt erzeugen lässt. Zuerst wird dabei die Anzahl der Headerbytes eingelesen. Mit dieser Information lassen sich die darauf folgenden Bytes als eigentlicher Nachrichtenheader interpretieren. Sobald der Header vollständig ist, wird aus den vorliegenden Daten ein Nachrichtenobjekt mit den entsprechenden Werten erzeugt.
Anschließend wird eingelesen wie viele Bytes der eigentliche Payload umfasst. An diesem Punkt ist die Gesamtgröße der Nachricht bekannt. Sind alle Bytes eingelesen worden, werden die bisher unverarbeiteten Bytes an das erzeugte Nachrichtenobjekt übergeben, welches selbstständig seine Nutzdaten wiederherstellt.

### 3.1.2 ConnectionManager

Im `ConnectionManager` werden alle aktiven Verbindungen verwaltet. Wird eine Nachricht zu einem anderen Knoten gesendet, wird die entsprechende Verbindung im `ConnectionManager` angefordert. Hat dieser keine offene Verbindung zum geforderten Zielknoten, wird eine aufgebaut. Der Verbindungsaufbau erfolgt dabei mit Hilfe der `NIOConnectionCreator` Klasse.

Wenn der `ConnectionManager` über das `NetworkInterface` aktiviert wurde, beschränkt dieser die Anzahl gleichzeitig offener Verbindungen auf einen vorab festgelegten Wert. Wenn diese Grenze erreicht ist und eine weitere Verbindung hinzukommt, wird anhand eines *Ratings* entschieden, welche Verbindung geschlossen wird. Dazu wird die Verbindung gewählt, welche das geringste Rating aufweist. Das Rating einer Verbindung wird durch die Verbindungen selbst anhand der Anzahl empfangener Nachrichten bestimmt. Dabei ist es jedoch auch möglich einzelnen Nachrichten eine abweichende Gewichtung zuzuteilen. Wird die individuelle Gewichtung der Nachrichten nicht genutzt, wird standardgemäß die Verbindung mit dem geringsten Nachrichtenaufkommen geschlossen. Die Größe der Nachrichten findet hierbei keine Berücksichtigung.

### 3.1.3 NIOConnectionCreator

Wie der Name der Klasse `NIOConnectionCreator` vermuten lässt werden mit Hilfe dieser Klasse Verbindungen zu anderen Knoten aufgebaut. Weiterhin ist in dieser Klasse der gesamte Verbindungsablauf mittels Java NIO implementiert. Um dies umzusetzen ist im `NIOConnectionCreator` ein weiterer *Worker* implementiert, welcher ebenfalls in einem eigenen Thread arbeitet. Die eigentlichen Verbindungen werden durch die `NIOConnection` Klasse dargestellt.

### NIOConnection

Die Klasse *NIOConnection* repräsentiert eine Verbindung zu einem anderen Knoten im Netzwerk. Um vom `NetworkHandler` verwendet werden zu können, wird die abstrakte Oberklasse `AbstractConnection` erweitert.
Die `AbstractConnection` stellt dabei allgemeine Verbindungsinformationen bereit, wie etwa die Node ID des Verbindungspartners oder ein Rating, welches dem `ConnectionManager` als Entscheidungshilfe dient. Zusätzlich stellt sie Methoden zum Lesen und Schreiben auf die Verbindung bereit. Um dem `NetworkHandler` mitzuteilen, wann die Verbindung bereit zum Lesen ist, hält diese einen Delegate zum `NetworkHandler`. Dieser wird aufgerufen, sobald die Verbindung neue Daten empfangen hat.

Die `NIOConnection` ist auf die Verwendung mit Java NIO angepasst. Zum einen hält die Klasse einen Pointer auf den `SocketChannel`, damit das Objekt korrekt auf die entsprechende Java NIO Verbindung abgebildet werden kann. Des Weiteren hat jede Verbindung jeweils eine verkettete Liste für ein- und ausgehende Buffer. In diesen Listen werden die serialisierten Nachrichtenobjekte verwaltet. Werden durch den `NetworkHandler` neue Daten zum Versenden eingehangen, signalisiert die `NIOConnection` dem Worker, dass neue Daten zum Senden bereit stehen.

### Worker

Der *Worker* der Klasse `NIOConnectionCreator` realisiert die eigentliche Netzwerkkommunikation. Dieser wartet darauf, dass neue Daten am Selector anliegen und arbeitet diese ab. Beim Empfangen von Daten werden diese in einen neuen `ByteBuffer` geschrieben und der eingehenden Liste der entsprechenden `NIOConnection` hinzugefügt. Ist eine Verbindung bereit zum Schreiben, ruft der Worker solange die ausgehende Liste ab, bis keine weiteren Buffer mehr vorhanden sind und übergibt diese an Java NIO. Weiterhin ist der Worker dafür zuständig, Verbindungen über Java NIO aufzubauen oder zu schließen. Verbindungen, welche nicht vom lokalen Knoten initiiert wurden, werden dabei dem `ConnectionManager` gemeldet. Ebenso werden geschlossene Verbindungen gemeldet.

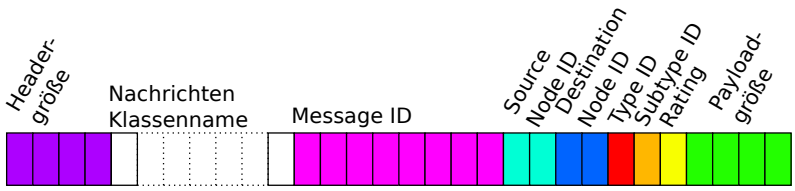



Abbildung 3.2: Bisheriger Nachrichtenheader

### 3.1.4 AbstractMessage

Die Kommunikation mit dem Netzwerkmodul erfolgt ausschließlich mit Klassen, die von der abstrakten Oberklasse `AbstractMessage` erben. In dieser Klasse sind alle notwendigen Informationen implementiert, welche zum Versenden benötigt werden. Dazu gehören folgende Felder:

- *Message ID*: Lokal eindeutige Identifikationsnummer der aktuellen Nachricht
- *Source Node ID*: Knoten, von dem die Nachricht ausgeht
- *Destination ID*: Zielknoten, an den die Nachricht übermittelt wird
- *Type*: Nachrichtentyp
- *Subtype*: Erweiterter Subtyp
- *Rating*: Gewichtung der Nachricht

Die Klasse `AbstractMessage` ist für ihre Serialisierung und Deserialisierung selbst verantwortlich. Es stehen entsprechende Methoden zur Verfügung, welche die Nachricht in Byteform zurück geben. Umgekehrt ist es möglich aus einem Byte Array wieder ein Nachrichtenobjekt erzeugen zu lassen.

Der gesamte Nachrichtenheader ist in Abbildung 3.2 zu sehen. Jedes Feld der Abbildung repräsentiert ein vollständiges Byte. Die Ausnahme dabei bildet der Klassenname. Dieses Feld beinhaltet eine Zeichenkette, welche den vollständigen Java Klassennamen des Nachrichtenobjekts darstellt. Dementsprechend kann dieser Teil des Nachrichtenheaders

beliebig lang sein. Der Klassenname ist notwendig weil die Nachrichten mithilfe der Java Reflection API[1] deserialisiert werden. Diese ermöglicht das Erzeugen von Java Objekten aus Zeichenketten zur Laufzeit.

Im Anschluss an den Header folgen die eigentlichen Nutzdaten, die übermittelt werden sollen. Dazu müssen alle Klassen, welche die `AbstractMessage` erweitern, entsprechende Methoden implementieren um die Nutzdaten lesen und schreiben zu können. Wie dies genau umgesetzt wird ist von der Implementierung abhängig.

Zum Deserialisieren stellt `AbstractMessage` zusätzlich die Klasse *PayloadHandler* zur Verfügung. In dieser Klasse können die empfangen Bytes für den Nachrichtenpayload zwischengespeichert werden. Wenn die Payload-Bytes vollständig sind, können diese geschlossen an die Nachricht weitergereicht und wiederhergestellt werden.

#### AbstractRequest und AbstractResponse

Neben normalen Nachrichten gibt es einen weiteren Nachrichtentyp, welcher zwingend eine Antwort vom Empfänger erwartet. Durch das Senden einer Nachricht, welche die abstrakte Oberklasse *AbstractRequest* erweitert, lassen sich benötigte Informationen anfordern. Diese Anfrage kann durch den Empfänger mittels einer von *AbstractResponse* abgeleiteten Nachricht beantwortet werden. Die Antwort kann von der Netzwerkkomponente direkt der Anfrage zugeordnet werden.

Es ist möglich die AbstractRequest Nachricht synchron oder asynchron, mittels eines Delegate, zu versenden. Im synchronen Fall wird der sendende Thread solange blockiert bis die Antwort eingetroffen ist. Andernfalls wird der Delegate ausgeführt.

[1] `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html`

## 3.2 Limitierungen

Auch wenn die bisherige Implementierung fehlerfrei funktioniert stößt sie doch relativ früh an ihre Grenzen. Insbesondere bei hoher Last machen sich die limitierenden Faktoren bemerkbar. Diese Faktoren sollen im folgenden Abschnitt aufgezeigt werden.

### 3.2.1 Serialisierung

Bei der Serialisierung der Nachrichten werden die Nachrichten häufig umkopiert. Das führt insbesondere bei sehr vielen kleinen Nachrichten, welche bei DXRAM den Hauptteil darstellen, zu einer sehr hohen Belastung der *Garbage Collection*.

In Listing 3.2 ist die Methode zum Senden von Nachrichten in der Klasse `NetworkHandler` zu sehen. Der aufgeführte Quellcode zeigt einen geschachtelten Aufruf, bei dem zuerst eine Byte Darstellung der Nachricht angefordert wird. Anschließend wird das resultierende Byte Array in einen `ByteBuffer` umkopiert, bevor es schlussendlich der Verbindung übergeben wird.

Der Hauptgrund für die Vielzahl an Kopiervorgängen ist jedoch in Listing 3.3 abgebildet. Dort wird zum Serialisieren der Nachrichten auf die `ByteArrayOutputStream` Klasse zurückgegriffen. Im Hintergrund arbeitet diese Klasse auf Basis dynamisch wachsender Arrays[2]. Somit wird das zugrunde liegende Array bei Schreibvorgängen regelmäßig umkopiert.
Sind alle Daten an den `ByteArrayOutputStream` übergeben worden, wird das vollständige Byte Array angefordert. Wie in Listing 3.4 zu sehen, kopiert die `ByteArrayOutputStream` Klasse zu diesem Zweck das Array erneut in ein neues Array mit finaler Größe.

[2] `http://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html`

```
public void sendMessage(final AbstractMessage p_message) throws
    NetworkException {
    /* ... */
    connection.write(ByteBuffer.wrap(p_message.getBytes()));
    /* ... */
}
```

Listing 3.2: sendMessage() in *NetworkHandler*

```
public final byte[] getBytes() {
    byte[] ret = new byte[0];
    ByteArrayOutputStream out;
    DXRAMOutputStream stream;
    /* ... */
    try {
        out = new ByteArrayOutputStream();
        stream = new DXRAMOutputStream(out);
        /* ... */
        ret = out.toByteArray();
        /* ... */
    } catch (final IOException e) { /* ... */ }
    return ret;
}
```

Listing 3.3: getBytes() in *AbstractMessage*

```
public synchronized byte[] toByteArray() {
    return Arrays.copyOf(buf, count);
}
```

Listing 3.4: toByteArray() in *ByteArrayOutputStream*

Somit gibt es folgende Kopiervorgänge:

1. Zur Übergabe an die Verbindung
2. Potenziell bei jedem Schreibvorgang
3. Beim Finalisieren des Byte Arrays

Zusätzlich zu den häufigen Kopiervorgängen werden auch bei jedem Sendevorgang neue Instanzen der Klasse `ByteArrayOutputStream` und `DXRAMOutputStream`[3] erzeugt, was ebenfalls die Garbage Collection zusätzlich belastet. Effektiv müssen im Rah-

[3]Die Klasse `DXRAMOutputStream` ist ein Wrapper für andere `OutputStream` Klassen, erweitert um Aufrufe zum Schreiben von bspw. Node IDs

men des Serialisierungvorgangs mindestens vier Objekte von der Garbage Collection bereinigt werden: Die beiden `OutputStream` Objekte und mindestens zwei Byte Arrays. Der verwendete Buffer wird am Ende an Java NIO übergeben und gesendet. Erst danach wird er für die Garbage Collection frei gegeben.

### 3.2.2 Deserialisierung

Wie auch schon bei der Serialisierung wird beim Deserialisieren der Nachrichten mit stream-basierten Klassen gearbeitet, namentlich `DXRAMInputStream` und `ByteArrayInputStream`. Diese arbeiten äquivalent zu ihren `OutputStream` Gegenstücken auf dynamisch wachsenden Arrays.

Die Deserialisierung wird dabei jedoch in zwei Schritte unterteilt. Im ersten Schritt wird das entsprechende Nachrichtenobjekt instanziiert und die Headerdaten wiederhergestellt. Dazu werden zunächst alle Headerfelder aus dem `InputStream` gelesen und zwischengespeichert. Anschließend wird das Objekt erzeugt und die Daten übertragen. Somit zeigt sich auch hier doppelter Kopieraufwand beim Lesen des Headers.

Im zweiten Schritt wird der Payload der Nachricht mittels der Klasse `PayloadHandler` ausgelesen und wiederhergestellt. Für jede empfangene Nachricht muss jedoch ein eigener `PayloadHandler` instanziiert werden. Damit die Nachricht ihre Daten wiederherstellen kann, muss vom `PayloadHandler` der Nachricht eine Instanz der `DXRAMInputStream` Klasse übergeben werden. Der `DXRAMInputStream` benötigt wiederum einen `ByteBufferInputStream`. Diese Instanzen werden nicht wiederverwendet und müssen somit für jede einzelne Nachricht neu erzeugt werden.

Im Rahmen der Deserialisierung werden also mindestens fünf Objekte alloziert, welche von der Garbage Collection wieder aufgeräumt werden müssen. Hinzu kommen noch die zugrunde liegenden Arrays der Stream-Instanzen, welche ebenfalls potenziell bei jedem Zugriff umkopiert werden müssen.

Ein weiteres Problem zeigt sich in Listing 3.5. Die abgebildete Methode wird verwendet um eine neue Instanz des aktuell vorliegenden Nachrichtenobjekts zu erzeugen. Wie

```
private static AbstractMessage getInstance(final String p_className)
    throws NetworkException {
     AbstractMessage ret;
     Class<?> c;
     Constructor<?> constructor;
     boolean accessible;
     try {
         c = Class.forName(p_className);
         constructor = c.getDeclaredConstructor();
         accessible = constructor.isAccessible();
         constructor.setAccessible(true);
         ret = (AbstractMessage)constructor.newInstance();
         constructor.setAccessible(accessible);
     } catch (final Exception e) { /* ... */ }
     return ret;
}
```

Listing 3.5: getInstance() in *AbstractMessage*

hier zu sehen ist wird das Objekt, wie in Unterabschnitt 3.1.4 beschrieben, mittels der Java Reflection API erzeugt. Problematisch dabei ist, dass der Zugriff auf die Reflection API, je nach Testszenario, um etwa 800 % langsamer ist als der konventionelle Weg eine Klasse zu instanziieren [TOR13]. Da diese Methode für jede eingehende Nachricht aufgerufen wird wirkt sich dies negativ auf die effektive Laufzeit aus.

### 3.2.3 Nachrichtenheader

Ein großes Problem für den Netto Datendurchsatz ist der, relativ betrachtet, sehr große Nachrichtenheader, welcher in Abbildung 3.2 zu sehen ist. Dieser besteht aus mindestens 23 Byte, wobei dort noch der vollständige Java Klassenname des verwendeten Nachrichtenobjekts hinzukommt. Der Java Klassenname bildet sich aus dem eigentlichen Namen der Klasse, sowie dem Java Package, in welchem diese Klasse liegt.

Nimmt man als Beispiel die Klasse *de.uniduesseldorf.dxram.core.net.AbstractMessage* und geht von einer UTF-8 Kodierung aus, werden allein 48 Byte für den Klassennamen verwendet. Zusammengerechnet kommt man somit auf einen 71 Byte Nachrichtenheader für jede einzelne Nachricht. Zieht man die erwartete Nutzdatengröße von 16 bis 64 Byte heran, ergibt sich ein Mehraufwand von ca. 444 % bei 16 Byte, und ca. 111 % bei 64

Byte. Um den gleichen Faktor verringert sich folglich auch der Netto Datendurchsatz, welcher effektiv übertragen werden kann.

### 3.2.4 Multi Threading

Wie Abschnitt 3.1 zu entnehmen ist, arbeiten in der bisherigen Implementierung zwei separate Threads, welche primär beim Empfangen der Nachrichten zum Einsatz kommen. Das Problem dabei ist, dass die Threads statisch an ihre Aufgaben gebunden sind. So ist es nicht möglich die Anzahl der Threads zu ändern. Es können weder mehr, noch weniger Threads eingesetzt werden. Ebenso können die verwendeten Threads nicht aus ihrem Aufgabengebiet ausbrechen, um so etwa dynamisch auf Last zu reagieren. Es ist also möglich, dass ein Thread voll ausgelastet ist, während der Zweite keine Aufgaben vorliegen hat.

#### Worker im NIOConnectionCreator

Der Worker im `NIOConnectionCreator` ist, wie oben beschrieben, hauptsächlich für die Handhabung von Java NIO zuständig. Dabei wartet der Thread auf den Selector bis mindestens ein Channel bereit ist Daten zu empfangen oder zu senden. Sobald der Thread seine Aufgabenliste durch den Selector erhalten hat, werden die Channels sequenziell abgearbeitet. Grundsätzlich wäre es jedoch möglich mehrere Channels parallel zu bearbeiten. Insbesondere wird es von Java NIO unterstützt bei Bedarf gleichzeitig aus einem Channel zu lesen oder zu schreiben[4].

#### Worker im NetworkHandler

Um die eigentlichen Nachrichten zu deserialisieren kommt der Worker im `NetworkHandler` zum Einsatz. Sobald eine Verbindung neue Daten gelesen hat, liest der Worker die Daten aus und übergibt sie dem zuständigen `IncomingMessageCreator`,

[4] `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/SocketChannel.html`

wo sie anschließend deserialisiert werden. Der Worker im `NetworkHandler` ist jedoch als einziger Empfänger für neu gelesene Daten im `NIOConnectionCreator` registriert. Somit kann nur er alleine alle eingehende Nachrichten sequenziell abarbeiten. Wie in Unterabschnitt 3.2.2 beschrieben ist der Deserialisierungsvorgang sehr aufwendig. Bei mehreren Verbindungen, könnten grundsätzlich mehrere Threads simultan für diese Aufgaben zum Einsatz kommen. Dies begründet sich durch die Unabhängigkeit der verschiedenen Datenströme. Insbesondere die Deserialisierung und Auslieferung der Nachrichten sind unabhängig von anderen Aufgaben.
Weiterhin liefert der Worker im `NetworkHandler` vollständig empfangene und deserialisierte Nachrichten an die Anwendung aus. Sind aufgrund der eingegangenen Nachricht zeitaufwendige Operationen notwendig, kann der Worker währenddessen keine weiteren Nachrichten verarbeiten. Es bietet sich damit an, diesen Teil ebenfalls in einem unabhängigen Thread auszuführen. Somit ließe sich die Netzwerkkomponente besser vom Rest der Anwendung separieren.

