Karlsruher Institut für Technologie

Portierung und Analyse einer verteilten Datenbank auf einen Hochleistungscluster

Bachelorarbeit
von

cand. inform. Stephan Leonhard

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.-Inform. Jens Kehne
	Dipl.-Inform. Marius Hillenbrand

Bearbeitungszeit: 05. Dezember 2012 – 04. April 2013

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 16. Juni 2011

Zusammenfassung

Während der letzten Jahre gewannen verteilte Dienste zunehmend an Bedeutung, da normale Serversysteme den steigenden Teilnehmerzahlen und Datenmengen nicht mehr gewachsen sind. Anfangs wurden einzelne Server dupliziert und per Anycast angesprochen, mittlwerweile stehen ganze Rechenzentren für einzelne Dienste zur Verfügung. Werden allerdings dieselben Protokolle für die interne Knotenkommunikation auf kurze Distanzen eingesetzt, die auch zwischen gewöhnlichen Netzen im Einsatz sind, ist nur ein Teil der möglichen Leistung nutzbar. Durch passende Hardware und Software lassen sich Latenz, Durchsatz und Prozessorbelastung jedoch optimieren.

Das Anwendungsgebiet der verteilten Datenbanken ist im Web 2.0 besonders durch NoSQL Datenbanken interessant geworden, da sie eine natürlichere Modellierung der Daten erlauben als mit relationalen Datenbanken. Mit Google und Amazon als Vorbild, gibt es bereits zahlreiche dokumentenorientierte Open-Source Datenbanken. MongoDB ist dabei eine der verbreitetsten und bekannt für ihre Schnelligkeit und Skalierbarkeit. LibRIPC ist dagegen eine nachrichtenbasierte Kommunikationsbibliothek, die einen einfachen Zugriff auf die RDMA-Funktionen von Infiniband ermöglicht, einer Hochgeschwindigkeitsübertragungstechnologie. Wir glauben, dass sich diese beiden Technologien nutzbar miteinander vereinen lassen, um eine verteilte Datenbank effizient auf einem Clustersystem betreiben zu können.

In der vorliegenden Arbeit beschreiben wir daher die Portierung von MongoDB nach libRIPC, wodurch es ermöglicht wird, MongoDB in einem Infiniband-Netzwerk zu betreiben, ohne auf die Nachteile von IPoIB angewiesen zu sein. Wir haben unsere Portierung hinsichtlich Latenz, Durchsatz und Prozessorbelastung sowohl im Labor, als auch unter realen Bedingungen getestet. Dadurch können wir aufzeigen, in welchen Fällen die Vorteile durch libRIPC besonders ausgeprägt sind.

Inhaltsverzeichnis

Zusammenfassung **v**

Inhaltsverzeichnis **1**

1 Einführung **3**

2 Verwandte Arbeiten **7**
2.1 Infiniband 7
2.2 SDP 7
2.3 RDMA-Modifikationen 8
2.4 libRIPC 8

3 Design **9**
3.1 Architektur 10
3.2 Kommunikation 11
3.3 Speicherverwaltung 13
3.4 Paketverlust 14
3.5 Schlussfolgerung 14

4 Implementierung **17**
4.1 Datenstrukturen 17
4.2 Paketverlust 18
4.3 Erfahrungen 20

5 Evaluation **21**
5.1 Zielsetzung 21
5.2 Portierungsaufwand 22
5.3 Testumgebung 25
5.4 Versuchsaufbau 25
5.5 Ergebnisse 26

6 Fazit **35**

Bibliographie **37**

Kapitel 1

Einführung

Netzwerkdienste werden in den letzten Jahren bezüglich wachsenden Datenaufkommens und wachsender Teilnehmerzahlen zunehmend gefordert. Um den daraus folgenden Anforderungen an Serversysteme gerecht zu werden, wird häufig auf eine Clusterlösung zurückgegriffen, da durch die Zusammenschaltung mehrerer Hochleistungsrechner sowohl eine erhöhte Rechenkapazität, als auch eine erhöhte Verfügbarkeit gewährleistet werden kann [24]. Ein Problem hierbei ist die Kommunikation zwischen den einzelnen Knoten eines Clusters, da dafür Protokolle und Schnittstellen eingesetzt werden, die nicht für diesen Zweck entwickelt oder optimiert wurden und in diesem Umfeld einige Nachteile aufweisen. Im Folgenden betrachten wir einige Probleme, die durch den Einsatz herkömmlicher Netzwerkprotokolle wie Transmission Control Protocol/Internet Protocol (TCP/IP) entstehen.

Wie der Name Internet Protocol bereits suggeriert, wurde das Protokoll dazu entworfen, um Daten zwischen Computern zu übertragen, die sich nicht zwangsläufig im selben Netz befinden. Auch das Transmission Control Protocol löst viele Probleme wie Fluss- oder Staukontrolle, die in Zwischennetzkommunikationen wesentlich häufiger auftreten, als in einer Clusterknotenkommunikation [23]. Das hat zur Folge, dass einige Merkmale von TCP/IP zwar zur Verfügung stehen, aber nicht zum Einsatz kommen. Die entsprechenden Parameter müssen berechnet und versendet werden, was einen erkennbaren Mehraufwand mit sich bringt. Konkret bedeutet das, sowohl eine Einschränkung der Rechenkapazität, als auch der verfügbaren Bandbreite und der Latenz.

Um dem entgegenzuwirken, wurden neue Netzwerkprotokolle entwickelt, die an die speziellen Anforderungen der Clusterknoten angepasst sind und die sich in zwei Gruppen gliedern lassen. Einerseits werden verbreitete APIs für die neue Technologie adaptiert [22] [6] und andererseits werden neue APIs direkt für die

Anwendungen entwickelt [12] [20]. Ersteres vereinfacht die Integration in bestehende Systeme, da jede Anwendung, die ohnehin für diese API entwickelt wurde, die neue Technologie ohne Mehraufwand nutzen kann. Letztere sind zwar mit einem erhöhten Arbeitsaufwand verbunden, da Teile der Server- und Clientprogramme unter Umständen neu geschrieben werden müssen, können dafür aber den vollen Leistungsumfang der Hardware nutzen, da sie nicht zwangsläufig zu einer bestehenden API kompatibel sein müssen. Ein derartiges Protokoll wurde Beispielsweise in *libRIPC* implementiert, welches auf der Hochgeschwindigkeitsübertragungstechnologie *Infiniband* aufsetzt. Für diese Bibliothek wurden bereits einige Anwendungen wie zum Beispiel Webserver portiert, was in dieser Arbeit um eine verteilte Datenbank erweitert wird [12] [10].

Die dokumentenorientierte Datenbank *MongoDB* [2] eignet sich dafür besonders gut, da sie durch ihre hochperformante Architektur und ihre Skalierbarkeit für den Einsatz auf Computerclustern geradezu prädestiniert ist. Derzeit wird diese Datenbank aufgrund der Flexibilität, der Fähigkeit mit großen Datenmengen umzugehen oder der Skalierbarkeit unter anderem bei CERN CMS, theguardian, Disney, craigslist, Cisco und The national Archives erfolgreich eingesetzt [3]. Da die Datenbank unter der Apache Lizenz steht und somit Quelloffen ist, sind Modifikationen ohne größeren Aufwand möglich. Daher soll MongoDB nun auf libRIPC portiert werden, um die Leistungsvorteile von Infiniband nutzen zu können. Die Änderungen sollen dabei jedoch derart minimal sein, dass die neue Netzwerkschicht rein optional zum Übersetzungszeitpunkt auswählbar ist, MongoDB ansonsten aber unverändert bleibt.

MongoDB ist eine NoSQL-Datenbank [13], das bedeutet sie verzichtet auf ein festes Tabellenschema, wie es von relationalen Datenbanken benutzt wird, und baut stattdessen auf einem dokumentenorientierten Ansatz auf. Dadurch ist es möglich, auch komplexere Datenstrukturen als einfache Tabellen in die Datenbank aufzunehmen, was den darauf aufbauenden Anwendungen eine natürlichere Modellierung erlaubt.

Ein Nachteil von relationalen Datenbanken sind dagegen Leistungseinbußen bei einer hohen Anzahl von Schreibzugriffen [7], welche in realen Clusterumgebungen jedoch nicht vermeidbar sind. NoSQL Datenbanken sind dafür optimiert, große Datenaufkommen effizient verarbeiten zu können. Dafür ist der Funktionsumfang für das Verknüpfen von Datensätzen in der Regel eingeschränkt. Bekannte Beispiele für NoSQL-Datenbanken sind BigTable von Google und Dynamo von Amazon [8] [15].

Die Datenverteilung auf mehrere Server ist bei MongoDB durch *horizonta-*

le Fragmentierung [5] realisiert, das in MongoDB auch als *sharding* bezeichnet wird. Für diesen Zweck existiert die Anwendung *mongos*, die mit Hilfe eines *Fragmentierungsschlüssel* kontrolliert, welche Daten auf welchen Servern abgelegt werden. Diese Anwendung dient dem Anwender als Anlaufstelle, koordiniert Anfragen und schickt die angeforderten Daten wieder an den Anwender zurück.

Kapitel 2

Verwandte Arbeiten

Im Folgenden werden einige alternative Ansätze zu TCP/IP basierten Netzwerken vorgestellt, die für die Verbindung zwischen Clusterknoten besser geeignet sind.

2.1 Infiniband

Infiniband ist eine Industriestandardspezifikation zur Hochgeschwindigkeitsübertragung zwischen Endgeräten [10] und wird sowohl als Bussystem, als auch als Netzwerktechnologie zur Verbindung von Clusterknoten eingesetzt. Der Vorteil von Infiniband gegenüber Ethernet liegt in der geringeren Latenz und der Auslagerung des Protokollstacks auf die Hardware [19].

2.1.1 libibverbs

Libibverbs ist eine Bibliothek, die den direkten Zugriff auf Infinibandgeräte ermöglicht [17]. Die Benutzung ist allerdings sehr aufwändig, da praktisch nicht von der Hardware abstrahiert wird. Die resultierenden Programme sind dadurch sehr umfangreich, schlecht wartbar und nicht portabel.

2.2 SDP

Das Socket Direct Protocol (SDP) ist ein Netzwerkprotokoll, das auf Infiniband aufsetzt und eine Remote Direct Memory Access (RDMA) beschleunigte Alternative für TCP darstellt. Durch die RDMA-Technik, können Daten über das Netzwerk transportiert werden, ohne sie sowohl beim Client, als auch beim Server innerhalb des Betriebsystems kopieren zu müssen. Ein weiteres Ziel das SDP verfolgt, ist Transparenz gegenüber den Anwendungen, das durch Kompatibilität zur

Berkeley Socket Schnittstelle erreicht wird, Dadurch gehen allerdings wieder viele Vorteile von Infiniband verloren, was sich deutlich in der Leistung bemerkbar macht.

2.3 RDMA-Modifikationen

HBase ist eine verteilte Open-Source Key/Value Datenbank in Java, die sich an der Idee von BigTable [8] orientiert. Durch eine RDMA-Modifikation [9] wurde diese Datenbank um den Faktor 3.5 beschleunigt. Verglichen wurde dabei mit einer 10 Gigabit Ethernet Netzwerkverbindung.

Memcached [11] ist ebenfalls eine verteilte Key/Value Datenbank, die auf BSD Sockets basiert. Diese wurde durch RDMA um den Faktor 4 beschleunigt. Verglichen wurde wieder mit einer 10 Gigabit Ethernet Netzwerkverbindung.