### 3.2.5 Log4J

Zur Überwachung der Funktionalitäten in DXRAM kommt *Apache Log4J*[5] als Logger zum Einsatz. Dieser ist überaus flexibel und ermöglicht das Mitschreiben der Ausgaben in unterschiedliche Dateien. Außerdem lassen sich die Ausgaben in verschiedenen Stufen filtern, geordnet nach Dringlichkeit. Leider sind die Log4J Aufrufe vergleichsweise teuer. Dies betrifft auch Ausgaben, welche anschließend ausgefiltert werden [TG03].

Aktuell befinden sich diverse Log4J Aufrufe auch in Programmabschnitten, welche hochfrequent durchlaufen werden, wie etwa beim Senden, Empfangen und Deserialisieren. Entsprechend wirkt sich Log4J auf die allgemeine Laufzeit aus. Logger werden natürlich nicht grundlos eingesetzt, jedoch empfiehlt es sich nicht zwingend benötigte Aufrufe für die produktive Version zu entfernen. Das gilt insbesondere für Aufrufe in leistungskritischen Programmabschnitten.

[5] `http://logging.apache.org/log4j/`

# Kapitel 4

# Implementierung

Der Hauptteil dieser Arbeit beschäftigt sich mit der Verbesserung der Netzwerkkomponente für DXRAM. Insbesondere werden dabei die in Abschnitt 3.2 aufgezeigten Schwächen beseitigt, sodass der Netzwerkcode den Ansprüchen eines verteilten In-Memory Storage gerecht wird. Dies hat einige Änderungen in der Implementierung zur Folge, wozu unter anderem auch strukturelle Anpassungen gehören. Im folgenden Kapitel werden diese Änderungen vorgestellt und erläutert.

## 4.1 MessageDirectory

Neu in der Netzwerkkomponente ist das `MessageDirectory`. Das `MessageDirectory` ist eine Klasse, in der verwendete Nachrichtenklassen mit vorgegebener Typ und Subtyp ID registriert werden. Die Abbildung von Typ und Subtyp ID kann anschließend genutzt werden um ein neues Nachrichtenobjekt des entsprechenden Typs zu instanziieren.

In Listing 4.1 ist das Interface der Klasse `MessageDirectory` zu sehen. Zum Registrieren der Nachrichtenklassen wird die Methode `register()` verwendet. Unter Angabe von Typ ID, Subtyp ID und der gewünschten Klasse wird diese registriert. Dieser Schritt muss zwingend für jeden Nachrichtentyp zum Programmstart ausgeführt werden.

Nachrichtentypen, welche zum Zeitpunkt des Empfangens nicht im `MessageDirectory` bekannt sind, können nicht deserialisiert werden und müssen verworfen werden. Weiterhin muss der Programmierer darauf achten, dass ein Typ-Subtyp ID Tupel höchstens einmal verwendet wird. Sollte ein ID Tupel ein weiteres mal registriert werden wird eine `IllegalArgumentException`[1] geworfen und der Vorgang ignoriert. Um eine Exception zu vermeiden steht dem Programmierer die Methode `contains()` zur Verfügung. Mit Hilfe dieser Methode lässt sich abfragen, ob unter einem Typ-Subtyp ID Tupel bereits eine Nachrichtenklasse abgelegt wurde.

```
public final class MessageDirectory {

    public static synchronized void register(final byte p_type, final
       byte p_subtype, final Class<?> p_class);

    public static boolean contains(final byte p_type, final byte
       p_subtype);

    public static AbstractMessage getInstance(final byte p_type, final
        byte p_subtype) throws NetworkException;
}
```

Listing 4.1: Interface der Klasse *MessageDirectory*

Die letzte verbleibende Methode ist `getInstance()`. Mit deren Hilfe lässt sich ein neues Nachrichtenobjekt erzeugen, welches dem angegebenen Typ-Subtyp ID Tupel entspricht. Diese Methode wird beim Empfangen der Nachrichten verwendet. Die `NetworkException` wird ausgelöst, wenn der angeforderte Nachrichtentyp nicht registriert wurde, oder wenn beim Erzeugen des Objekts Probleme aufgetreten sind.

Intern arbeitet das `MessageDirectory` auf Basis eines zweidimensionalen Arrays, welches auf Klassenkonstruktoren verweist. Als Indizes für das Array dient das mitgelieferte Typ-Subtyp ID Tupel. Das hat zwangsläufig direkte Folgen auf das Verhalten der Klasse. Liegt der neue Index außerhalb der bisherigen Arrays müssen diese entsprechend vergrößert werden. Folglich ist es notwendig alle bisherigen Einträge in ein größeres Array zu kopieren. Um keinen unnötigen Speicherplatz zu belegen wird die Größe der neuen Arrays genau so groß gewählt, dass das aktuell einzufügende Objekt abgelegt werden

[1] `http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html`

kann. Somit kommt es zu vielen Kopiervorgängen, wenn Nachrichtentypen aufsteigend nach ihrer ID eingefügt werden. Um den Kopieraufwand zu vermeiden empfiehlt es sich, die Nachrichten mit den größten Typ bzw. Subtyp ID Werten zuerst einzufügen. Weiterhin sollte im Rahmen der Speichereffizienz darauf geachtet werden, dass möglichst keine ungenutzten ID Werte innerhalb der Wertebereiche existierten, da diese dennoch grundsätzlich Teil des zweidimensionalen Arrays sind.

Trotz des initialen Aufwands bietet das `MessageDirectory` einen großen Vorteil. Durch die Abbildung von Typ-Subtyp auf den Konstruktor des Nachrichtenobjekts lassen sich die einzelnen Nachrichtenobjekte sehr viel schneller erzeugen, als bei der Verwendung der Java Reflection API. Auch die in Unterabschnitt 3.2.3 beschriebenen Nachrichtenheader können somit deutlich verkürzt werden, dadurch dass die vollständigen Java Klassennamen nicht mehr übermittelt werden müssen und zur Identifikation die Typ und Subtyp ID genügen.

Das die Vorteile dieser Klasse die Nachteile überwiegen zeigt sich, wenn man betrachtet, wann die Stärken und Schwächen zum Tragen kommen. Die Hauptschwäche liegt beim registrieren neuer Nachrichtentypen. Für den Programmierer bedeutet das zusätzlichen Aufwand, da alle verwendeten Nachrichtentypen einzeln registriert werden müssen. Je nach Reihenfolge kann es zusätzlich dazu kommen, dass die unterliegenden Arrays häufig umkopiert werden müssen. Der Vorteil der Klasse `MessageDirectory` liegt jedoch in der sehr schnellen Abbildung von Typ-Subtyp ID auf den gesuchten Konstruktor, was sich überaus positiv auf den eigentlichen Betrieb auswirkt. Geht man davon aus, dass während des laufenden Betriebs sehr selten oder gar keine neuen Nachrichtentypen mehr registriert werden müssen, bestehen die Nachteile nur zu Beginn der Anwendung. Betrachtet man weiterhin, dass das Programm im Idealfall nur selten gestartet wird und dafür lange in Betrieb ist, profitiert das Programm deutlich von den Änderungen.

Weiterhin ist, aufgrund von Schreibzugriffen, nur für die `register()` Methode ein synchronisierter Zugriff erforderlich. Das Löschen von Einträgen ist nicht vorgesehen, weswegen die anderen beiden Methoden uneingeschränkt parallel aufgerufen werden können. Sollte `register()` simultan mit einer anderen Methode aufgerufen werden schränkt dies die Verwendung ebenfalls nicht ein. Das Einzige, was passieren kann, ist, dass die lesenden Aufrufe auf veralteten Daten arbeiten und ein Nachrichtentyp noch

nicht gefunden werden kann. Werden alle Nachrichtentypen vor Beginn der Netzwerkkommunikation registriert ist dieser Fall jedoch ausgeschlossen.

### 4.1.1 Nachrichtenheader

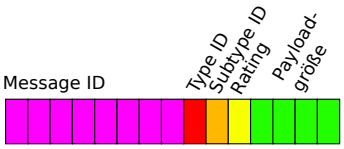



Abbildung 4.1: Neuer Nachrichtenheader

Wie oben beschrieben kann durch die Einführung der `MessageDirectory` Klasse der Java Klassenname aus dem Nachrichtenheader entfernt werden. Jedoch können einige andere Felder ebenfalls aus dem Header entfernt werden. Der finale Header ist in Abbildung 4.1 dargestellt. Im Vergleich zum alten Header in Abbildung 3.2 werden nun nur noch konstant 15 Byte für den Nachrichtenheader benötigt, unabhängig von Nachrichtentyp und Nutzdatengröße. Wie in Unterabschnitt 3.2.3 beschrieben, bestand der Nachrichtenheader bisher aus mindestens 23 Byte, zzgl. dem vollständigen Java Klassennamen.
Nimmt man wieder das Rechenbeispiel aus Unterabschnitt 3.2.3 zur Hand, ergibt sich für den neuen Header bei 16 Byte Nutzdaten ein Mehraufwand von ca. 94 % (vorher ca. 444 %), und ca. 23 % für 64 Byte Nutzdaten (vorher ca. 111 %). Im Kontext von DXRAM ist dies somit eine deutliche Leistungssteigerung, da der Anteil der Nutzdaten im Vergleich zu den Metadaten deutlich verringert werden konnte.

Der Großteil der Einsparung geht dabei auf den Wegfall der Java Klassennamen zurück. Dieser allein hat bereits sehr viele Bytes in Anspruch genommen. Hinzu kam, dass durch die dynamische Länge ein Feld voran gehen musste, welches die Bytegröße des gesamten Nachrichtenheaders enthielt. Da alle verbleibenden Einträge eine feste Byteanzahl benötigen, konnte das Größenfeld weggelassen, und so erneut ein Integer (vier Byte) eingespart werden.
Die Felder für Quell und Ziel Node ID waren ebenfalls obsolet. Die Node ID Einträge werden lediglich angewendet, um im Programm die Kommunikationspartner der vorliegenden Nachricht zu identifizieren. Da DXRAM kein eigenes Routing betreibt sind

die hier übermittelten Werte jederzeit für Sender und Empfänger eindeutig, da die Verbindung, über welche die Nachrichten übermittelt werden, bekannt ist. Somit können diese Felder automatisch beim Deserialisieren befüllt werden und müssen nicht über die Leitung gesendet werden. Damit konnten wiederum zwei Short Werte (je zwei Byte) eingespart werden.

Alle weiteren Felder sind weiterhin in Verwendung und notwendig. Theoretisch ließe sich der Nachrichtenheader jedoch auch noch weiter verkürzen. Dabei wäre zu überlegen, wie lange die Werte der Nachrichten ID eindeutig sein müssen, und ob 64 Bit für die Nachrichten ID tatsächlich notwendig sind. So ließen sich leicht weitere vier Byte vom Nachrichtenheader einsparen. Zu beachten ist, dass die Nachrichten ID bereits jetzt nur eindeutig in Kombination mit der Node ID des Senders ist. Umgekehrt bedeutet das, dass jeder Knoten seinen eigenen ID Raum besitzt. Bei einem System, welches nur sehr selten neugestartet werden soll, bedeutet dies zwangsläufig, dass es zum Überlauf des ID Zählers und somit zur Neuvergabe von IDs kommt.
Ähnliche Überlegungen kann man für das Feld der Nutzdatenlänge anstellen. Die hier eingeräumten vier Byte gehen weit über die üblichen Nutzdatengröße von 16 bis 64 Byte hinaus. Ist man sicher, dass nicht deutlich mehr Daten in einer Nachricht übermittelt werden, ließen sich auch hier weitere zwei Byte, durch die Verwendung eines Short anstatt eines Integer, einsparen.

Die beiden aufgezeigten Einsparungsmöglichkeiten sind jedoch aktuell nur rein theoretische Überlegungen. Aus Gründen der Kompatibilität wurden beide Felder auf die gewählten Längen festgesetzt. Stellt sich jedoch im weiteren Verlauf der Entwicklung von DXRAM heraus, dass die Felder gekürzt werden können, ist dies technisch schnell und einfach in der Netzwerkkomponente umsetzbar und könnte zu einer weiteren Verbesserung der Netzwerkleistung führen.

## 4.2 Reactor mit Thread Pool

Im folgenden Abschnitt werden einige strukturelle Änderungen erläutert welche die in Unterabschnitt 3.2.4 aufgezeigten Probleme beheben oder umgehen sollen. Ziel ist es,

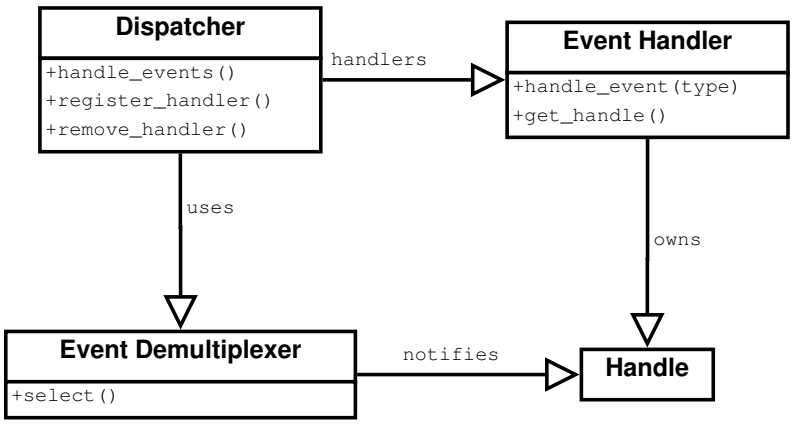



Abbildung 4.2: UML Klassendiagramm des *Reactor Pattern*

die allgemeine Verwendung von Threads innerhalb der Netzwerkkomponente zu optimieren. Die Lösung besteht aus der Kombination zweier Konzepte. Zum einen dem *Reactor Pattern* und zum anderen dem Einsatz eines *Thread Pools*. Beide Konzepte sollen nun vorgestellt und entsprechende Anpassungen in der Netzwerkkomponente aufgezeigt werden.

### 4.2.1 Reactor Pattern

Einfach gesagt beschreibt das *Reactor Pattern*, dass asynchrone Ereignisse zentral gesammelt werden. Anschließend werden sie an ihre zuständigen Empfänger weitergeleitet. Ereignisse werden in der Regel als unabhängig voneinander betrachtet. Eine direkte Konsequenz davon ist, dass die Bearbeitungen auf verschiedene Threads aufgeteilt werden können. Somit können alle Ereignisse parallel abgehandelt werden. In Abbildung 4.2 ist das UML Klassendiagramm des Reactor Pattern zu sehen. Nach [Sch95a] gibt es dabei im Kern des Reactor Pattern folgende Hauptakteure:

- Ein oder mehrere *Handles*
- Einem *Event Demultiplexer*
- Einem *Dispatcher*

- Ein oder mehrere *Event Handler*

Die Handles umschreiben dabei in der Regel vom Betriebssystem verwaltete Ressourcen, wie etwa geöffnete Dateien oder Netzwerkverbindungen. Sie können dabei unvorherbestimmbare Events auslösen, auf welche reagiert werden kann oder muss. Diese Events werden über den Event-Demultiplexer synchronisiert und gesammelt und können kollektiv abgerufen werden. Dieser arbeitet dabei nicht blockierend. In Java NIO entspricht der Selector dem Event-Demultiplexer.
Die gesammelten Ereignisse werden an den Dispatcher weitergereicht. Dieser kann entscheiden welcher Event Handler für die anschließende Verarbeitung zuständig ist. Im Event Handler kann dann schlussendlich auf das Event reagiert werden.

Diese Aufteilung ermöglicht eine vereinfachte Abarbeitung von Ereignissen, welche aus unterschiedlichen Quellen stammen können. Weiterhin ist das ganze System sehr dynamisch, was neue Handles oder Event Handler betrifft. Diese können nach belieben an- oder abgemeldet werden, ohne den Programmfluss zu stören. Ebenfalls lässt sich das Programm auf diese Weise von der tatsächlich vorhandenen Anzahl von Threads abkoppeln, womit sich die Anwendung besser an das System anpassen lässt, auf dem sie ausgeführt wird [Sch95b].
Die Eigenschaften dieser Mechanik lassen sich für den hier vorliegenden Anwendungsfall sehr gut ausnutzen. Wie im folgenden Abschnitt zu sehen sein wird, lässt sich das Reactor Pattern dabei sogar mehrfach anwenden.

### 4.2.2 Task Executor

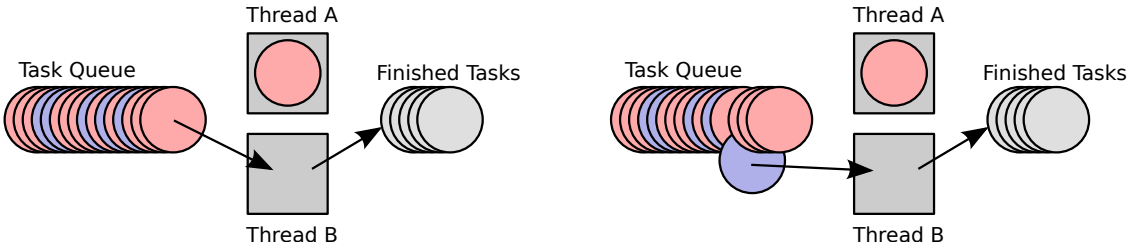
Um aus dem *Reactor Pattern* effizienten Nutzen für Multithreading zu ziehen, bedarf es eines Mechanismus, welcher anfallende Aufgaben effizient auf verschiedene Threads aufteilen kann. Das Erzeugen eines Threads ist relativ teuer, weswegen nicht für jede Aufgabe ein eigener Thread erstellt werden sollte. Außerdem erfordern viele Threads auch einen häufigen Kontextwechsel durch das Betriebssystem. Dadurch würde das System insgesamt verlangsamt werden. Deutlich einfacher und effizienter ist dagegen die Nutzung eines Thread Pools [Kri04].

Ein Thread Pool ist eine Klasse, in der eine Reihe von Threads vorgehalten werden um anfallende Aufgaben zu erledigen. Die Aufgaben werden meistens in einer `Queue` organisiert, welche der Reihe nach von freien Threads abgearbeitet werden. Als Gemeinsamkeit benötigten die Aufgaben nur einen festen Einstiegspunkt für den Thread. In Java wäre dies z.B. eine Implementierung des `Runnable` Interfaces[2]. In Java gibt es bereits einige Implementierungen für verschiedene Thread Pools, welche unter dem `ExecutorService` Interface[3] vereint sind. Zum Erzeugen der verschiedenen Implementierungen wird mit der Klasse `Executors`[4] eine Factory[5] zur Verfügung gestellt. Diese bietet Funktionen zum Erzeugen von Thread Pools mit fester und dynamischer Anzahl an Threads. Bei einem dynamischen Thread Pool hat man weiterhin die Auswahl zwischen verschiedenen Erzeugungsstrategien.