2.4 libRIPC

LipRipc ist eine leichtgewichtige Kommunikationsbibliothek, die von libibverbs abstrahiert und damit eine portable Schnittstelle für Infiniband bereitstellt [12]. In libRIPC existieren angelehnt an L4 [14] zwei Nachrichtentypen mit unterschiedlichen Eigenschaften. Die Shortmessage ist für kleinere Nachrichten konzipiert und kopiert dabei die Daten vom Sender zum Empfänger. Die Longmessage dagegen, nutzt RDMA, um vom Sender direkt in den Speicher des Empfängers zu schreiben. Durch die aufwändigere Vorbereitungen lohnt sich eine Longmessage erst ab einer gewissen Paketgröße, um einen Geschwindigkeitsvorteil zu erlangen. Messungen ergaben verglichen mit Socketlösungen zudem eine geringere Prozessorbelastung. Im Vergleich zu SDP, wird keine Transparenz gegenüber den Anwendungen angestrebt, sodass die Vorteile von Infiniband effizienter genutzt werden können. Beispielsweise werden Pakete nicht fragmentiert, wie das in SDP der Fall ist. Dadurch können mittels RDMA-Zugriff große Datenmengen als einzelne Nachricht übertragen werden. Durch die fehlende Transparenz müssen die Anwendungen allerdings speziell angepasst werden. Die vorliegende Arbeit befasst sich daher mit Portierung von MongoDB nach libRIPC.

Kapitel 3

Design

Dieses Kapitel befasst sich mit den Designentscheidungen, die im Kapitel 4 Implementierung umgesetzt werden. Dabei gehen wir im ersten Abschnitt auf die Softwarearchitektur von MongoDB ein, um einordnen zu können, welche Teile der Software verändert werden müssen. Danach betrachten wir die zentralen Datenstrukturen die in MongoDB verwendet werden und wie Nachrichten zwischen einem Server- und einem Clientprogramm ausgetauscht und verarbeitet werden. Der vierte Abschnitt behandelt die Verwaltung von dynamisch reserviertem Speicher, da es für die Portierung von besonderer Bedeutung ist und im fünften Abschnitt gehen wir auf Paketverluste ein, da dieses Problem einige Designentscheidungen erfordert, die vor der Implementierung zu treffen sind. Die Informationen für dieses Kapitel wurden aus den Quelltexten im Verzeichnis src/mongo/util/-net erarbeitet, da die existierende Dokumentation der Datenbank, primär an die Anwender, nicht aber an die Entwickler gerichtet ist.

3.1 Architektur

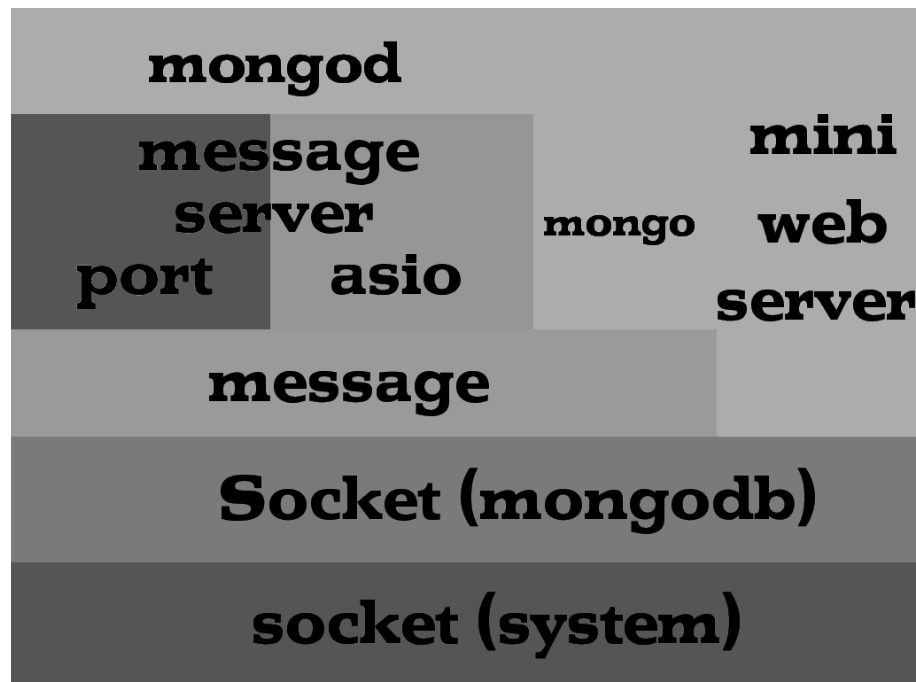

Abbildung 3.1: Schichtmodell des Netzwerksystems von MongoDB.

MongoDB orientiert sich an einem Schichtmodell, welches vom zugrundeliegenden Netzwerk abstrahiert und somit die Implementierung einer alternativen Transportlösung ermöglicht. Um Plattformunabhängigkeit zu ermöglichen, werden verschiedene Netzwerk-APIs genutzt. Unter POSIX-kompatiblen Systemen werden BSD-Sockets und unter Windows Winsockets eingesetzt. MongoDB abstrahiert davon in Form einer eigenen Socket-Klasse im Namensraum mongo, die im Folgenden Mongosocket genannt wird. In MongoDB ist ein sogenannter *miniwebserver* zur Ausgabe von Statusmeldungen und Systeminformationen enthalten, der direkt auf der Mongosocket-Schicht aufsetzt. Für alle datenbankrelevanten Anwendungen ist über der Mongosocket-Schicht die *Message*-Schicht angesiedelt, welche Netzwerkpakete zu einer logischen Nachricht zusammenfasst. Jedes Nachrichtenobjekt hat dabei unter anderem einen Operationskode und einen Datenteil. Alle Clientanwendungen wie Datenbanktreiber oder die *MongoShell* bauen unmittelbar auf dieser Message-Schicht auf.

Serveranwendungen nutzen dagegen eine weitere Schicht, um mehrere Anfragen pro Verbindung bedienen zu können. Im Konzept vorgesehen sind dafür

je nach Anwendungszweck entweder *Message-Server-Port* oder *Message-Server-Asio*. Die Message-Server-Port-Schicht verarbeitet die Anfragen sequentiell, Message-Server-Asio ist dagegen in der Lage, eingehende Anfragen eines Clients asynchron zu verarbeiten. Welche der beiden Lösungen benutzt wird, wird zum Übersetzungszeitpunkt festgelegt.

Für die Portierung nach libRIPC bietet sich die asynchrone Lösung als Ansatz an, da durch die RDMA-Funktionsweise bereits ein asynchrones Verhalten vorliegt. Derzeit ist die Implementierung von Message-Server-Asio in MongoDB jedoch noch nicht vollständig abgeschlossen, daher beschränkt sich diese Arbeit auf die sequenzielle Verarbeitung der Anfragen per Message-Server-Port.

3.2 Kommunikation

Für eine einfache Datenbankanwendung mit MongoDB sind zwei Akteure nötig; Eine Serveranwendung und eine Clientanwendung, die mit Hilfe eines passenden Treibers Daten mit der Serveranwendung austauscht. Im Folgenden wird der Ablauf einer solchen Kommunikation skizziert.

Das Serverprogramm befindet sich nach der Initialisierung in einer Programmschleife, in der ausschliesslich auf eingehende Verbindungen gewartet wird. Sobald sich das Clientprogramm mit dem Server verbindet, wird für diesen Kommunikationspartner ein neuer Thread erstellt, damit die Serveranwendung sofort für Verbindungsanfragen von weiteren Clients bereit steht. Dieser Thread pro Verbindung läuft eigenständig, ist also nicht auf eine Interprozesskommunikation mit anderen Threads angewiesen und besteht im wesentlichen aus einer Schleife, die auf eine Anfrage wartet, diese verarbeitet, daraufhin eine Antwort generiert und an den Client zurückschickt.

Das Clientprogramm besteht aus einer Programmschleife, in der eine Anfrage gesendet und daraufhin auf eine Antwort gewartet wird. Das Warten auf eine Antwort kann für einige Anfragen wie Einfügeoperationen zwar unterbunden werden, allerdings können Fehler dann nicht mehr erkannt werden. Aus diesem Grund verwenden wir stets den sogenannten *safemode*, in dem nach jeder Anfrage auf die Rückmeldung des Servers gewartet wird.

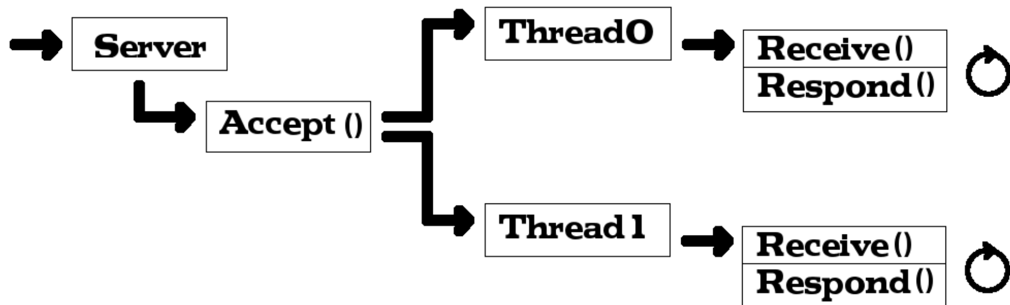

Abbildung 3.2: Verbindungsverwaltung der MongoDB Serveranwendung mit Sockets.

Um diesen Ablauf mit libRIPC realisieren zu können, sind einige strukturelle Änderungen nötig. Denn da der Absender in libRIPC erst nach dem Empfang der Nachricht bekannt ist, können die Nachrichten der entsprechenden Verbindung nicht im Vorfeld zugeordnet werden. Demzufolge kann dieses Konzept nicht übernommen werden und wird daher folgendermaßen abgeändert:

Der Empfang wird nicht mehr im Verbindungsthread abgesetzt, sondern erfolgt global für alle Verbindungen in der Serverhauptschleife. Damit die anderen Verbindungen während der Abarbeitung einer Abfrage aber nicht blockiert werden, wird eine eingehende Nachricht unmittelbar nach Empfang in eine der Verbindung zugehörigen Warteschlange eingereiht, von welcher sie vom entsprechenden Thread bei Bedarf abgeholt wird.

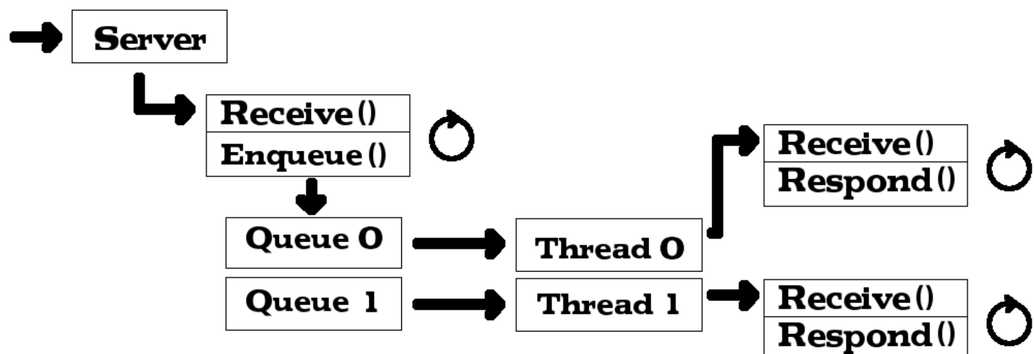

Abbildung 3.3: Verbindungsverwaltung der MongoDB Serveranwendung mit libRIPC.

3.3 Speicherverwaltung

In MongoDB wird beim Empfang einer Nachricht dynamisch neuer Speicher in Abhängigkeit der Nachrichtengröße reserviert, der im Laufe der Verarbeitung wieder freigegeben wird.