Somit liegt die grundlegende Logik für einen Thread Pool bereits in Java vor. Allerdings gibt es durch die `ExecutorService` Implementierung keine Möglichkeit Abhängigkeiten zwischen den verschiedenen Aufgaben darzustellen. Abhängigkeiten sind für einen effizienten Multithreading Betrieb zwar unerwünscht, lassen sich aber bei der Anwendung von Java NIO nicht verhindern. Ein kleines Fallbeispiel:

Der Selector empfängt viele kleinere Datenfragmente einer umfangreichen Netzwerkübertragung. Jedes Datenfragmet muss nun verarbeitet und interpretiert werden. Dazu werden die entsprechenden Aufgaben in einen Thread Pool mit zwei Threads eingehangen. Das Problem ist, dass die Daten als Datenstrom empfangen werden. Somit können die Datenfragmente erst interpretiert werden, nachdem alle vorausgehenden Fragmente bearbeitet wurden. Die Threads müssten somit abwechselnd aufeinander warten und blockieren, womit sie nicht für andere Aufgaben zur Verfügung stehen.
Erhält man nun Daten über eine weitere Verbindung, könnten die hier anfallenden Datenfragmente nicht sofort bearbeitet werden. Dadurch, dass die Daten der ersten Verbindung hintereinander weg in die Aufgabenwarteschlange eingehangen wurden, müssen diese zuerst bearbeitet werden. Grundsätzlich würde zwar für jede Verbindung ein eige-

[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html`
[3] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html`
[4] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html`
[5] *Factory Pattern*, vgl. [ES13]




(a) Normale Warteschlange (b) Optimierte Warteschlange

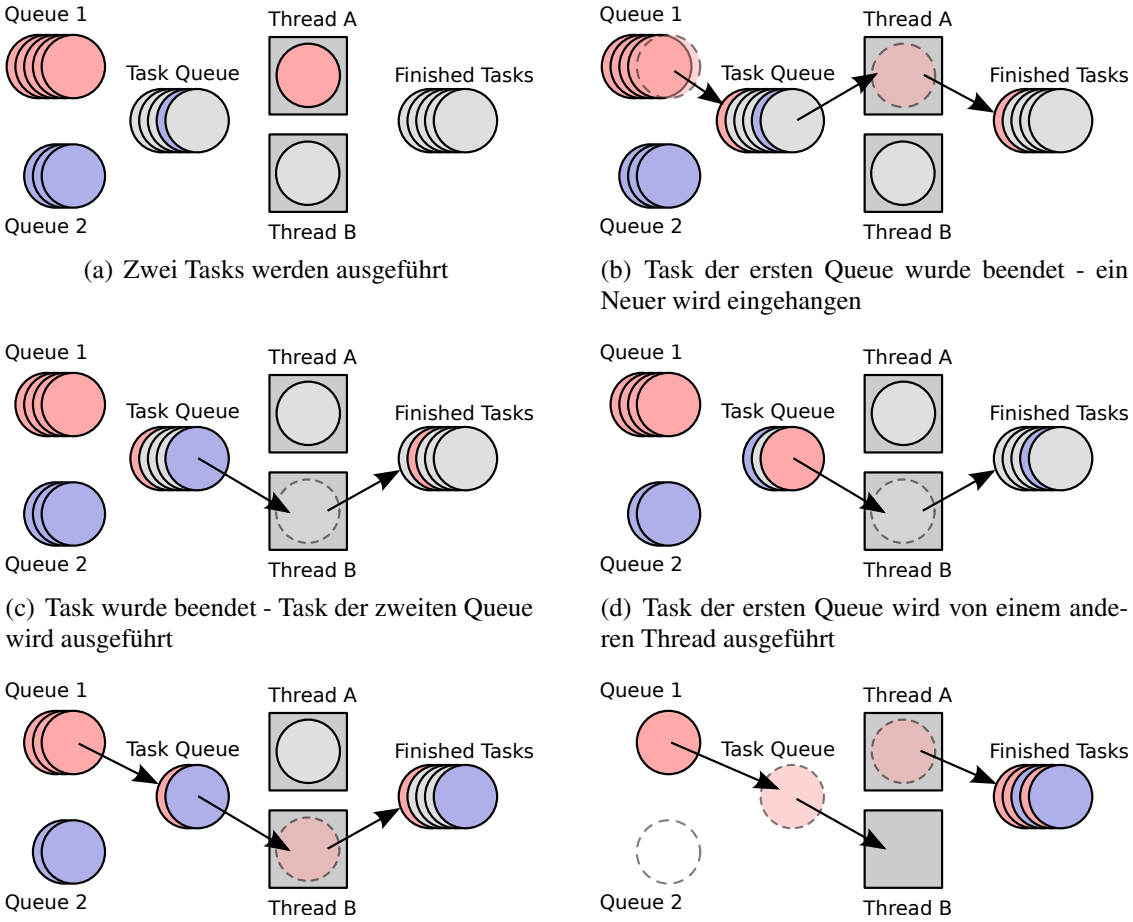
Abbildung 4.3: Warteschlangen für einen Thread Pool

ner Thread bereit stehen, jedoch können diese nicht effektiv genutzt werden. Durch die Eigenschaften von Java NIO und dem Thread Pool können die Threads nicht unabhängig voneinander arbeiten. Die Nachrichten würden schlussendlich doch die meiste Zeit sequenziell bearbeitet werden.

Abbildung 4.3(a) zeigt eine solche Situation. Zwei Threads mit einer gemeinsamen Warteschlange haben eine Reihe von Aufgaben, welche abhängig von ihren farblichen Vorgängern sind. Thread B könnte in dem Fall erst mit der Bearbeitung beginnen, wenn Thread A mit seinen Berechnungen fertig ist. Im Anschluss müsste Thread A auf die Ergebnisse von Thread B warten. Besser wäre es, wenn Thread B direkt eine Aufgabe aus der zweiten Gruppe zugewiesen bekäme, wie es in Abbildung 4.3(b) skizziert ist.

Das genannte Beispiel lässt sich natürlich mit wachsender Threadanzahl beliebig erweitern. Um dennoch einen Vorteil aus dem Thread Pool ziehen zu können benötigt man in diesem Fall Abhängigkeiten. Diese können dem Thread Pool bei der Entscheidung helfen, welche Aufgabe als nächstes ausgeführt werden kann, ohne das diese aufgrund anderer Aufgaben blockiert. Aus diesem Grund wurde mit dem `TaskExecutor` eine Klasse implementiert, welche den `ExecutorService` unter Berücksichtigung von Abhängigkeiten bedienen kann.

Für die in Abbildung 4.3(b) dargestellte Verteilung der Aufgaben müsste man überblicken können, an welcher Stelle der Liste sich das nächste geeignete Element befindet. Um ein solches Element in der Warteschlange finden zu können muss jederzeit bekannt sein welche Aufgabe jeder Thread aktuell ausführt. Anschließend müsste ermittelt werden, ob sich ein Element in der Warteschlange befindet, welche keine Abhängigkeiten




(a) Zwei Tasks werden ausgeführt

(b) Task der ersten Queue wurde beendet - ein Neuer wird eingehangen

(c) Task wurde beendet - Task der zweiten Queue wird ausgeführt

(d) Task der ersten Queue wird von einem anderen Thread ausgeführt

(e) Tasks der beiden Queues wechseln sich in einem Thread ab

(f) Nur noch eine Queue hat Tasks - Tasks werden abwechselnd von beiden Threads ausgeführt

Abbildung 4.4: Thread Pool mit vorgeschalteten Warteschlangen

zu den derzeit ausgeführten Aufgaben besitzt. Das führt dazu, dass dem Thread Pool ein großer Verwaltungsaufwand aufgelastet wird, welcher sich negativ auf die Ausführungsgeschwindigkeit auswirkt. Um den Zusatzaufwand gering zu halten, besitzt der `TaskExecutor` eigene Warteschlangen für jede Aufgabengruppe.

Die Arbeitsweise der vorgeschalteten Warteschlangen wird in Abbildung 4.4 dargestellt. Wie in Abbildung 4.4(a) zu sehen ist, wird für jede Gruppe eine eigene Warteschlange vom `TaskExecutor` angelegt. Die `TaskQueue` in der Abbildung stellt die allgemeine Warteschlange des `ExecutorService` dar. Dabei ist stets nur eine Aufgabe jeder Gruppe gleichzeitig in der allgemeinen Warteschlange oder gerade in Bearbeitung durch

einen Thread. Zu beachten ist, dass nicht zwangsläufig jede Aufgabe Teil einer Aufgabengruppierung sein muss. Hat eine Aufgabe keine bindenden Abhängigkeiten zu anderen Aufgaben kann diese direkt in die Warteschlange der `ExecutorService` Klasse eingehangen und ausgeführt werden, ohne das der `TaskExecutor` diese Aufgabe selber verwaltet.

Abbildung 4.4(b) zeigt, dass die nächste Aufgabe der Gruppe erst eingehangen wird wenn die aktuelle abgeschlossen ist. Zwischen diesen beiden Punkten können Aufgaben anderer Gruppe eingehangen und bearbeitet werden. Um zu verhindern, dass zwei aufeinander folgende Aufgaben sich direkt hintereinander in der allgemeinen Warteschlange befinden, wird beim Beginn der Bearbeitung noch keine neue Aufgabe eingereiht (siehe Abbildung 4.4(c)). Wie in Abbildung 4.4(d) gezeigt wird die eigentliche Ausführung durch den Thread Pool nicht beeinflusst. Dieser kann nicht zwischen den verschiedenen Gruppierungen der Aufgaben unterschieden, somit werden Aufgaben der selben Gruppe auch nicht zwingend vom selben Thread bearbeitet.

Sind keine weiteren unabhängigen Aufgaben mehr verfügbar entspricht die Länge der allgemeinen Warteschlange genau der Anzahl aktiver Gruppen (siehe Abbildung 4.4(e)). Alle Gruppen werden gleichmäßig durch die verfügbaren Threads abgebaut. Sind beispielsweise zwei Threads und nur noch in zwei Gruppen wartende Aufgaben aktiv, arbeitet jeder Thread eine Gruppe ab. Enthält, wie in Abbildung 4.4(f)), nur noch eine Gruppe Aufgaben, wird diese sequenziell abgearbeitet. Die Threads wechseln sich jedoch bei der Bearbeitung ab, sofern keine weiteren Aufgaben dazwischen kommen. Das liegt daran, dass die Folgeaufgabe beim Abschluss der aktuellen Aufgabe eingehangen wird. Somit ist formell Thread A noch mit der ersten Aufgabe beschäftigt, sodass Thread B als freier Thread die Aufgabe übernimmt.

Die Implementierung der Klasse `TaskExecutor` ist in Listing 4.2 angedeutet. Bei der Instanziierung erzeugt der `TaskExecutor` mittels der `Executors` Factory einen Thread Pool mit einer konstanten Anzahl an Threads. Ein dynamischer Thread Pool mit wechselnder Anzahl an Threads kann, je nach Anwendung, verschiedene Vor- und Nachteile mit sich bringen[6]. Um einen verlässlichen Mittelweg zu finden wird deswegen auf

[6] Dies ist schlussendlich vom Zugriffsmuster der auf DXRAM aufbauenden Anwendung abhängig. Geht man von einem annähernd gleichverteilten Zugriff auf die Daten und Knoten aus, liegt eine konstante Last auf dem Netzwerkmodul.

```
public final class TaskExecutor {
    private final ExecutorService m_executor;
    private final HashMap<Short, TaskQueue> m_taskMap;

    public TaskExecutor(final String p_name, final int p_threads) {
        m_executor = Executors.newFixedThreadPool(p_threads, new
            ExecutorThreadFactory());
    }

    public void execute(final Runnable p_runnable) {
        try { m_executor.execute(p_runnable); }
        catch (final RejectedExecutionException e) { /* ... */  }
    }

    public synchronized void execute(final short p_id, final Runnable
        p_runnable) {
        TaskQueue taskQueue;
        synchronized (m_taskMap) {
            if (!m_taskMap.containsKey(p_id)) {
                taskQueue = new TaskQueue();
                m_taskMap.put(p_id, taskQueue);
            } else { taskQueue = m_taskMap.get(p_id); }
            taskQueue.add(p_runnable);
        }
    }

    public void shutdown();
    public static TaskExecutor getDefaultExecutor();

    private class TaskQueue implements Runnable {
        private final ArrayDeque<Runnable> m_queue;
        public void add(final Runnable p_runnable);
    }
}
```

Listing 4.2: Auszug der Klasse *TaskExecutor*

eine sich selbst anpassende Thread Anzahl verzichtet. Sollte erkennbar sein, dass eine flexible Anpassung Vorteile mit sich bringt, ließe sich das an dieser Stelle sehr schnell anpassen.

Zum Ausführen von Aufgaben dient die Methode `execute()`. Dieser Methode wird ein Objekt übergeben, welches das Runnable Interface implementiert. Diese Aufgabe wird als unabhängig betrachtet und direkt an den darunter liegenden Thread Pool weitergeleitet. Wird der `execute()` Methode zusätzlich ein Short Wert mit übergeben,

```
public void run() {
    Runnable runnable;
    synchronized (m_queue) { runnable = m_queue.peek(); }
    try { runnable.run(); }
    catch (final Exception e) { /* ... */ }
    finally {
        synchronized (m_queue) {
            m_queue.remove();
            if (m_queue.size() > 0) { execute(this); }
        }
    }
}
```

Listing 4.3: *run()* Methode der Klasse *TaskQueue*

wird dieser Wert als Gruppen ID verwendet. Über diese ID werden zusammengehörige Aufgaben identifiziert. Aufgaben mit Gruppen ID werden nicht direkt an den `ExecutorService` weitergeleitet. Damit die Warteschlangen im `TaskExecutor` erkennen können, wann eine Aufgabe abgeschlossen ist und die nächste eingehangen werden kann, werden die Warteschlangen selber an den Thread Pool übergeben. Hierfür dient die Klasse `TaskQueue` als *Wrapper*[7] für die eigentliche Liste, in der die Aufgaben angehangen werden.

Die `TaskQueue` implementiert selber das Runnable Interface und ist somit in der Lage ausgeführt zu werden. Die `run()` Methode ist in Listing 4.3 zu sehen. Sobald diese aufgerufen wird, wird der Aufruf an die erste Aufgabe der internen Liste weitergeleitet. Wurde die Aufgabe abgearbeitet wird diese aus der internen Liste entfernt. Anschließend wird geprüft, ob die interne Liste weitere Elemente enthält. Ist das der Fall, reiht sich die `TaskQueue` selbstständig wieder in die Warteschlange des `ExecutorService` ein. Dieser Vorgang wiederholt sich, bis keine Elemente mehr enthalten sind.
Die `TaskQueue` wird nur explizit der allgemeinen Warteschlange hinzugefügt, wenn ein neues Element in eine leere TaskQueue eingefügt wird. Sind bereits Einträge vorhanden, wird weiter auf den Aufruf durch den Thread Pool gewartet. Auf diese Weise ist es möglich voneinander abhängige Aufgaben kontrolliert mit einem Thread Pool abzuarbeiten, ohne das Threads auf die Ergebnisse anderer Aufgaben warten müssen.

[7] Vgl. [ES13]

Die verbleibenden Aufrufe `shutdown()` und `getDefaultExecutor()` in Listing 4.2 dienen der Steuerung des Thread Pools. Mittels `getDefaultExecutor()` erhält man eine Referenz auf eine zentrale Instanz der Klasse `TaskExecutor`[8]. Diese dient als Standard Thread Pool, wenn kein eigener Thread Pool benötigt wird. Mittels `shutdown()` lässt sich der Thread Pool kontrolliert beenden. Zu beachten ist, dass ab dem Zeitpunkt des Aufrufs nur die verbleibende Warteschlange des `ExecutorService` abgearbeitet werden kann. Alle verbleibenden Aufgaben in den vorgeschalteten Warteschlangen werden verworfen.
Wird eine `TaskQueue` nicht mehr benötigt wird diese über die Methode `purgeQueue()`, unter Angabe der Gruppen ID, entfernt. Diese Methode wird durch den `ConnectionManager` aufgerufen sobald eine Verbindung geschlossen wurde. Alternativ könnte die `TaskQueue` auch gelöscht werden, sobald diese leer ist. Das kann allerdings dazu führen, dass sie sehr häufig neu erzeugt werden muss. Um die Garbage Collection zu entlasten wird also davon ausgegangen, dass die `TaskQueue` so lange gebraucht wird, wie auch die entsprechende Verbindung offen ist.

### 4.2.3 Anpassung der Verbindungslogik

Das Reactor Pattern bietet sich für Java NIO überaus an, weil es im Grunde bereits innerhalb von Java NIO umgesetzt wurde [WQ12]. Die verwendeten Channels entsprechen den Handles im Pattern, und der Selector übernimmt die Rolle des Event-Demultiplexers. Lediglich die zweite Hälfte des Patterns muss vom Entwickler selbst umgesetzt werden. Das war bisher in der DXRAM Komponente nicht der Fall.

Wie in Unterabschnitt 3.1.3 beschrieben sammelt der Worker Thread im `NIOConnectionCreator` die Keys vom Selector ab. Diese werden jedoch nicht, wie bei einem Dispatcher, an dafür vorgesehene Event Handler übermittelt, sondern direkt beim Aufkommen bearbeitet. Aus diesem Grund wurde die Klasse `SelectionKeyHandler` eingeführt. Wie der Name vermuten lässt, übernimmt diese Klasse die Rolle des Event Handlers für die `SelectionKey` Objekte, welche Java NIO zurück gibt. Dadurch, dass die `SelectionKey` Referenzen durch Java NIO wiederverwendet werden, kön-

[8] *Singleton Pattern*, vgl. [ES13]

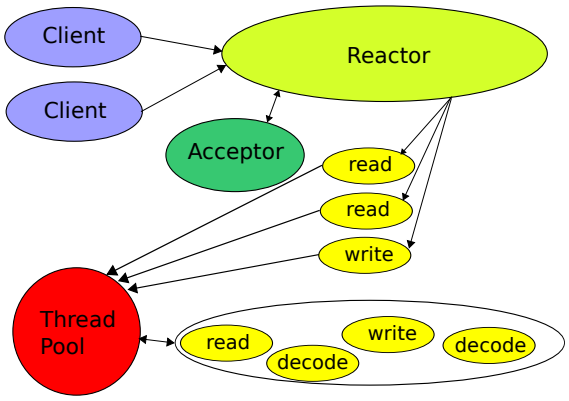



Abbildung 4.5: Reactor Pattern mit Java NIO

nen auch die `SelectionKeyHandler` für eine erneute Nutzung abgelegt werden. So wird die Garbage Collection nicht unnötig belastet.

Der `SelectionKeyHandler` beinhaltet dabei sämtliche Logik, welche zum Lesen, Schreiben und Aufbauen einer Verbindung notwendig ist. Anstatt die Operationen selbst durchzuführen kann der Worker Thread die Handler nun an den TaskExecutor weiterreichen. Dies ermöglicht ein simultanes lesen und schreiben mehrerer Verbindungen. Eine Ausnahme bilden dabei neue Verbindungen, weil diese beim Selector registriert werden müssen. Wie in Unterabschnitt 2.2.2 beschrieben besteht die Gefahr eines Deadlocks. Aus diesem Grund werden neue Verbindungen zuerst vom selben Thread bearbeitet, welcher den Selector abgerufen hat.