Für libRIPC ist diese dynamische Speichernutzung allerdings ungünstig, da im Fall von Longmessages bereits vor dem Empfang ein *Receivewindow* bereitstehen muss und sich libRIPC im Fall von Shortmessages selbst um das Alloziieren kümmert. Da das Reservieren von libRIPC-Puffern aber relativ teuer in Bezug auf die Laufzeit ist, sollten diese nach Möglichkeit auch wiederverwendet werden können. Um dieses Problem zu lösen, wurde ein Speicherpool eingeführt, der libRIPC-Speicher bereits beim Initialisieren reserviert und Funktionen bereitstellt, um diesen wiederverwenden zu können. Dieser Speicherpool wird im Folgenden als RipcBufferPool bezeichnet.

Wir betrachten nun die Mechanik der Speicherreservierung in MongoDB, die nicht unmittelbar ersichtlich ist, da sich eine logische Nachricht über mehrere Netzwerkpakete erstrecken kann. Trotzdem sollen die Nutzdaten der Nachricht in einen zusammenhängenden Speicherblock geschrieben werden, der vor dem Empfang reserviert werden sollte. Da beim Transport per TCP vor dem Empfang nicht feststeht, wie groß die logische Nachricht sein wird, wird die Nachrichtengröße, von MongoDB vor den eigentlichen Nutzdaten versendet. Die Speicheralloziierung für die Nutzdaten findet statt, nachdem die ersten vier Bytes einer Nachricht empfangen und interpretiert wurden. Dadurch gibt es eine minimale Differenz zwischen den Daten die über das Netzwerk versendet werden und den Daten, die in den Speicher geschrieben werden. Das macht es bei der Benutzung von libRIPC schwieriger die atomar empfangenen Daten direkt als Nachrichtendatenstruktur weiterzuverarbeiten. Aus diesem Grund wurde die neue Datenstruktur *RipcBuffer* eingeführt, die kompatibel zur Nachrichtendatenstruktur *MessageData* von MongoDB ist und durch libRIPC-Aufrufe direkt beschrieben werden kann. Diese Struktur stellt ausserdem noch eine transparente Schnittstelle bereit, um von Short- und Longmessages zu abstrahieren und um die Freigabe des Puffers in Abhängigkeit der Verwendung zu realisieren.

Ein Nachteil des vorreservierten Speichers im RipcBufferpool ist die konstante Größe, die beim Initialisieren der Anwendung bekannt sein muss. Im Fall von MongoDB ist das nicht weiter problematisch, da die Anzahl der Ergebniszeilen pro Anfrage auf 20 limitiert ist. Falls mehr übertragen werden sollen, werden diese durch nachfolgende *continue*-Anfragen in weiteren Blöcken zu 20 Zeilen ausgeliefert. Des Weiteren ist die maximale Größe eines MongoDB-Objekts hart

auf 16 MByte limitiert.

3.4 Paketverlust

Neben Longmessages die über RDMA-Zugriffe funktionieren, bietet libRIPC auch Shortmessages an, bei denen die Daten intern kopiert werden müssen. Shortmessages werden aus diesem Grund für kleine Nutzdaten verwendet, für die es sich zeitlich nicht lohnt einen RDMA-Zugriff auszuhandeln. Da Shortmessages in libRIPC gegen Paketverlust aber nicht abgesichert werden, muss sich der Anwendungsentwickler selbst darum kümmern, dies zu erkennen und zu beheben. Es gibt nun verschiedene Situationen, in denen ein Paketverlust auftreten kann, die im Folgenden zusammen mit einer Lösung aufgezeigt werden: Der Client stellt eine Anfrage an den Server und erwartet eine Antwort. Der Server empfängt eine Anfrage, verarbeitet sie und sendet eine Antwort. In diesem Fallbeispiel kann sowohl die Anfrage, als auch die Antwort verlorengehen, sofern die Nutzdaten ausreichend klein sind und daher Shortmessages für den Transport benutzt werden. Da der Client aber nicht ohne weiteres feststellen kann, ob die Anfrage oder die Antwort verlorenging, wiederholt er nach einer Zeitüberschreitung die Anfrage mit derselben MongoDB-Sequenznummer. Der Server erkennt daraufhin, ob er diese Sequenznummer bereits verarbeitet hat, damit Einfüge- oder Schreiboperationen keinesfalls wiederholt werden. Falls aber die Anfrage bereits verlorenging, ist dem Server die Sequenznummer unbekannt und die Anfrage wird normal abgearbeitet. Durch dieses Konzept ist es ausreichend die Paketverluste lediglich clientseitig zu berücksichtigen [24].

Die Wahl der Dauer einer Zeitüberschreitung kann sich allerdings negativ auswirken. Ist der Wert zu hoch, wird das die Latenz im Fall eines Paketverlustes stark erhöhen. Ist der Wert zu gering, werden zeitintensive Anfragen fälschlicherweise als Paketverlust erkannt.

3.5 Schlussfolgerung

In diesem Kapitel wurde der Aufbau und einige Eigenschaften von MongoDB untersucht und basierend darauf wurden die Designentscheidungen für die nun folgende Implementierung getroffen. Für die Portierung selbst erwarten wir keine Schwierigkeiten, allerdings wird die Wahl der Parameter eine Herausforderung. Insbesondere die Werte für die Zeitüberschreitungen, die in Abschnitt 3.4 auf Seite 14 diskutiert wurden, müssen empirisch ermittelt werden. Die Parameter für die Größe der Empfangsfenster und die Anzahl der Poolelemente, die in Abschnitt 3.3

auf Seite 13 besprochen werden, können dagegen aufgrund der Randbedingungen abgeschätzt werden.