Der Fall, dass ein Channel gleichzeitig bereit zum Lesen und Schreiben ist muss ebenfalls abgefangen werden. Der zugehörige `SelectionKeyHandler` wird zwangsläufig doppelt dem Thread Pool hinzugefügt. Im `SelectionKeyHandler` wird anschließend durch Abfragen der Markierungen geprüft, welche Aktion auszuführen ist. Weil die Markierungen erst zurückgesetzt werden nachdem die Aktion abgeschlossen ist muss sichergestellt werden, dass nicht beide Threads parallel die gleiche Aktion behandeln. Hierfür wird ein zusätzliches Feld verwendet, mit dem überprüft werden kann, ob sich bereits ein Thread der Aufgabe gewidmet hat und entsprechend mit einer anderen Aufgabe fortgefahren werden kann.

Um den Worker Thread im `NIOConnectionCreator` effektiver nutzen zu können

wird dieser nicht mehr länger von einem dedizierten Thread ausgeführt. Der Worker wird nun ebenfalls als Aufgabe über den Thread Pool ausgeführt. Anstatt also den Worker während der Erzeugung der `NIOConnectionCreator` Instanz in einem eigenen Thread zu starten, wird dieser dem Thread Pool übergeben. Um die Aufgabe abzuschließen und den Thread wieder freigeben zu können, wird im Worker nun darauf verzichtet in einer Schleife bis zur Programmterminierung den Selector abzufragen. Stattdessen werden einmal alle fälligen Keys über ihre Handler dem Thread Pool hinzugefügt. Anschließend fügt sich der Worker ebenfalls selbst in den Thread Pool ein und läuft aus. Somit kann der Worker Thread effektiv genutzt werden um anfallende Aufgaben zu erledigen.

Durch die Einführung der `SelectionKeyHandler`, welche über einen Thread Pool ausgeführt werden, entspricht die Arbeitsweise im `NIOConnectionCreator` nun dem Reactor Pattern und ähnelt dem in [Lea03] vorgeschlagenen Aufbau für eine Netzwerkkommunikation über Java NIO. In Abbildung 4.5 wird das aktuelle Schema skizziert. Die Client-Verbindungen laufen über einen Reactor zusammen. Alle anfallende Aufgaben können nun von mehreren Threads simultan abgearbeitet werden.

### 4.2.4 Verbesserung der Nachrichtenverarbeitung

Nachdem der Worker im `NIOConnectionCreator` effektiv mehrere Threads ausnutzen kann muss die Deserialisierung der eingehenden Nachrichten und deren Verarbeitung ebenfalls optimiert werden. Wie in Unterabschnitt 3.2.4 beschrieben ist in der bisherigen Version ein einzelner Thread für die komplette Nachrichtenverarbeitung zuständig. Um die Nachrichten effizient deserialisieren und verarbeiten zu können, wird die Arbeitsweise des Workers im `NetworkHandler` ähnlich dem Reactor Pattern angepasst.

Wie auch im `NIOConnectionCreator` betrachten wir im `NetworkHandler` eine Reihe von Verbindungen, welche ungefähr den Handles entsprechen. Jedoch werden diese Verbindungen nicht mehr direkt vom Betriebssystem, sondern vom eigenen Programm verwaltet. Ein weiterer Unterschied ist, dass es keinen synchronisierten Event-Demultiplexer gibt. Die hier vorliegenden Verbindungen sind in der Lage dem `NetworkHandler` über einen Callback mitzuteilen wenn neue Daten für die Verarbeitung

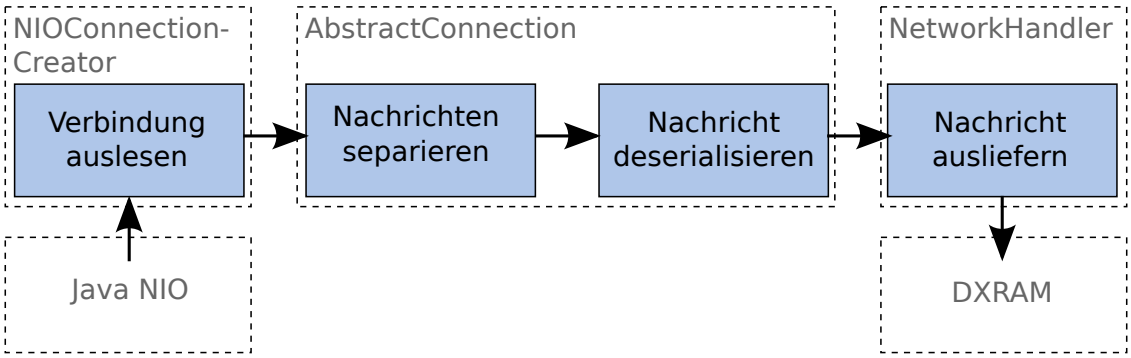



Abbildung 4.6: Aufgabenfolge beim Empfangen

vorliegen. Somit ist es nicht nötig die Verbindungen explizit nach neuen Daten abzufragen. An diesem Punkt wartet bisher der Worker Thread im `NetworkHandler` darauf seine Arbeit zu verrichten. Ähnlich wie auch im `NIOConnectionCreator` lässt sich diese Stelle als Dispatcher nutzen, sodass die weitere Bearbeitung über Event Handler realisiert wird.
Zunächst einmal werden jedoch die einzelnen Arbeitsschritte des Workers aufgeteilt. Der Datenstrom wird nun innerhalb der `AbstractConnection` Klasse verarbeitet. Dies ist sinnvoll, weil der Datenstrom ein individueller Teil einer Verbindung ist. Eine zentrale Verarbeitung aller Datenströme bietet sich deswegen nicht an. Auch das eigentliche Deserialisieren der Nachrichten kann innerhalb der `AbstractConnection` geschehen, weil auch dieser Schritt unabhängig von anderen Verbindungen, oder dem restlichen System, getätigt werden kann. Somit empfängt der `NetworkHandler` nur noch fertige Nachrichten, welche an die dazugehörigen registrierten Empfänger ausgeliefert werden können.

Somit ergibt sich eine Aufteilung in drei Teile:

- Unterteilung des Datenstroms in separate Nachrichten
- Deserialisierung der einzelnen Nachrichten
- Auslieferung der Nachrichten an die Anwendung

Für die Implementierung bedeutet das konkret, dass der `IncomingMessageCrea-`

tor aus dem Worker Thread der NetworkHandler Klasse extrahiert wird. Im NetworkHandler existiert nun lediglich noch ein MessageHandler, welcher auf fertige Nachrichten wartet und diese an DXRAM ausliefert. Der IncomingMessageCreator zieht in die AbstractConnection um und wird dabei in den ByteStreamInterpreter und dem MessageCreator aufgeteilt. Der ByteStreamInterpreter untersucht nun nur noch den eingehenden Datenstrom und gibt einzelne Nachrichten in ihrer serialisierten Form zurück. Diese werden dann im zweiten Schritt durch den MessageCreator deserialisiert.

Zusammen mit dem Auslesen teilt sich der komplette Empfangsprozess nun auf vier separate Aufgaben auf. Jede ist grundsätzlich unabhängig von den anderen Schritten. Somit könnten theoretisch bis zu vier unabhängige Threads parallel an einer Verbindung arbeiten. Rechnet man den ehemaligen Worker Thread mit, welcher den Selector abruft, sind es sogar fünf. Mit steigender Verbindungsanzahl potenziert sich entsprechend das Potenzial an parallel ausführbaren Aufgaben. Jedoch zeigt sich hier auch, warum es notwendig war den TaskExecutor mit der Möglichkeit auszustatten einfache Abhängigkeiten unter den Aufgaben abzubilden. Namentlich betrifft das den ByteStreamInterpreter. Die Untersuchung eines Datenstroms kann weiterhin nur sequenziell geschehen, weil nicht bekannt ist, wie groß eine Nachricht im einzelnen ist. Somit kann auch nicht bestimmt werden, ab welchem Punkt eine weitere Aufgabe in einem anderen Thread ansetzen könnte. Um also keine anderen Aufgaben durch blockierende Threads auszubremsen, wird die Aufgabe des ByteStreamInterpreters als Teil einer sequenziellen Aufgabengruppe behandelt. Als Gruppen ID für den TaskExecutor dient hier die Node ID des Verbindungspartners. Über diese lässt sich eine Verbindung eindeutig identifizieren. Entsprechend befinden sich auch nur die eigenen Aufgaben unter dieser ID im TaskExecutor.

### 4.2.5 Einsatz verschiedener Thread Pools

Während der Nachrichtenverarbeitung durchlaufen die Daten, wie oben beschrieben, mehrere Stationen. In diesen Stationen werden die Daten in Form von Aufgaben durch den TaskExecutor bearbeitet. Bisher ungenannt ist dabei, dass jede Station eine eigene Instanz der TaskExecutor Klasse besitzt. Der NetworkHandler und der NIO-

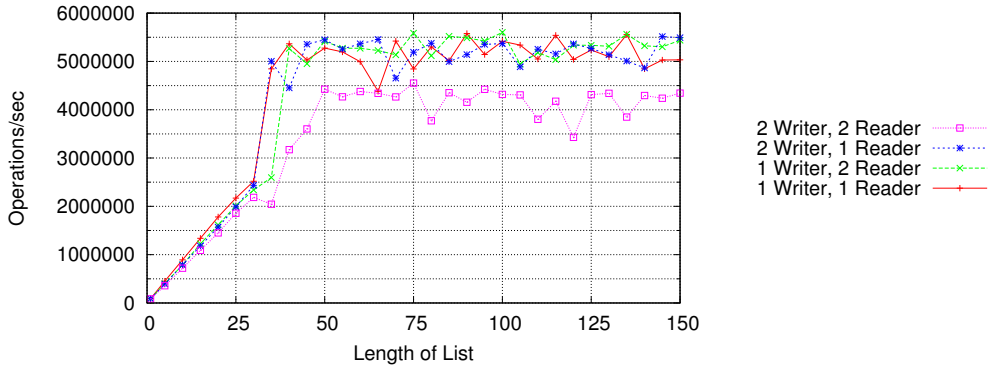



Abbildung 4.7: Listendurchsatz bei Zugriff durch mehrere Threads

`ConnectionCreator` halten ihre eigenen Instanzen. Die Verarbeitung innerhalb der `AbstractConnection` geschieht über den allgemeinen Thread Pool, welcher über den `getDefaultExecutor()` Aufruf im `TaskExecutor` erreicht werden kann. Der Einsatz mehrerer Thread Pools hat zwei Gründe. Der erste ist rein praktischer Natur. Durch den eigenen Thread Pool für die Auslieferung der Nachrichten im `NetworkHandler` lässt sich einstellen wie viele Threads Daten an DXRAM ausliefern sollen. Sind im Rahmen der Nachrichtenverarbeitung innerhalb von DXRAM aufwendigere Operationen notwendig, eventuell sogar mit blockierenden Aufrufen, kann man die Anzahl der Threads erhöhen. Sind weniger parallele Zugriffe gewünscht, weil Aufgaben häufig kritische Abschnitte mit beschränktem Zugriff durchlaufen, kann die Threadnummer reduziert werden um Ressourcen zu sparen. Des Weiteren wird durch den separaten Thread Pool der eigentliche Arbeitsablauf im inneren der Netzwerkkomponente nicht beeinflusst. Dies war der Fall in der bisherigen Version, als der Worker Thread im `NetworkHandler` sowohl für die Deserialisierung, als auch für die Auslieferung zuständig war. Es konnten also keine Nachrichten gleichzeitig ausgeliefert und deserialisiert werden. Ebenfalls verhindert die Trennung nun, dass alle verfügbaren Threads gleichzeitig Nachrichten ausliefern und sich die empfangen Daten anstauen.

Der zweite Grund ist, dass auf diese Weise ein Leistungsengpass umgangen wird. Um dieses Argument nachvollziehen zu können muss man wissen, dass der `ExecutorService` eine `BlockingQueue`[9] zum Arbeiten benötigt, welche als Warteschlange

[9] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html`

für Aufgaben genutzt wird. Die Executors Factory benutzt dafür standardgemäß eine `LinkedBlockingQueue`. Wie auch für die meisten anderen Datenstrukturen gibt es insbesondere beim synchronisierten Zugriff durch mehrere Threads ein Durchsatzlimit, welches nicht überschritten wird. Dieses Limit wurde in Abbildung 4.7 ermittelt, auf dem gleichen System, welches in Abschnitt 5.1 für die Testumgebung genutzt wird[10]. Dieses Limit liegt im Falle von vier gleichzeitig zugreifenden Threads bei etwas unter 4'500'000 Operationen pro Sekunde. Geht man davon aus, dass etwa fünf Aufgaben pro Nachricht anfallen, könnte man mit einem einzelnen Task Executor etwa 900'000 Nachrichten pro Sekunde verarbeiten. Nimmt man weiterhin eine gesamte Nachrichtengröße von 64 Byte (inkl. Header) an, wäre der Datendurchsatz bei einem Wert von etwa 57 MB/s limitiert. Der theoretische Maximaldurchsatz von Gigabit Ethernet liegt bei etwa 125 MB/s. Dieser Wert hängt natürlich stark von der tatsächlich Nachrichtengröße ab. Dennoch zeigt er, dass ein einzelner `ExecutorService` nicht die Leistung bietet, um im typischen Größenbereich von DXRAM performant zu arbeiten.

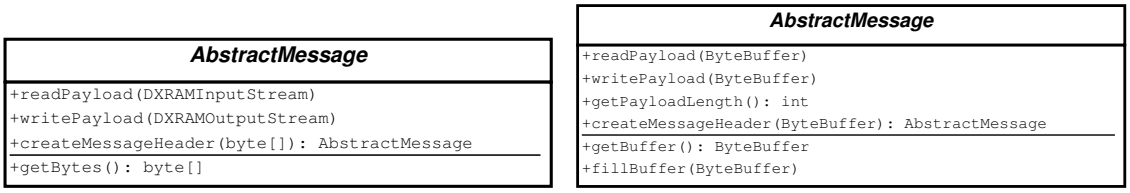
Dieses einfache Rechenbeispiel zeigt, dass mindestens drei Instanzen der `TaskExecutor` Klassen zum Einsatz kommen müssen, um auf dem Testsystem genügend Leistungsreserven zu bieten. Bei der gewählten Architektur, mit drei zu durchlaufenden Klassen, bietet sich die Einteilung in drei Thread Pools somit auch an.

## 4.3 Verbesserte (De-)Serialisierung

Wie in Unterabschnitt 3.2.1 beschrieben mussten die Daten beim Serialisieren der Nachricht häufig umkopiert werden, was auf die Nutzung der `InputStream` Klasse zurückzuführen ist. Weil für Java NIO sowieso `ByteBuffer` verwendet werden müssen, bietet es sich an die Nachricht direkt in einen solchen Buffer zu serialisieren. Aus diesem Grund wurde die `AbstractMessage` Klasse entsprechend überarbeitet.

In Abbildung 4.8 sind die Änderungen aufgelistet. Die `getBytes()` Methode (siehe Listing 3.3) wurde entfernt. Stattdessen kommt nun die `getBuffer()` Methode zum Einsatz. Diese ist in Listing 4.4 abgebildet. Hier wird nun auf einen `OutputStream`

[10] Das Testprogramm dazu ist in Anhang A zu sehen.




(a) Bisherige *AbstractMessage* (b) Neue *AbstractMessage*

Abbildung 4.8: Unterschiede in der *AbstractMessage*

verzichtet und direkt ein `ByteBuffer` der finalen Größe erzeugt. Der Buffer wird dann über die, ebenfalls neue, Methode `fillBuffer()` mit den Daten der aktuellen Nachricht beschrieben. Um die benötigte Größe des Buffers vorab bestimmen zu können wurde die `getPayloadLength()` Methode der `AbstractMessage` hinzugefügt. Diese muss von Klassen implementiert werden, welche die `AbstractMessage` erweitern. Diese gibt zurück, wie viele Bytes an Nutzdaten mit der aktuellen Nachricht übermittelt werden sollen. Diesem Wert wird dann der konstante Nachrichtenheader hinzuaddiert. Ebenfalls haben sich auch die Parameter der `readPayload()` und `writePayload()`

```
public final ByteBuffer getBuffer() {
    int payloadSize = getPayloadLength();

    ByteBuffer buffer = ByteBuffer.allocate(HEADER_SIZE + payloadSize)
        ;
    buffer = fillBuffer(buffer);
    buffer.flip();

    return buffer;
}

public final ByteBuffer fillBuffer(final ByteBuffer p_buffer) {
    p_buffer.putLong(m_messageID);
    p_buffer.put(m_type);
    p_buffer.put(m_subtype);
    p_buffer.put(m_ratingValue);
    p_buffer.putInt(getPayloadLength());

    writePayload(p_buffer);

    return p_buffer;
}
```

Listing 4.4: *getBuffer()* und *fillBuffer()* in *AbstractMessage*

```
protected static AbstractMessage createMessageHeader(final ByteBuffer
    p_buffer) throws NetworkException {
     AbstractMessage ret;
     long messageID;
     byte type, subtype, ratingValue;

     Contract.checkNotNull(p_buffer, "no bytes given");

     if (p_buffer.remaining() < HEADER_SIZE - BYTES_PAYLOAD_SIZE) {
         throw new NetworkException("Incomplete header");
     }

     messageID = p_buffer.getLong();
     type = p_buffer.get();
     subtype = p_buffer.get();
     ratingValue = p_buffer.get();

     try {
         ret = MessageDirectory.getInstance(type, subtype);
     } catch (final Exception e) {
         throw new NetworkException("Unable to create message", e);
     }

     ret.m_messageID = messageID;
     ret.m_type = type;
     ret.m_subtype = subtype;
     ret.m_ratingValue = ratingValue;
     return ret;
}
```

Listing 4.5: *createMessageHeader()* in *AbstractMessage*

Methoden geändert. Diese Methoden werden von erbenden Klassen überschrieben um ihre Nutzdaten zu serialisieren, bzw. zu deserialisieren. Auch hier werden nun `ByteBuffer` anstatt `InputStream` oder `OutputStream` Klassen verwendet.

Genau wie die Serialisierung wurde auch die Deserialisierung vollständig von Byte Arrays und Streams auf ByteBuffer umgestellt. Mit `createMessageHeader()` wurde auch die letzte Methode in `AbstractMessage` umgestellt. Diese Methode profitiert ebenfalls von der Einführung der `MessageDirectory` Klasse in Abschnitt 4.1. Weiterhin wurde der in Unterabschnitt 3.1.4 beschriebene `PayloadHandler` entfernt. Der `ByteStreamInterpreter` verwaltet nun die Nutzdaten der einzulesenden Nachricht selber in einem Buffer.