Kapitel 4

Implementierung

In diesem Kapitel werden Implementierungsdetails vorgestellt, die entweder besonders wichtig sind oder unerwartet während der Implementierungsphase hinzukamen. Der erste Abschnitt behandelt sowohl die Datenstrukturen aus MongoDB die modifiziert wurden, als auch neu erstellte. Der zweite Abschnitt geht auf die Details zum Speichermanagement aus dem Abschnitt 3.3 auf Seite 13 ein. Zuletzt wird noch die technische Lösung zum Problem der Paketverluste vorgestellt, das wir im Abschnitt 3.4 auf Seite 14 diskutiert haben.

4.1 Datenstrukturen

MongoDB überträgt und speichert Datensätze in Form der Binary Javascript Object Notation (BSON) [1] und erweitert dadurch die Flexibilität der Javascript Object Notation (JSON) um binäre Datentypen. JSON ist ein Format zum Datenaustausch zwischen Anwendungen und erlaubt die Serialisierung von komplexen Datenstrukturen, ähnlich der Extensible Markup Language (XML).

Um ein BSON-Objekt mit einer Operation wie Einfügen oder Ändern zu verknüpfen, wird es zusammen mit einem Operationscode und weiteren Informationen in ein *Message*-Objekt geschrieben, welches dann über das Netzwerk transportiert werden kann. Diese Messageobjekte bestehen aus einem Kopfbereich konstanter Länge und einem Datenteil variabler Länge. Dieser dynamisch allozierte Datenteil macht es erforderlich, dass der damit verknüpfte Speicherbereich explizit wieder freigegeben werden muss, was im Destruktor des Messageobjektes erledigt wird. Da die Nutzdaten der Nachrichten im Rahmen der Portierung direkt in libRIPC-Speicherbereichen gehalten werden, verweist dieser Freigabemechanismus nun auf die libRIPC Freigabefunktionen.

Es hat sich während der Implementierungsphase allerdings herausgestellt, dass Messageobjekte auch für serverinterne Zwecke eingesetzt werden. Die Objekte werden dabei nicht gesondert gekennzeichnet, da weder beim Verarbeiten noch beim Freigeben darauf geachtet werden muss. Mit libRIPC ist diese Unterscheidung jedoch erforderlich, da Speicherbereiche die für den Transport benutzt wurden, nicht an gewöhnliche Freigabefunktionen weitergereicht werden dürfen. Es wäre an dieser Stelle auch möglich gewesen, Messageobjekte grundsätzlich immer mit libRIPC-Speicherbereichen auszustatten. Aber da in diesem Fall netzwerkspezifische Puffer zur serverinternen Kommunikation benutzt worden wären, wurde dieser Ansatz nicht weiter verfolgt.
Des Weiteren muss auch noch unterschieden werden, ob der Nutzdatenspeicher im Messageobjekt aus dem RipcBufferPool stammt, oder ob er intern von libRIPC reserviert wurde. Falls für den Transport eines Messageobjekts Longmessages verwendet wurden, stehen die Daten innerhalb eines wiederverwendbaren Receivewindows bereit. Diese Empfangsfenster werden nach Gebrauch wieder als verwendbar markiert und in den Pool zurückgeführt. LibRIPC ist intern zwar auch in der Lage freigegebenen Speicher wiederzuverwenden, sucht sich dafür jedoch in O(n) innerhalb einer *freelist* einen passenden Bereich heraus. Durch das übergeordnete Wissen um die verwendeten BSON-Datenstrukturen, kann für den RipcBufferPool eine Warteschlange statt einer Liste verwendet werden, die es in konstanter Zeit O(1) ermöglicht, einen freien Bereich auszuwählen. Falls für den Transport Shortmessages verwendet wurden, reserviert libRIPC beim Empfang der Nachricht automatisch Speicher. Dieser wird im Gegensatz zu den Empfangsfenstern unmittelbar mit einem Aufruf von ripc_buf_free() freigegeben, da er an dieser Stelle innerhalb von MongoDB nicht ohne weiteres wiederverwendet werden kann. Die intern von MongoDB verwendeten Puffer werden nach wie vor mit free() freigegeben.

4.2 Paketverlust

In Abschnitt 3.4 auf Seite 14 haben wir eine Lösung für Paketverluste aufgezeigt, die auf Zeitüberschreitungen basiert. Da libRIPC jedoch keine Zeitüberschreitungen oder Unterbrechungen der blockierenden Funktionen vorgesehen hat, obliegt es dem Anwendungsentwickler, ein derartiges Verhalten zu implementieren. Zu diesem Zweck wird im Client ein zusätzlicher Thread gestartet, der die blockierenden libRIPC-Aufrufe überwacht und gegebenenfalls mit einem Signal abbricht und die Fehlerbehandlung einschaltet.