Durch den Verzicht auf Streams wurden einige Kopiervorgänge pro Nachricht entfernt. Insbesondere auf das Verhalten der Garbage Collection hat die reduzierte Anzahl allozierter Objekte eine positive Auswirkung. Generell verhindern lassen sich Kopiervorgänge durch das Reactor Pattern nicht. Grundsätzlich empfiehlt sich für Java NIO der Einsatz von `DirectByteBuffer` Objekten. Im Gegensatz zu den normalerweise verwendeten `HeapByteBuffer` Objekten liegen `DirectByteBuffer` Objekte außerhalb des Speicherbereichs, welcher von der Garbage Collection überwacht wird[11]. Diese werden spätestens für die nativen Methoden innerhalb von Java NIO benötigt, sodass Java NIO alle `HeapByteBuffer` in `DirectByteBuffer` umkopiert. Der Nachteil ist jedoch, dass `DirectByteBuffer` teuer in der Allokation sind, weswegen sie sich nur für langlebige Objekte eignen. Für die Verwendung in Java NIO bedeutet dies, dass `DirectByteBuffer` in einem Cache gespeichert werden und wiederverwendet werden müssten. Für diese Kopiervorgänge hat Java NIO intern einen solchen `BufferCache` implementiert. Innerhalb der eigenen Implementierung bietet sich aufgrund der Abkopplung von Serialisieren und Senden so ein Cache nicht an. Um einen großen Teil der zu serialisierenden Nachrichten in einen `DirectByteBuffer` zu schreiben müsste dieser Cache sehr viele Buffer vorhalten. Die Verwaltung und Suche nach einem geeigneten `DirectByteBuffer` würde schließlich mehr Zeit beanspruchen als der eingesparte Kopiervorgang innerhalb von Java NIO. Deswegen sind die verwendeten Buffer in dieser Implementierung fast ausschließlich `HeapByteBuffer`. Ausnahme bilden hier einige Buffer, welche im `ByteStreamInterpreter` zum Zwischenspeichern der Nachrichtenheader genutzt werden. Da diese wiederverwendet werden können, kommen hier `DirectByteBuffer` zum Einsatz.

## 4.4 Überlastkontrolle

Grundsätzlich ist eine eigene Überlastkontrolle in einem Netzwerkmodul nicht notwendig, sofern man auf TCP zurück greift. TCP hat bereits von Haus aus alle notwendigen Mechanismen implementiert, um sowohl auf ein überlastetes Netzwerk, als auch einen ausgelasteten Empfänger zu reagieren. Jedoch wird das System durch das Reactor

[11] Vgl. *Direct vs. non-direct buffers* in `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html`

Pattern mit Java NIO gewissermaßen ausgehebelt. Das Problem ist, dass die Netzwerkschnittstelle durch Java NIO nicht direkt angesprochen wird. Alle ein- und ausgehenden Daten werden zwischengespeichert und zu einem späteren Zeitpunkt weiterverarbeitet. Für die Anwendung ist es ohne Weiteres möglich schneller Daten zu erzeugen als durch das Netzwerk abgeführt werden können. Diese Daten werden in der Verbindung in die Warteschlange für ausgehende Daten eingehangen, welche von zuständigen Threads abgearbeitet und versendet wird. Dadurch, dass mehr Daten in die Warteschlange eingefügt werden als durch das Netzwerk gesendet werden können, wächst die Warteschlange immer weiter an. Limitiert man den Zugriff beim Senden nicht, kann es passieren, dass auf diese Weise der Anwendungsspeicher vollläuft.

Ein ähnliches Problem liegt auf der Empfängerseite vor. Dort werden die Daten von Java NIO ausgelesen und ebenfalls für die weitere Verarbeitung zwischengespeichert. Hierbei muss angemerkt werden, dass das Empfangen von Nachrichten ungleich aufwendiger ist, als das einfache Senden. Beim Senden müssen die Nachrichtenobjekte lediglich in einen ByteBuffer geschrieben und an Java NIO übergeben werden. Der Empfänger muss, wie in Unterabschnitt 3.1.1 beschrieben, den eingehenden Datenstrom jedoch vollständig untersuchen um die empfangen Bytes den einzelnen Nachrichten zuordnen zu können. So kann es passieren, dass der Empfänger schneller Nachrichten empfängt als er diese verarbeiten kann, und die entsprechenden Warteschlangen ebenfalls immer weiter anwachsen.

Eine Beschränkung auf Seiten des Senders wäre einfach umzusetzen, da alle dafür benötigten Daten lokal vorliegen. Sendende Aufrufe ließen sich verzögern, bis wieder Platz in der Warteschlange frei ist, oder eine definierte Anzahl Bytes durch Java NIO versendet wurde. Das Problem auf Seiten des Empfängers ist jedoch nicht ohne Weiteres zu lösen. Die TCP Überlastkontrolle greift in diesem Fall nicht, weil durch das Abrufen der Daten von Java NIO durch das Netzwerkmodul die Zwischenspeicher für das Netzwerkprotokoll geleert werden. Somit hat es für TCP den Anschein, dass die Daten schnell genug verarbeitet werden können und keine Überlast vorliegt. Damit der Sender weiß wie schnell er seine Daten senden kann, muss er eine Art Rückmeldung vom Empfänger erhalten. Aus diesem Grund wurde mit der `FlowControlMessage` eine eigene Flusskontrolle implementiert.

Das Ziel der `FlowControlMessage` ist es, dem Sender mitzuteilen wie viele Bytes durch den Empfänger bearbeitet werden konnten. Ähnlich wie in TCP durch ACK Signale bestätigt der Empfänger dem Sender, wie viele Bytes seit der letzten `FlowControlMessage` verarbeitet wurden. Dadurch ist es dem Sender bei Erreichen eines vorab definierten Schwellenwerts möglich sendende Aufrufe zu verzögern. Diese Zeit kann der Empfänger nutzen um seinen Zwischenspeicher abzubauen. Hat der Empfänger etwa die Hälfte des definierten Schwellwerts an Bytes verarbeitet, setzt dieser eine `FlowControlMessage` an den Sender ab. Die Hälfte des Schwellwerts wurde ausgewählt, da ein unnötig frühes Senden dieser Nachrichten dazu führt, dass ansonsten mehr Netzwerkbandbreite für Kontrollnachrichten beansprucht wird. Werden die Nachrichten jedoch erst sehr spät abgesetzt, agiert der Sender stets sehr nah am Schwellwert für unbestätigte Bytes. Dadurch können leichte Schwankungen in der Übertragungs- oder Verarbeitungsgeschwindigkeit bereits ein unnötiges Aussetzen der Übertragung zur Folge haben. Entsprechend sollte das Sendefenster, also die Anzahl unbestätigter Bytes, hinreichend groß gewählt werden, um sowohl den Anteil an Kontrollnachrichten gering zu halten, als auch einer stockenden Übertragung vorzubeugen. Im Idealfall sollte das Sendefenster so gewählt werden, dass eine Verbindung mehrere Sekunden ohne Unterbrechung die Netzwerkbandbreite ausschöpfen kann.

Weiterhin ist anzumerken, dass dieses Problem hauptsächlich auf die sehr kleinen Nachrichten zurückzuführen ist. Der Aufwand beim Verarbeiten des Datenstroms ist das Auffinden der Nachrichtenheader. Dieser Aufwand steigt linear mit der absoluten Nachrichtenanzahl. Mit der Nachrichtengröße steigt lediglich der Kopieraufwand. Weil Kopiervorgänge innerhalb des Arbeitsspeichers naturgemäß schneller als Netzwerkübertragungen sind, und bei steigender Nachrichtengröße die absolute Nachrichtenrate sinkt, verschiebt sich der Engpass mit wachsender Nachrichtengröße stetig in Richtung der Netzwerkübertragung. Somit dient der hier vorgestellte Mechanismus hauptsächlich als Schutz vor Überlast durch sehr viele sehr kleine Nachrichten. Auf der Netzwerkschicht kommen weiterhin die Kontrollmechanismen von TCP zu tragen.

## 4.5 Sonstige Optimierungen

Neben den vorgestellten Änderungen, welche Struktur und Ablauf angepasst haben, wurden auch noch kleinere Anpassungen vorgenommen. Überwiegend betrifft dies kleine Aufräumarbeiten. Nennenswert für die Leistungsfähigkeit ist jedoch, dass die in Unterabschnitt 3.2.5 angesprochenen Log4J Aufrufe aus den leistungsrelevanten Methoden entfernt wurden. Logging Aufrufe sind jetzt nur noch in den Initialisierungsmethoden und im Fehlerfall zu finden.
Ebenfalls relevant ist, dass nun `ArrayDeques`[12] anstatt von `LinkedList` für die Verwaltung von Warteschlangen verwendet werden. Anstelle von objektbasierten arbeitet die `ArrayDeques` auf Basis eines Arrays. Das bedeutet, dass zu Beginn des Programms Kopieraufwand anfallen kann, weil das interne Array angepasst werden muss. Hat die Liste jedoch das Fassungsvermögen für die typische Länge erreicht, fällt für die Garbage Collection praktisch keine Arbeit mehr an. Im Gegensatz zu den Objekten in der `LinkedList` werden die Array Einträge wiederverwendet. Des Weiteren bietet die `ArrayDeques` eine Zugriffszeit von $\mathcal{O}(1)$ für die meisten Operationen.

Eine weitere unscheinbare Änderung betrifft den schreibenden Zugriff auf den Java NIO `SocketChannel`. Diesem kann über die `write()` Methode ein einzelner `ByteBuffer` übergeben werden, welcher über die Verbindung gesendet werden soll. Weiterhin ist es auch möglich direkt ein Array aus Buffern zu übergeben, sodass alle übermittelt werden. Interessant ist, dass es bei diesen beiden Überladungen der `write()` Methode signifikante Unterschiede in der Leistung gibt. Die Auswirkungen der verschiedenen Aufrufe werden in Unterabschnitt 5.2.5 gezeigt.

[12] `http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html`

# Kapitel 5

# Evaluierung

In diesem Kapitel werden die Änderungen aus Kapitel 4 mit der Version vor den Änderungen verglichen. Hierzu werden Messungen durchgeführt, welche die Anzahl der verarbeiteten Nachrichten pro Sekunde ermitteln. Aus diesen Werten lässt sich der Netto Datendurchsatz ableiten (reine Nutzdaten ohne Nachrichtenheader). Weiterhin werden die Antwortzeiten auf Nachrichten bestimmt.

## 5.1 Testumgebung

Als Testumgebung kam der Lehrstuhl interne Cluster zum Einsatz. Je nach Testszenario wurden bis zu neun Knoten beansprucht. Die Knoten waren mit der in Tabelle 5.1 beschriebenen Konfiguration ausgestattet. Alle verwendeten Knoten befinden sich im selben Schrank und sind über ein 1000 Mbit/s Switched Gigabit Ethernet Netzwerk miteinander verbunden.

| Prozessor (CPU) | Intel Xeon E3-1220, 4 x 3,1 GHz |
|---|---|
| Arbeitsspeicher (RAM) | 16 GB (4 x 4 GB) |
| Betriebssystem | NodeFS (Debian “Wheezy” 7.6) |
| Kernel Version | 3.2.0 |
| Java Version | OpenJDK 1.7.0_65 |

Tabelle 5.1: Rechnerkonfiguration

## 5.2 Auswertung

Für die Auswertung wird ein Stresstest betrachtet, bei dem ein Knoten als Server fungiert und von anderen Knoten mit Daten versorgt wird. Die Unterteilung in *Server* und *Client* wird für diesen Testfall nur als Verständnishilfe verwendet. Die eigentliche Netzwerkkomponente unterscheidet nicht zwischen beiden Fällen und fungiert stets in beiden Rollen.
Um eine möglichst hohe Last zu erzeugen, senden die Clients ihre Daten in einer Schleife, womit simuliert wird, dass sie ihre Daten möglichst schnell über das Netzwerk versenden wollen. Der Server empfängt die Nachrichten und “verarbeitet” diese. Anschließend sendet er für jede empfangene Nachricht eine Empfangsbestätigung in Form einer weiteren Nachricht an den ursprünglichen Absender. Das Testprogramm wird mit Hilfe eines Scripts gleichzeitig auf allen teilnehmenden Knoten gestartet.

Zur Messung wird im Testprogramm ein `Timer`[1] verwendet, welcher zyklisch jede Sekunde ausgeführt wird. Dieser Timer ermittelt in jedem Zyklus wie viele Nachrichten versendet wurden und gibt die Differenz zum letzten Durchgang über die Standardausgabe aus. Der Timer ist sehr genau bei der zeitlichen Ausführung und weicht in der Regel nicht über eine Millisekunde von der gewünschten Wartezeit ab. Es kann jedoch bei einer sehr aktiven Garbage Collection dazu kommen, dass der Timer deutlich von der Zeitplanung abweicht. Die Messgenauigkeit wird dadurch jedoch nicht eingeschränkt. Durch einen hohen Aufwand für die Garbage Collection kommt es auch bei der Nachrichtenverarbeitung zu großen Schwankungen. Ist die Garbage Collection genau zwischen den Aufrufen des Timers aktiv, gibt es keine ungewöhnliche Abweichung. Erst wenn die Aufräumphase kurz vor der planmäßigen Ausführung des Timers startet kommt es zu einer

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/Timer.html`

Verzögerung. In dieser Zeit werden ebenfalls keine Nachrichten verarbeitet und der ausgegebene Wert entspricht weiterhin der Anzahl verarbeiteter Nachrichten. Jedoch ist der gemessene Zeitraum größer als die übliche Sekunde, womit die reale Anzahl an Nachrichten pro Sekunde geringer wäre, und sich der zeitliche Ablauf verschieben würde. Eine Verschiebung im Ablauf schränkt jedoch die allgemeine Vergleichbarkeit zwischen den Messungen nicht ein, da das Testszenario eine konstante Last über den kompletten Zeitraum vorsieht.

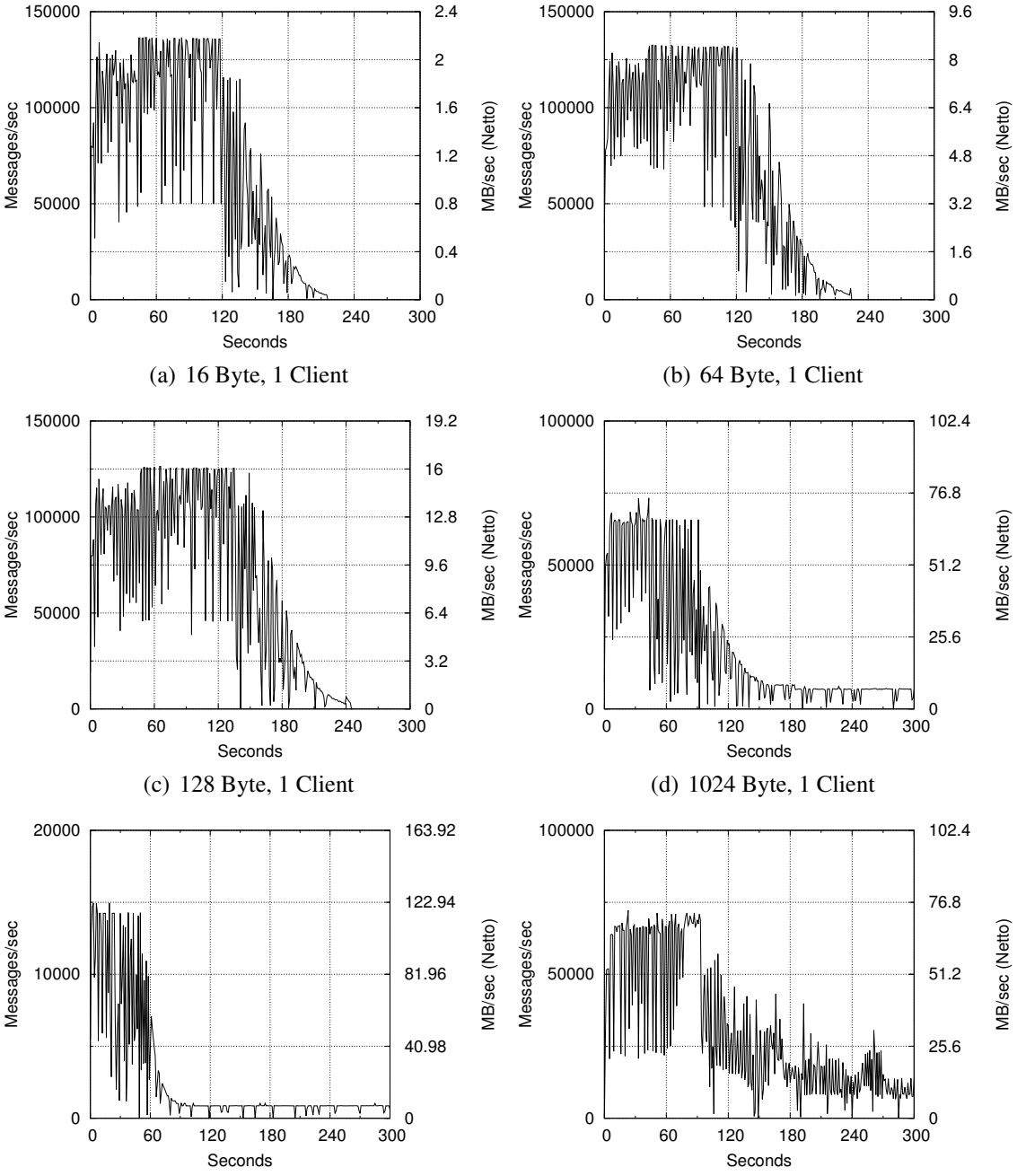
Das gewählte Testszenario spiegelt nicht unbedingt die realen Einsatzszenarien wider und stellt eher eine Ausnahme als die Regel dar. Jedoch ergibt sich durch diese Testweise ein Wert, welcher Aufschluss darauf gibt, wie viele Anfragen pro Sekunde bearbeitet, und wie viele Daten verarbeitet werden können. Zusätzlich lässt sich ermitteln, wie sich die Netzwerkkomponente unter maximaler Auslastung verhält. Zudem würde die Entwicklung eines realitätsnahen Verfahrens zum Test eines verteilten Systems den Rahmen dieser Masterarbeit sprengen.

Um die Antwortzeiten zu ermitteln, sendet der Client einen `AbstractRequest` an den Server und misst die Zeit bis die Antwort vorliegt. Erst dann wird die nächste Anfrage abgesetzt. Die Messungen beginnen auf Seiten des Clients unmittelbar vor dem Senden der Nachricht.

### 5.2.1 Konfiguration

Für die Messungen der neuen Version kommen vier Threads zum Einsatz. Diese Teilen sich so auf, dass für jede der vier Hauptaufgaben je ein Thread zur Verfügung steht. Die Bearbeitung der NIO Operationen werden dabei zusammen gefasst, da diese Aufgaben grundsätzlich nur Daten kopieren und ohne aufwendige Logik auskommen. Somit steht den Thread Pools der Klassen `NetworkHandler` und `NIOConnectionCreator` jeweils ein Thread zu Verfügung, während der globale Thread Pool für die `AbstractConnection` zwei Threads beinhaltet. Um einen möglichst geringen Einfluss auf die Verbindungen zu haben, gilt für die Flusskontrolle ein Sendefenster von 100 MB. Für alle Testreihen beider Versionen stehen dabei 12 GB Anwendungsspeicher zur Verfügung, welche von Beginn an vollständig der Anwendung zugewiesen sind.

Sofern nicht explizit angegeben werden die hier genannten Parameter für alle vorgestellten Testreihen angewandt.




(a) 16 Byte, 1 Client

(b) 64 Byte, 1 Client

(c) 128 Byte, 1 Client

(d) 1024 Byte, 1 Client

(e) 8192 Byte, 1 Client

(f) 1024 Byte, 4 Clients

Abbildung 5.1: Netzwerkdurchsatz der alten Version

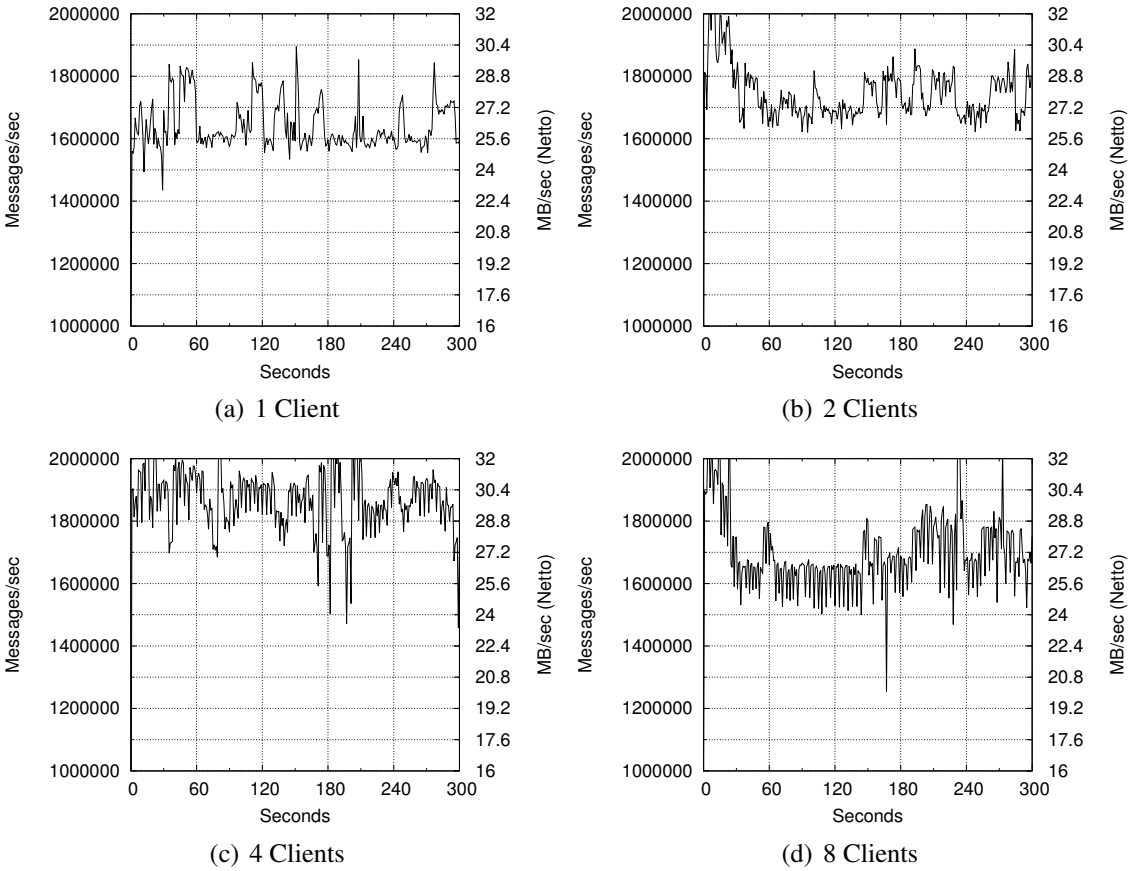
### 5.2.2 Datendurchsatz

Um die Leistungsbeeinflussung der in dieser Arbeit vorgestellten Änderungen einordnen zu können, bietet sich ein Vergleich mit der alten Version der Netzwerkschnittstelle an. In Abbildung 5.1 sind einige Testreihen mit verschiedenen Parametern abgebildet. Der oben beschriebene Stresstest wurde mit verschiedenen Nutzdatengrößen vollzogen, beginnend mit 16 Byte bis hoch zu 8 kB pro Nachricht (ohne Nachrichtenheader), welche von einem einzelnen Client gesendet werden. Um die Auswirkungen von parallel aktiven Verbindungen zu messen wurde der Test mit 1024 Byte Nutzdaten ebenfalls mit vier Clients vollzogen.

Die erste Gemeinsamkeit der Testreihen ist, dass alle Versuche massive Leistungseinbrüche aufweisen. Je größer die Nachrichten sind, desto eher bricht auch der Durchsatz ein. Der Grund dafür ist die in Abschnitt 4.4 beschriebene Problematik, dass eingehende Daten nicht schnell genug verarbeitet werden können und die Anzahl der zwischengespeicherten Daten immer weiter anwächst. Als direktes Resultat dessen ist die Garbage Collection überwiegend damit beschäftigt Speicher zu finden, welcher freigegeben werden kann. Weil jedoch der meiste Speicher durch wartende eingehende Daten belegt wird, kann nur sehr wenig bis gar kein Speicher freigegeben werden. Somit muss immer mehr Rechenzeit für die Garbage Collection aufgewendet werden, bis am Ende die Ausführung des eigentlichen Programms nur noch einen Bruchteil der Zeit in Anspruch nimmt. Der Durchsatz bricht damit zwangsläufig ein.
Bei Testreihen mit kleineren Nachrichten resultiert dieses Verhalten in einem `OutOfMemoryError`[2] mit der Fehlermeldung *GC overhead limit exceeded*. Dieser Fehler tritt auf, wenn 98 % der Rechenzeit von der Garbage Collection in Anspruch genommen wird und dabei weniger als 2 % des Heaps freigegeben werden kann. Folglich stürzt das Programm ab. Bei größeren Nachrichten stürzt das Programm zwar nicht ab, der Datendurchsatz ist jedoch konstant sehr gering.

Weiterhin fällt auf, dass der Durchsatz sehr stark schwankt und auch erst bei sehr großen Nachrichten die Bandbreite von Gigabit Ethernet ausnutzen kann. Die Schwankungen lassen sich dabei ebenfalls auf die Aktivitäten der Garbage Collection zurückführen. Ein

[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html`



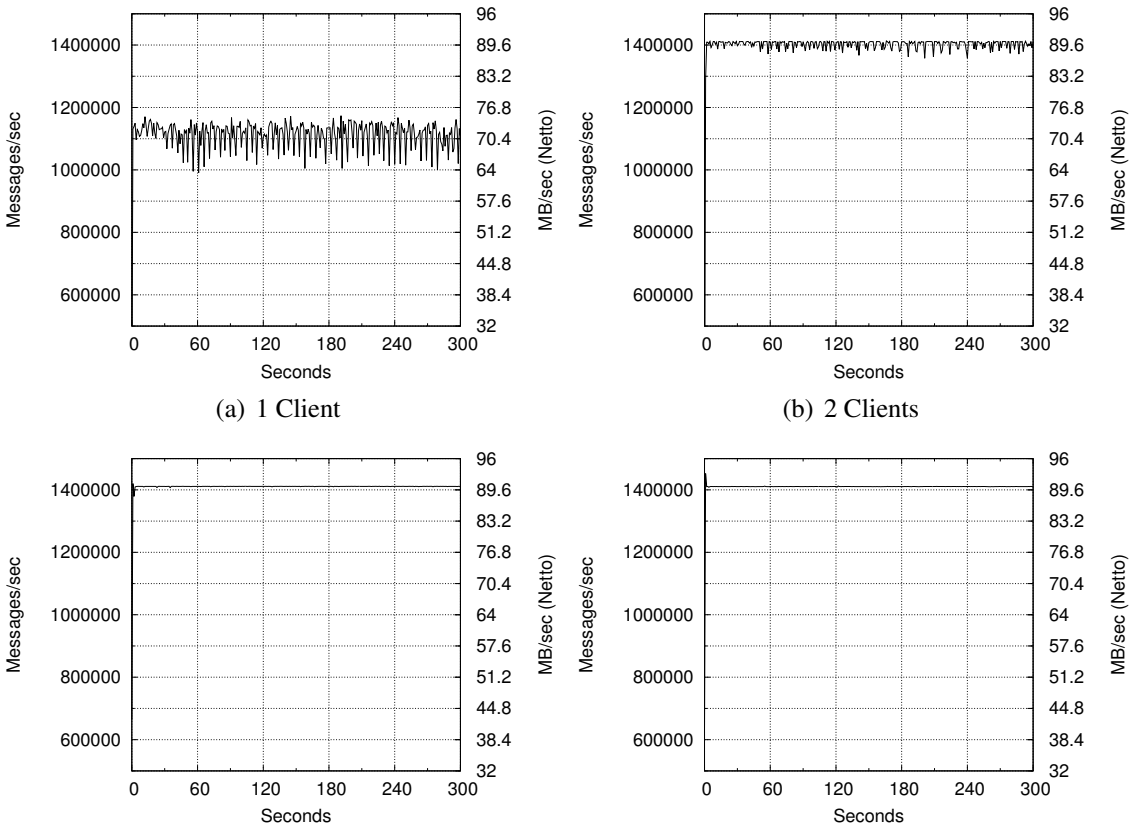

(a) 1 Client (b) 2 Clients

(c) 4 Clients (d) 8 Clients

Abbildung 5.2: Netzwerkdurchsatz bei 16 Byte Payload

weiterer Grund findet sich in der `write()` Methode des Channels. Wie in Unterabschnitt 5.2.5 zu sehen sein wird, weist der bisher verwendete Aufruf bereits von sich aus eine große Streuung auf.
Bei den kleinen Nachrichten sieht man, dass die Netzwerkkomponente einen maximalen Nachrichtendurchsatz von ca. 130'000 Nachrichten pro Sekunde nicht überschreitet. Dieser Wert kann jedoch nicht gehalten werden bis der Netzwerkdurchsatz zum limitierenden Faktor wird. Durch steigende Nachrichtengrößen erhöht sich auch der in Abschnitt 3.2 beschriebene Kopieraufwand, weswegen größere Nachrichten auch einen größeren Aufwand bedeuten.

Betrachtet man nun die Messungen der neuen Version fallen deutliche Unterschiede auf. In Abbildung 5.2 sind Messungen mit 16 Byte Nutzdaten mit einer wechselnden Anzahl von Clients zu sehen. Vergleicht man die Abbildungen 5.1(a) mit 5.2(a) sieht man,




(a) 1 Client

(b) 2 Clients

(c) 4 Clients

(d) 8 Clients

Abbildung 5.3: Netzwerkdurchsatz bei 64 Byte Payload

dass in der neuen Version der Nachrichtendurchsatz nicht mehr einbricht und sich relativ stabil über die Zeit hinweg verhält. Es kommt jedoch immer noch zu Schwankungen, welche durch eine sehr starke CPU Auslastung hervor gerufen werden. Das TCP/IP Protokoll erfordert bei schnellen Verbindungen grundsätzlich einen höheren Rechenaufwand [TTD06]. Hinzu kommt, dass zwischen 1,6 und 1,8 Millionen Nachrichten pro Sekunde verarbeitet werden. Für jede Nachricht müssen ByteBuffer alloziert werden, welche natürlich auch wieder von der Garbage Collection freigegeben werden müssen. Weil das Programm selber schon die CPU sehr gut auslasten kann, wirken sich die Garbage Collection Phasen direkt auf den Durchsatz aus. Aus demselben Grund können auch bei mehreren parallelen Verbindungen nicht mehr Daten verarbeitet werden. Aufgrund einer höheren CPU Belastung, hervorgerufen durch zusätzliche Verbindungen, steigen auch die Schwankungen sichtbar.

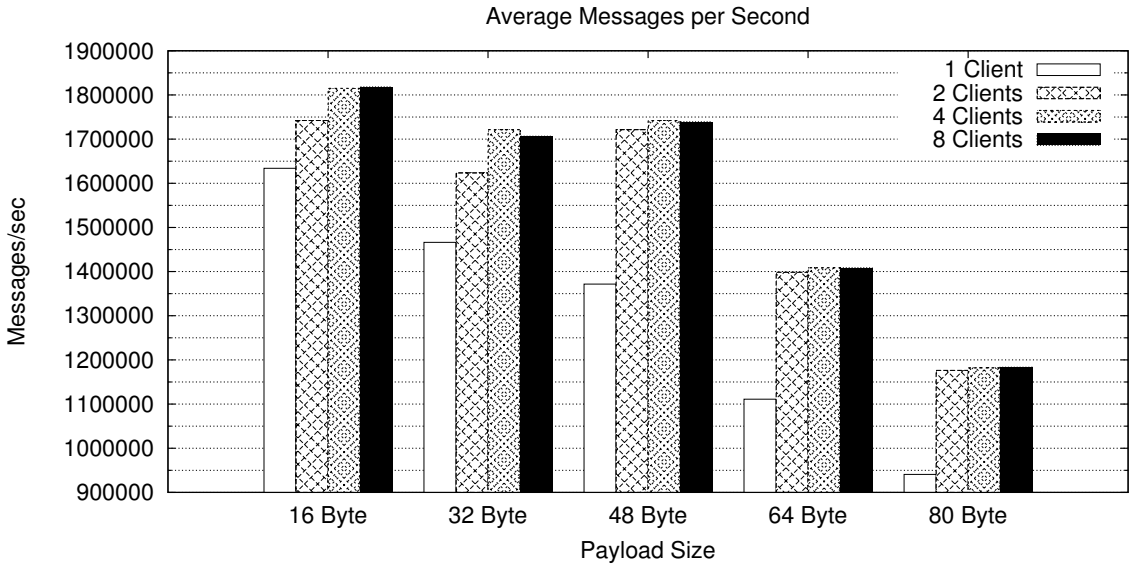



Abbildung 5.4: Durchschnittlicher Nachrichtendurchsatz

Vergleicht man den Netto Datendurchsatz, so konnte dieser deutlich gesteigert werden. Dieser ist von ehemals 2 MB/s auf etwa 25 MB/s gestiegen. Hierbei muss jedoch beachtet werden, dass zu jeder Nachricht noch 13 Byte Header hinzukommen. Die absolute Belastung des Netzwerks ist entsprechend fast doppelt so hoch.

Wie Abbildung 5.3 zeigt ist ab einer Nutzdatengröße von ca. 64 Byte die Grenze überschritten, an dem die verfügbaren Netzwerkkapazitäten ausgeschöpft sind, und die CPU nicht mehr länger der limitierende Faktor ist. Besonders interessant ist auch hier wieder der Fall mit einem Client. Dieser weist noch immer Schwankungen und, im Vergleich zu den Testreihen mit mehreren Clients, einen reduzierten Datendurchsatz auf. In diesem Fall liegen die Schwankungen auf Seiten des Clients. Neben den vier Threads, welche sich auf die Thread Pools aufteilen, ist der Hauptthread für die Erzeugung und anschließende Serialisierung der Nachrichten verantwortlich. Somit konkurrieren fünf aktive Threads um die vier Kerne des Testsystems. Durch die Garbage Collection kommen noch weitere Threads hinzu. Durch die Umschaltzeiten können also nicht durchgehend Nachrichten gesendet werden. Bei zunehmender Clientanzahl werden diese Schwankungen aus Serversicht auf die Nachrichtenrate kompensiert, sodass der Durchsatz konstant hoch ist.
Ebenfalls auffallend ist, dass die Datenrate bei einem Client deutlich geringer ist als

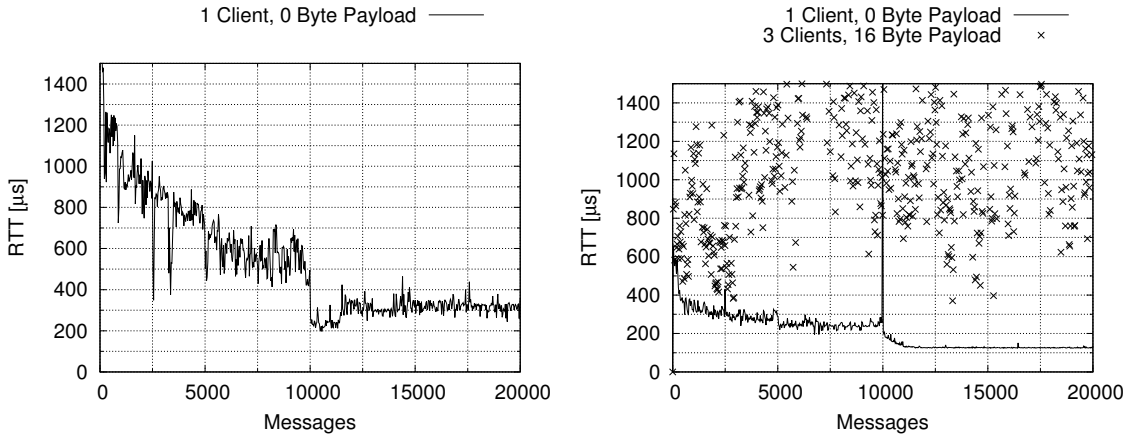
bei mehreren parallelen Verbindungen. Das liegt daran, dass Java NIO nicht mehr Daten auf die Leitung legen kann, sodass sich die Daten im Client im ausgehenden Zwischenspeicher aufstauen. Diese Beobachtung findet sich auch in anderen Arbeiten wieder [BOS04].

In Abbildung 5.4 ist der durchschnittliche Nachrichtendurchsatz für verschiedene Testreihen dargestellt. Auch hier spiegelt sich das Bild wider, dass eine einzelne Verbindung deutlich unter den theoretischen Möglichkeiten bleibt. Die Abweichungen zwischen den Testreihen mit mehreren Clients ergeben sich durch nicht deterministische Wechselwirkungen mit der Garbage Collection aufgrund der hohen CPU Auslastung. So fällt jede Testreihe in ihr eigenes Verhaltensmuster, welches um ihren eigenen Mittelwert herum schwankt. Testreihen zum Vergleich, als auch die Messungen für die weiteren Nachrichtengrößen sind in Anhang B zu finden.

### 5.2.3 Antwortzeiten

Neben dem Verhalten unter großer Last sind die Antwortzeiten sehr wichtig für die Arbeitsweise eines verteilten In-Memory Storage. Die benötigten Daten müssen in den meisten Fällen über das Netzwerk angefordert werden, weil diese selten bereits lokal vorliegen. Somit sind geringe Latenzen sehr wichtig für die Reaktionszeiten des gesamten Systems. In Abbildung 5.5 finden sich die Antwortzeiten für die beiden Versionen.

Die Antwortzeiten der alten Version liegen zu Beginn im Bereich von etwa 1,5 Millisekunden. Durch die Optimierungsvorgänge der Java Virtual Machine sinkt die mittlere Antwortzeit in den ersten 10'000 Nachrichten auf etwa 500 Mikrosekunden. Ab dort setzt dann die Java HotSpot Optimierung ein, womit sich die Antwortzeit auf etwa 300 Mikrosekunden einpendelt [KWM+08]. Betrachtet man daneben die Antwortzeiten der neuen Version, fallen diese deutlich geringer aus. Die Änderungen dieser Arbeit bewirken, dass bereits die ersten Antworten mit einer Zeit von etwa 600 Mikrosekunden deutlich schneller vorliegen. Dieser Wert liegt nach 2500 Nachrichten bereits unter 300 Mikrosekunden. Auch hier setzt die HotSpot Optimierung nach 10'000 Nachrichten ein. Die finale Antwortzeit liegt nun bei ca. 130 Mikrosekunden. Unmittelbar vor der Optimierung liegt eine einzelne Antwort erst nach etwa 1,5 Millisekunden vor. Eine solche




(a) Alte Version (b) Neue Version

Abbildung 5.5: Antwortzeiten

Verzögerung kann durch den Optimierungsvorgang selbst hervorgerufen werden, wenn etwa gerade ein Abschnitt neu kompiliert wird [LK05].