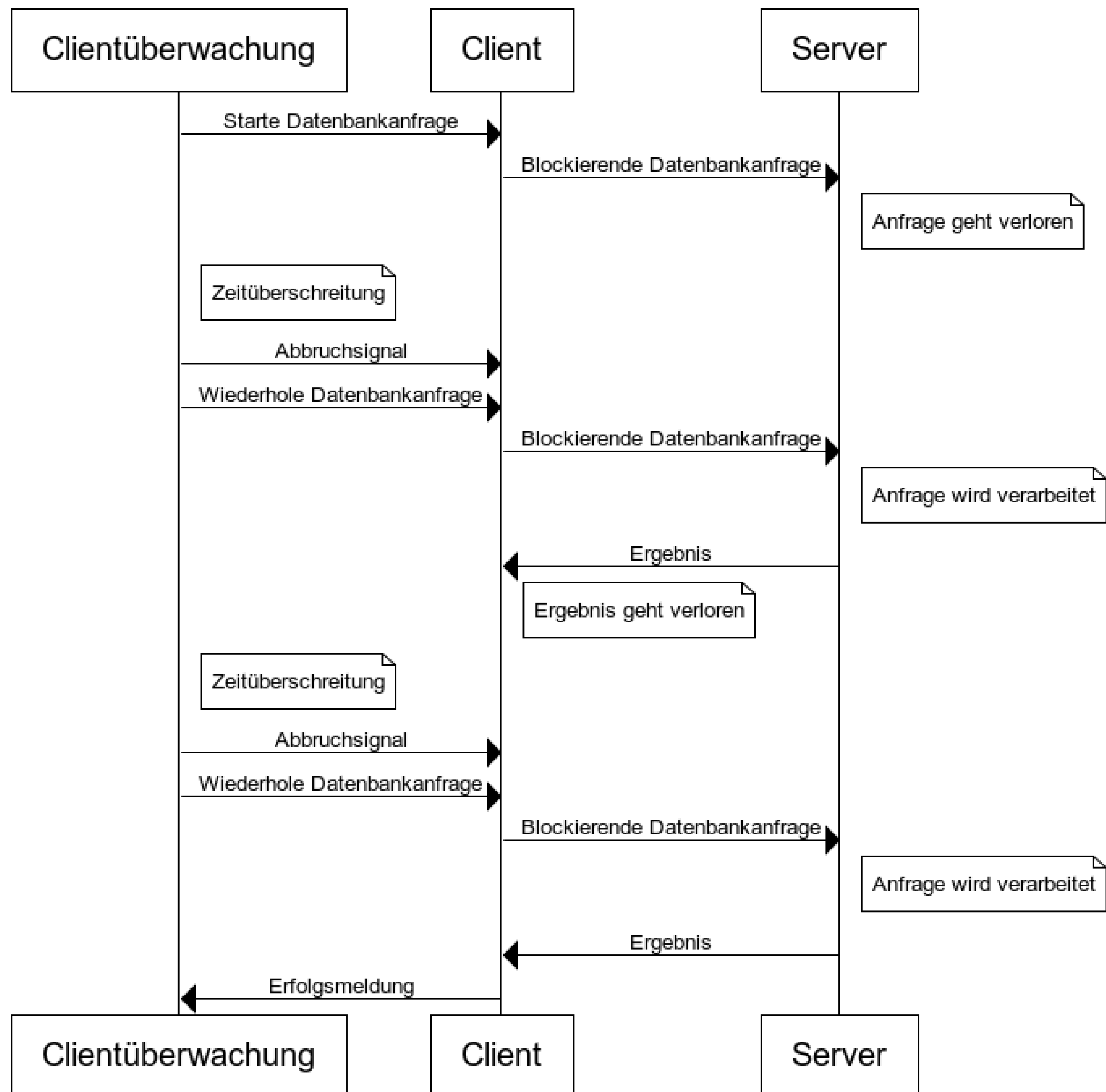

Abbildung 4.1: Clientseitiges Erkennen und Beheben von Paketverlusten.

In Abbildung 4.1 ist das Verhalten des Clients illustriert, falls Paketverluste auftreten. Es ist auch deutlich die Problematik der Zeitüberschreitung aus Abschnitt 3.4 auf Seite 14 erkennbar, nach welcher einerseits die Latenz erhöht wird und andererseits lange andauernde Anfragen abgebrochen werden.

4.3 Erfahrungen

In diesem Kapitel haben wir uns mit einigen Implementierungsdetails wie der Modifikation der verwendeten Datenstrukturen und der Behandlung von Paketverlusten beschäftigt. Wir haben auch gesehen, dass es aufgrund der Speicherverwaltung der Nachrichten nötig war, tiefer in MongoDB einzugreifen als nur die Netzwerkfunktionen auszutauschen. In der nun folgenden Evaluation wird sich zeigen, ob die Designentscheidungen korrekt und die Implementierung hinreichend war.

Kapitel 5

Evaluation

In diesem Kapitel wird das Testen der Implementierung aus Kapitel 4 auf Seite 17 beschrieben. Dazu formulieren wir im ersten Abschnitt zunächst die verfolgten Ziele. Im zweiten Abschnitt gehen wir darauf ein, wie aufwändig die Portierung war und in welcher Weise auf Aktualisierungen reagiert werden kann. Danach betrachten wir die eingesetzte Testumgebung und stellen im Anschluss daran die Messergebnisse vor.

5.1 Zielsetzung

Es soll in erster Linie herausgefunden werden, in wie weit sich Vorteile durch den Einsatz von libRIPC in MongoDB ergeben. Dazu wird der Aufwand der Portierung betrachtet und das Verhalten des Servers unter Laborbedingungen, sowie unter unterschiedlichen Arbeitsbelastungen untersucht.

Der Aufwand der Portierung wird daran gemessen, wie schwierig es war, das bestehende System zu erfassen, um die Ansatzpunkte für die Implementierung zu lokalisieren. Zusätzlich wird noch der Aufwand betrachtet, auf Softwareaktualisierungen seitens MongoDB zu reagieren.

Die Untersuchung unter Laborbedingungen gewährleistet, dass die Portierung erfolgreich war und dass der Einsatz von libRIPC unter bestmöglichen Bedingungen Vorteile bietet. Es soll dabei die Zeit gemessen werden, die zwischen einer Anfrage und der Antwort vergeht (Latenz), sowie die Zeit, die vergeht, bis eine bestimmte Menge Daten übertragen und verarbeitet wurden (Durchsatz).

Die unterschiedlichen Arbeitsbelastungen simulieren dagegen Verhältnisse, wie sie in realen Anwendungen vorkommen. Die Szenarien orientieren sich dabei

an dem Yahoo Cloud Service Benchmark (YCSB) [26]. Durch diese Untersuchung soll herausgefunden werden, ob es auch unter realen Bedingungen vorteilhaft ist, libRIPC einzusetzen.

Um die Analyse abzuschliessen, soll auch die Prozessorbelastung auf dem Server betrachtet werden. Für diese Messungen wird der Prozessmonitor top [25] eingesetzt und im Batchmodus betrieben.