Wie in Abschnitt 5.2 beschrieben umfassen diese Messwerte den gesamten Weg der einzelnen Nachricht, angefangen beim Senden der Nachricht bis zum erfolgreichen Empfang der Antwort. Die ermittelten Messwerte liegen im direkten Umfeld von MPJ Express. MPJ Express ist eine Implementierung des *Message Passing Interface* vollständig in Java, welche ebenfalls auf Java NIO basiert. Abhängig von der verwendeten Konfiguration liegen die Antwortzeiten von MPJ Express zwischen 160 [BCS06] und 105 Mikrosekunden [TTD12]. Andere MPJ Varianten können jedoch durch den Einsatz angepasster Sockets bessere Ergebnisse erzielen. Beispielsweise erzielt F-MPJ Werte im Bereich von 50 Mikrosekunden. F-MPJ verzichtet auf die von Java mitgegebenen Netzwerkbibliotheken. Stattdessen werden *Java Fast Sockets* verwendet, welche eigens entwickelt wurden [TTD08]. Diese sind deutlich schneller als die Java eigenen Sockets. In [ZHH+07] wurde ermittelt, dass allein die Java Bibliotheken 50 Mikrosekunden für eine Antwort beanspruchen.

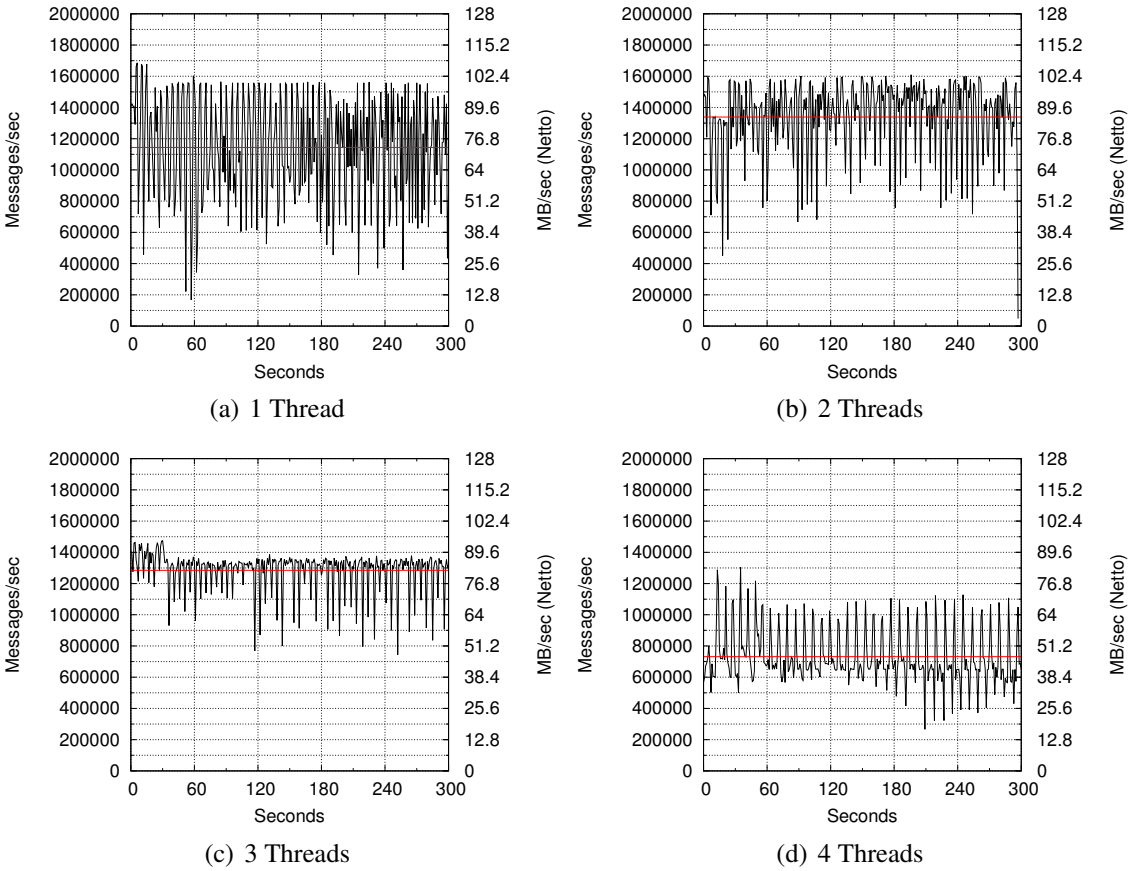
In Abbildung 5.5(b) ist ein zweiter Test zu sehen, welcher in Abbildung 5.5(a) fehlt. In diesem Test werden die Antwortzeiten ermittelt während der Server stark ausgelastet ist. Die Last wird in diesem Test durch zwei weitere Clients erzeugt, welche das Programm zum Durchsatztest durchlaufen und dem Server Nachrichten mit 16 Byte Nutzdaten zu-

senden. Dieser Test wird für die alte Version nicht abgebildet, weil die alte Version starke Probleme mit diesem Testfall hatte. Selbst bei einem Nachrichten-Time-Out von 20 Sekunden konnten keine Antworten empfangen werden. In der aktuellen Version lagen die Antworten auch unter Last in den meisten Fällen in unter 1,5 Millisekunden vor. Nur vereinzelnd wurde diese Zeit überschritten. Jedoch unterliegen hier die Antwortzeiten einem starken Streufaktor und weisen kein einheitliches Muster auf.

### 5.2.4 Multithreading

Um die Auswirkungen von einer unterschiedlichen Anzahl von Threads darzustellen, wurde zusätzlich noch eine Version getestet, welche statt den drei Thread Pools einen zentralen Thread Pool einsetzt. Die in Abbildung 5.6 gezeigten Ergebnisse entstanden in einem Test mit vier Clients und Nachrichten mit 64 Byte Nutzdaten. Die mittlere Durchsatzrate ist hier in rot eingezeichnet. In Abbildung 5.6(a) zeigt sich, dass auch die neue Version mit einem zentralen Thread Pool mit einem einzelnen Thread deutlich schneller ist als die bisherige Version. Dadurch, dass der einzelne Thread alle anfallenden Stationen immer nur sequenziell durchlaufen kann, schwankt folglich der Nachrichtendurchsatz sehr stark. Durch das Erhöhen der Thread Anzahl können die Schwankungen weiter ausgeglichen werden, sodass auch der durchschnittliche Nachrichtendurchsatz leicht erhöht wird. Jedoch bildet hier der in Abbildung 4.7 gezeigte Durchsatz der internen Liste eine obere Schranke, sodass drei Threads keinen nennenswerten Vorteil gegenüber zwei Threads bieten. Wie auch bei der Liste sinkt der Durchsatz folglich auch hier beim Einsatz von vier Threads. Dabei bleibt das Ergebnis sogar noch hinter den theoretischen Werten aus Unterabschnitt 4.2.5 zurück. Das liegt daran, dass die Threads nicht ununterbrochen durchlaufen können, sondern an einigen Passagen synchronisiert werden müssen.

Es zeigt sich, dass es zwar möglich ist, die Netzwerkkomponente mit einem einzelnen Thread Pool zu betreiben, jedoch bleibt diese hinter ihren Möglichkeiten. Der Einsatz eines zentralen Thread Pools ist jedoch aufgrund einer gleichmäßigeren Aufgabenverteilung theoretisch besser. Praktisch lässt sich dieser Ansatz nicht verfolgen, da die zur Verfügung stehende Liste nicht mit dem, durch sehr viele sehr kleine Nachrichten, verursachten Arbeitsaufwand mithalten kann. Die Aufteilung in mehrere Thread Pools bietet




(a) 1 Thread

(b) 2 Threads

(c) 3 Threads

(d) 4 Threads

Abbildung 5.6: Zentraler Thread Pool für 64 Byte Payload und 4 Clients

somit eine deutlich bessere Gesamtleistung.

### 5.2.5 NIO Channel Write

Wie bereits in Abschnitt 4.5 erwähnt, gibt es deutliche Leistungsunterschiede bei den verschiedenen Funktionsaufrufen zum Schreiben in einen Channel. In Abbildung 5.7 ist eine Testreihe zu sehen, welche ebenfalls dem bisherigen Testmuster entspricht. Getestet wurde mit zwei Knoten, die Nachrichten mit je 32 Byte Nutzdaten übertragen. Der einzige Unterschied zwischen den Testreihen stellt die Verwendung der `write()` Methode dar. In dem Testlauf betitelt mit “Single Message” wird die Methode, wie bisher, mit einem einzelnen Buffer als Parameter aufgerufen. In den anderen Testreihen wird ein Array

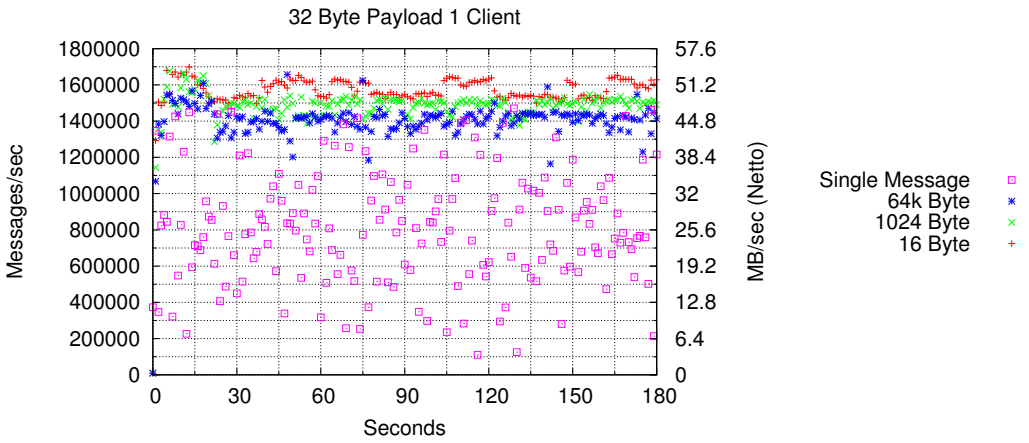



Abbildung 5.7: Vergleich der *write()* Methoden

aus Buffer Objekten übergeben, welches mindestens die angegebene Gesamtkapazität umfasst. Hier lässt sich beobachten, dass der Aufruf mit einem einzelnen Buffer starke Schwankungen aufweist. Der Aufruf mit einem Array ist in allen getesteten Varianten sowohl schneller, als auch stabiler. Dabei scheint es keinen weiteren Leistungsschub zu geben, wenn mehr Daten zeitgleich übergeben werden. Die Messungen zeigen, dass das Limit bereits bei einem Array bestehend aus einem einzigen Buffer erreicht wird. Dass die Vergleichsmessungen mit einem größeren Array langsamer sind, erklärt sich dadurch, dass durch die Zusammenstellung des Arrays mehr Zeit in Anspruch genommen wird. Dieser zusätzliche Aufwand bringt jedoch offensichtlich keinen Mehrwert. Es zeigt sich also, dass auch bei einer einzelnen Nachricht die Nutzung eines Arrays deutlich schneller, und somit empfehlenswert ist.

# Kapitel 6

# Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

In dieser Masterarbeit wurde das Netzwerkmodul von DXRAM umgebaut, um die Performanz durch die effiziente Nutzung mehrerer Threads erheblich zu steigern. Dabei wurde zunächst untersucht, welche Schwachstellen die bisherige Implementierung aufweist. Anschließend wurden die Änderungen erläutert, um die Netzwerkkomponente zu verbessern. Im Rahmen dieser Arbeit wurde insbesondere die Empfängerseite maßgeblich umgestaltet. Die Arbeitsschritte wurden sauber unterteilt und können nun mithilfe eines Thread Pools und einer beliebigen Anzahl von Threads parallel bearbeitet werden. Durch die Arbeitsweise von DXRAM ist hauptsächlich mit vielen kleinen Nachrichten zu rechnen. Entsprechend viele Arbeitsschritte fallen an. Wie Messungen gezeigt haben bietet die verwendete Liste jedoch nicht den nötigen Durchsatz um diese große Anzahl an Aufgaben im Thread Pool verwalten zu können. Aus diesem Grund wurde das Netzwerkmodul in drei Abschnitte unterteilt, in welchem je ein eigener Thread Pool arbeitet. Nur so kann die große Menge an Nachrichten verarbeitet werden.

Die abschließenden Messungen haben im Vergleich zur alten Version die Wirksamkeit der vorgestellten Änderungen gezeigt. Der Datendurchsatz konnte sich auch in schwierigen Situationen deutlich steigern lassen. Für Nachrichten mit 16 Byte Nutzdaten konnte der Durchsatz um den Faktor 16 gesteigert werden. Bei 64 Byte Nutzdaten liegt eine Stei-

gerung um Faktor 10 vor. Ebenso wurde die Bearbeitungsdauer einer Nachricht deutlich gesenkt, sodass Nachrichten jetzt doppelt so schnell beantwortet werden können. Die eigens implementierte Flusskontrolle verhindert nun, dass der Sender zu viele Nachrichten schickt, wenn der Empfänger stark ausgelastet ist. Dadurch staut sich der Datenverkehr nicht länger auf und der Durchsatz bleibt auch über einen langen Auslastungszeitraum hinweg stabil. Die Messungen haben jedoch auch gezeigt, dass die Verarbeitung einer hohen Anzahl von sehr kleinen Nachrichten den Prozessor sehr stark auslastet. So kann es passieren, dass die verfügbare Rechenkapazität zu einem limitierenden Faktor wird und den Datendurchsatz begrenzt. Diese Situation kann jedoch dadurch entschärft werden, dass Datenanfragen gebündelt werden. Werden zwei Datensätze benötigt und in einer einzelnen Anfrage angefordert, halbiert sich auch automatisch die Beanspruchung der Netzwerkkomponente.

Insgesamt ist das Netzwerkmodul nun in der Lage verfügbare Ressourcen auszuschöpfen. Theoretisch kann jetzt eine beliebige Anzahl an Threads sinnvoll auf anfallende Aufgaben verteilt werden. Damit lässt sich auch der Maximaldurchsatz von Gigabit Ethernet erreichen.

## 6.2 Ausblick

Obwohl in dieser Arbeit die Leistung der Netzwerkkomponente an vielen Punkten deutlich verbessert werden konnte, gibt es grundsätzlich immer noch Potenzial für eine weitere Leistungssteigerung. Wie bereits in Unterabschnitt 4.1.1 beschrieben, wäre ein möglicher erster Schritt zur weiteren Optimierung, den Nachrichtenheader weiter zu verkürzen. Aufgrund der üblicherweise sehr kleinen Nachrichten könnte durch einen kleineren Header die verfügbare Bandbreite deutlich effizienter genutzt werden.

Für eine effizientere Nutzung der verfügbaren Threads ist ein einheitlicher Thread Pool notwendig. Durch die notwendige Unterteilung in drei separate Thread Pools kann es immer geschehen, dass in einem Thread Pool Aufgaben vorliegen, während in einem anderen Thread Pool Threads nicht ausreichend beschäftigt werden können. Hier ist zu evaluieren, ob sich durch eine alternative Implementierung einer synchronisierten Liste

die Durchsatzrate deutlich steigern lässt. Als weitere Option wäre es möglich einen komplett eigenen Thread Pool umzusetzen, welcher nicht auf die `TaskExecutor` Klasse von Java zurückgreift. Dort könnte beispielsweise direkt mit mehreren Listen gearbeitet werden. Ebenfalls ist es denkbar, dass mehrere Thread Pools kooperativ arbeiten und freie Threads den anderen Thread Pools zur Verfügung stellen.

Ein dritter Ansatzpunkt wäre Java NIO selbst. In dieser Arbeit konnte, wie auch in anderen Arbeiten, festgestellt werden, dass Java NIO, im Falle einer einzelnen Verbindung, nicht die vollen Möglichkeiten von Gigabit Ethernet ausschöpfen kann. Um diese Beschränkung zu umgehen kann man prüfen, ob mehrere parallele Verbindungen zwischen einem Knotenpaar bessere Ergebnisse erzielen. Ebenfalls kann man evaluieren, ob eine alternative, native Netzwerkbibliothek besser für DXRAM geeignet wäre. Eine Möglichkeit wären die auch in F-MPJ verwendeten Java Fast Sockets. Gegebenenfalls lässt sich auch mit einer eigenen Implementierung eine Leistungssteigerung erzielen.

# Anhang A

# LinkedBlockingQueue Test

Quellcode zum Testen des Datendurchsatzes in einer *LinkedBlockingQueue*

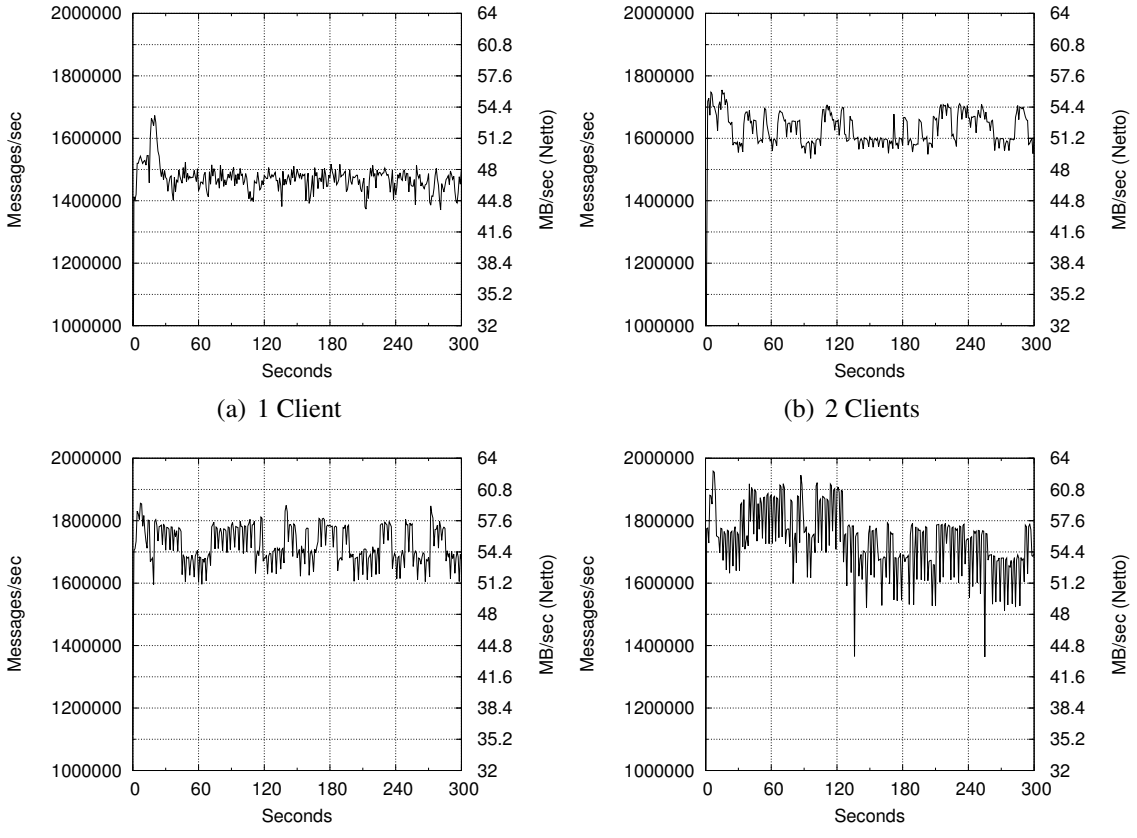
```
1  import java.util.concurrent.*;
2
3  public class Test {
4      static int READER = 2;
5      static int WRITER = 1;
6      static ExecutorService e = Executors.newFixedThreadPool(READER +
           WRITER);
7      static int N = 6000000;
8
9      public static void main(String[] args) throws Exception {
10         for (int i = 0; i < 61; i++) {
11             int length = (i == 0) ? 1 : i * 5;
12             System.out.print(length + "\t");
13             System.out.print(doTest(new LinkedBlockingQueue<Integer>(
                   length), N) + "\t");
14             System.out.println();
15         }
16
17         e.shutdown();
18     }
19
20     private static long doTest(final BlockingQueue<Integer> q, final
           int n) throws Exception {
21         long t = System.nanoTime();
```

```
22
23          for(int i = 0; i < WRITER; i++) {
24              e.submit(new Runnable() {
25                  public void run() {
26                      for (int i = 0; i < n/WRITER; i++) {
27                          try { q.put(i); } catch (InterruptedException
                                ex) {}
28                      }
29                  }
30              });
31          }
32
33          Future<Long>[] futures = new Future[READER];
34          for (int i = 0; i < futures.length; i++) {
35              futures[i] = e.submit(new Callable<Long>() {
36                  public Long call() {
37                      long sum = 0;
38                      for (int i = 0; i < n/(READER); i++) {
39                          try {
40                              sum += q.take();
41                          } catch (InterruptedException ex) {}
42                      }
43                      return sum;
44                  }
45              });
46          }
47
48          for (int i = 0; i < futures.length; i++) {
49              Long r = futures[i].get();
50          }
51
52          t = System.nanoTime() - t;
53
54          return (long)(1000000000.0 * n / t); // Throughput, items/sec
55      }
56  }
```

Listing A.1: BlockingQueueTest.java

# Anhang B

# Messungen zum Datendurchsatz



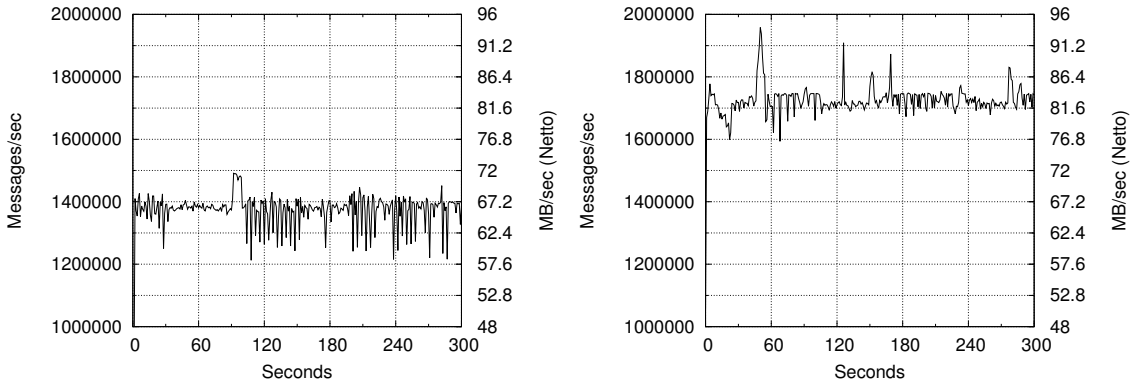

(a) 1 Client

(b) 2 Clients

(c) 4 Clients

(d) 8 Clients

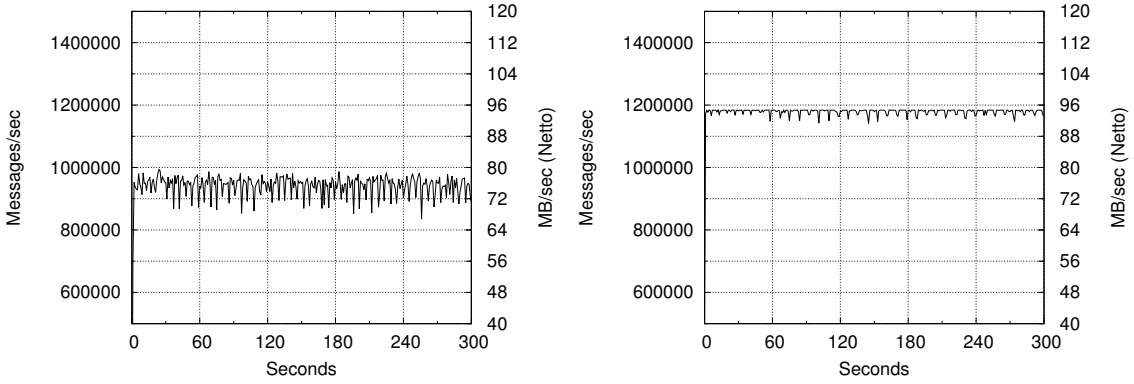
Abbildung B.1: Netzwerkdurchsatz bei 32 Byte Payload




(a) 1 Client (b) 2 Clients

(c) 4 Clients (d) 8 Clients

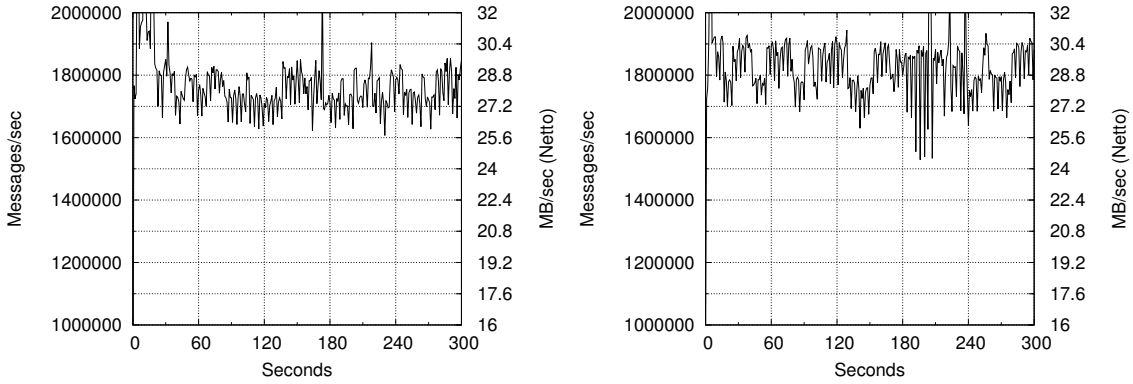
Abbildung B.2: Netzwerkdurchsatz bei 48 Byte Payload




(a) 1 Client (b) 2 Clients

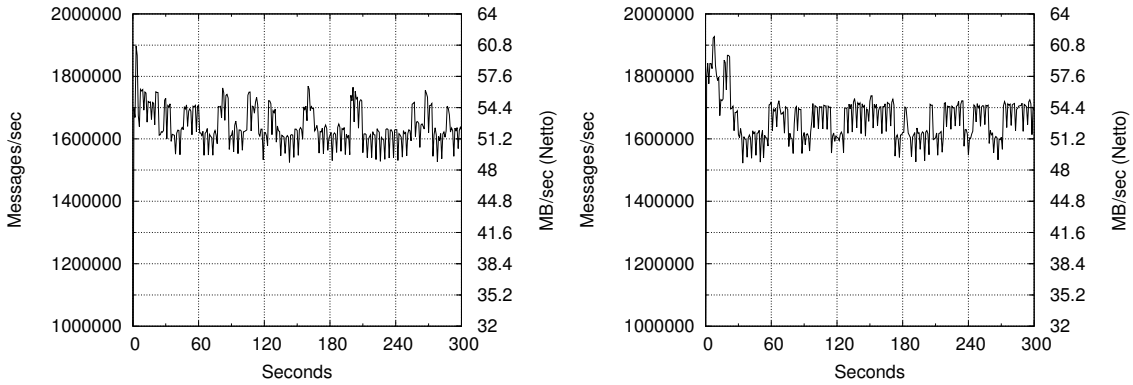
(c) 4 Clients (d) 8 Clients

Abbildung B.3: Netzwerkdurchsatz bei 80 Byte Payload



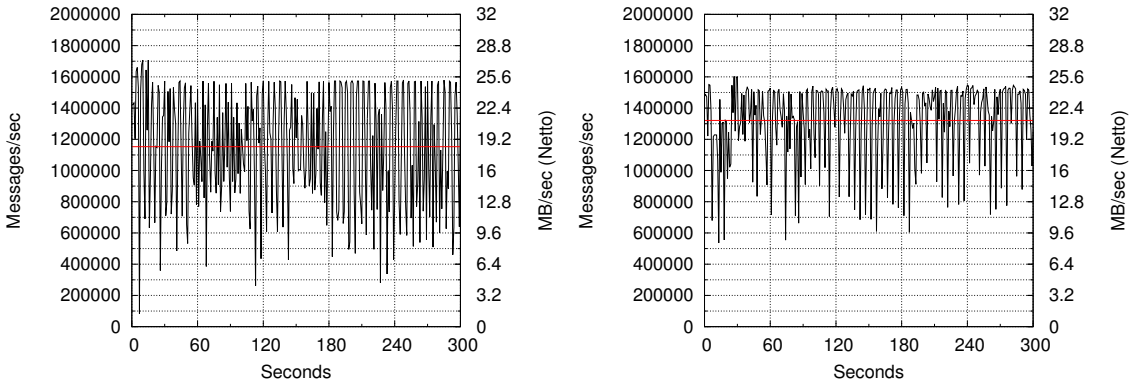

(a) Messung A (b) Messung B

Abbildung B.4: Alternative Testreihen mit 16 Byte Payload und 4 Clients




(a) Messung A (b) Messung B

Abbildung B.5: Alternative Testreihen mit 32 Byte Payload und 4 Clients




(a) 1 Thread

(b) 2 Threads

(c) 3 Threads

(d) 4 Threads

Abbildung B.6: Zentraler Thread Pool für 16 Byte Payload und 4 Clients

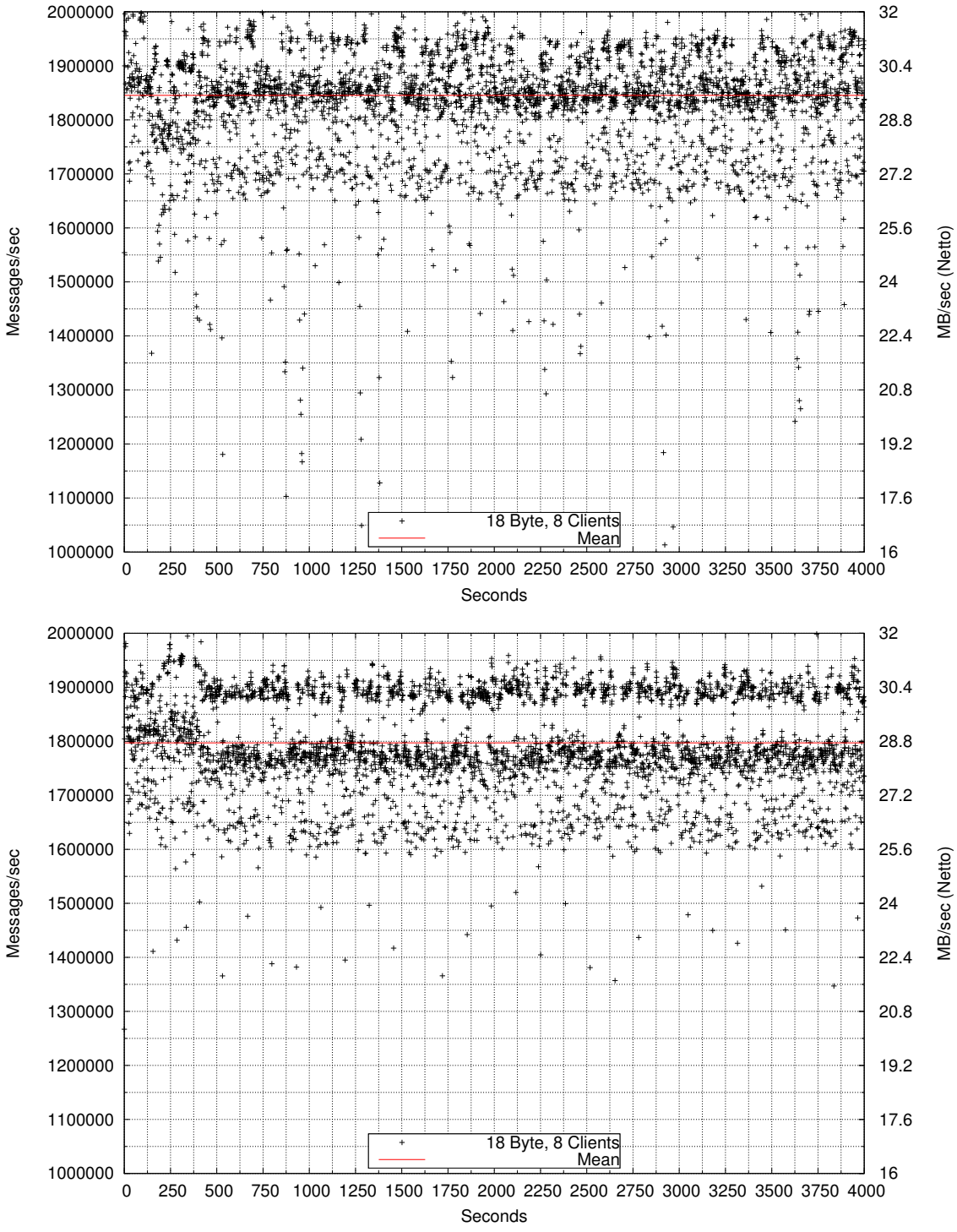



Abbildung B.7: Langzeitmessungen bei 16 Byte und 8 Clients

# Literaturverzeichnis


[BCS06] BAKER, Mark; CARPENTER, Bryan; SHAFT, A: MPJ Express: towards thread safe Java HPC. In: *Cluster Computing, 2006 IEEE International Conference on* IEEE, 2006, S. 1–10.

[BOS04] BAKER, Mark; ONG, Hong; SHAFI, Aamir: A Study of Java Networking Performance on a Linux Cluster. In: *Distributed Systems Group, University of Portsmouth, UK* (2004).

[ES13] EILEBRECHT, Karl; STARKE, Gernot: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 4. Springer Vieweg, 2013

[Jen13] JENKOV, Jakob: Java NIO. In: `http://tutorials.jenkov.com/java-nio/index.html` (2013).

[KBS14] KLEIN, Florian; BEINEKE, Kevin; SCHÖTTNER, Michael: Memory Management for Billions of Small Objects in a Distributed In-Memory Storage. In: *IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, S. 113–122.

[Kri04] KRIEMANN, Ronald: Implementation and Usage of a Thread Pool based on POSIX Threads. In: *MPI MIS Leipzig* (2004).

[KS13] KLEIN, Florian; SCHÖTTNER, Michael: DXRAM: A Persistent In-Memory Storage for Billions of Small Objects. In: *International Conference on Parallel and Distributed Computing, Applications and Technologies*. Taipei, Taiwan, 2013.

[KWM+08] KOTZMANN, Thomas; WIMMER, Christian; MÖSSENBÖCK, Hanspeter; RODRIGUEZ, Thomas; RUSSELL, Kenneth; COX, David: Design of the Java HotSpot™ client compiler for Java 6. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5 (2008), Nr. 1.

[Lea03] LEA, Doug: *Scalable IO in Java*. `http://practice-cpp.googlecode.com/svn-history/r21/trunk/notebooks/nio.pdf`, 2003.

[LK05] LANGER, Angelika; KREFT, Klaus: *Java Performance: Micro-Benchmarking: Wie wirkt sich die HotSpot-Technologie aufs Micro-Benchmarking aus?* `http://www.angelikalanger.com/Articles/EffectiveJava/22.JITCompilation/22.JITCompilation.html`, 2005. [Online; zugegriffen 15. Januar 2015]

[Ora14] ORACLE: *Java Platform, Standard Edition 7 API Specification*. `http://docs.oracle.com/javase/7/docs/api/`, 2014. [Online; zugegriffen 22. Dezember 2014]

[Sch95a] SCHMIDT, Douglas C.: Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In: COPLIEN, Jim (Hrsg.); SCHMIDT, Douglas C. (Hrsg.): *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[Sch95b] SCHMIDT, Douglas C.: Using design patterns to develop reusable object-oriented communication software. In: *Communications of the ACM* 38 (1995), Nr. 10, S. 65–74.

[TG03] TIERNEY, Brian; GUNTLER, Dan: NetLogger: A toolkit for distributed system performance tuning and debugging. In: *Integrated Network Management, IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)* Bd. 246, IEEE, März 2003, S. 97–100.

[TOR13] TUDOSE, Cătălin; ODUBĂŞTEANU, Carmen; RADU, Serban: Java Reflec-

tion Performance Analysis Using Different Java Development. In: *Advances in Intelligent Control Systems and Computer Science* Bd. 187, Springer Berlin Heidelberg, 2013, S. 439–452.

[Tra03] TRAVIS, Grev: *Getting started with NIO*. `http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html`, Juli 2003. [Online; zugegriffen 22. Januar 2015]

[TTD06] TABOADA, Guillermo L.; TOURIÑO, Juan; DOALLO, Ramón: Efficient Java Communication Protocols on High-speed Cluster Interconnects. In: *31st IEEE Conference on Local Computer Networks*. Tampa, FL : IEEE, November 2006, S. 264–271.

[TTD08] TABOADA, Guillermo L.; TOURIÑO, Juan; DOALLO, Ramón: Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. In: *Computer Communications* 31 (2008), Nr. 17, S. 4049–4059.

[TTD12] TABOADA, Guillermo L.; TOURIÑO, Juan; DOALLO, Ramón: F-MPJ: scalable Java message-passing communications on parallel systems. In: *The Journal of Supercomputing* 60 (2012), Nr. 1, S. 117–140.

[WQ12] WANG, Yang; QIN, Guofeng: A Multicore Load Balancing Model Based on Java NIO. In: *TELKOMNIKA Indonesian Journal of Electrical Engineering* 10 (2012), Nr. 6, S. 1490–1495.

[ZHH+07] ZHANG, Hongwei; HUANG, Wan; HAN, Jizhong; HE, Jin; ZHANG, Lisheng: A performance study of Java communication stacks over InfiniBand and giga-bit Ethernet. In: *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on* IEEE, 2007, S. 602–607.

# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 9.Februar 2015 Marc Ewert


**Diese CD enthält:**

- Eine *pdf* Version der Masterarbeit
- Alle LaTeXund Grafik Dateien sowie entsprechende Skripte, die verwendet wurden
- Der Quelltext der Software, welcher dieser Masterarbeit als Grundlage diente
- Der Quelltext der Software, welcher während der Masterarbeit entstanden ist
- Messdaten, die während der Auswertung angefallen sind
- Referenzierte Webseiten und wissenschaftliche Publikationen, sofern digital vorliegend