5.2 Portierungsaufwand

Vor der Portierung wurde der Quelltext sowohl manuell, als auch automatisch analysiert. Aufgrund der Komplexität von MongoDB wurde das Werkzeug Understand [21] eingesetzt, um die Abhängigkeiten der Klassen zu untersuchen. Speziell die Socket-Klasse, sowie die weiteren Netzwerkspezifischen Dateien aus dem Ordner src/mongo/util/net waren für die Analyse von Bedeutung, da die Portierung ausschliesslich in der Netzwerkschicht ansetzen sollte. Die Analyse ergab einige unerwartete Abhängigkeiten bezüglich der Systemsockets, da abweichend vom Schichtmodell zum Beispiel in db/commands/isself.cpp auch direkt darauf zugegriffen wurde. Da die Funktionen aufgrund des unterschiedlichen Designs von libRIPC und Sockets nicht einfach ausgetauscht werden können, mussten alle Quelltextstellen, die mit Sockets arbeiten, überprüft und bei Bedarf entsprechend umstrukturiert werden.

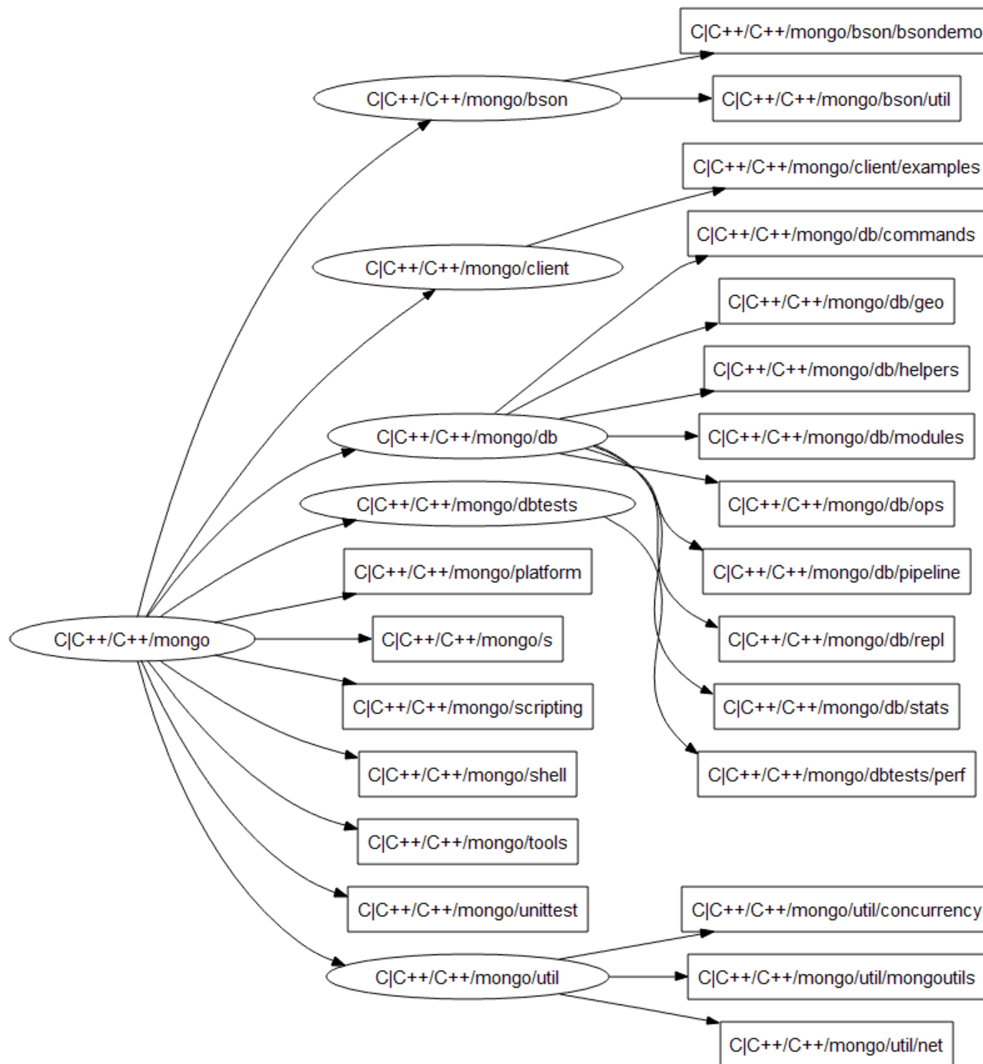

Abbildung 5.1: Die Softwarearchitektur von MongoDB. util/net ist von besonderem Interesse.

Der umfangreiche Abhängigkeitsgraph aus Understand weist 25 Klassen auf, die von der Mongosocketklasse abhängen. Problematisch sind dabei diejenigen, die sich nicht in der Netzwerkumgebung befinden, sondern von ausserhalb direkt auf die Sockets zugreifen, anstatt die Netzwerkfunktionen über die Messageschicht anzusprechen. Wir vermuten, dass es bei der Entwicklung von MongoDB nicht vorgesehen war, die Socketlösung einfach durch eine andere Netzwerkbibliothek ersetzen zu können.

MongoDB wurde in der Sprache C++ entwickelt, nutzt an zahlreichen Stellen allerdings auch Sprachelemente aus C. Inbesondere beim Reservieren von Speicher wird häufig malloc, statt new verwendet. Aufgrund der Tatsache, dass Speicher der mit new reserviert wurde, nicht einfach mit free freigegeben werden darf, muss bei jeder Modifikation klar sein um welchen Speicher es sich gerade handelt. Dadurch wird die Portierung erschwert, da die Einarbeitung in den Quelltext mehr Zeit in Anspruch nimmt.

Die meisten Änderungen wurden in der Message-Klasse und in den Serverroutinen vorgenommen, die nun statt auf die Socket-Klassen, optional auf die neu entwickelten RIPC-Klassen verweisen. Die Socketaufrufe ausserhalb der Netzwerksektion wurden entweder mit äquivalenter Funktion versehen, oder durch Platzhalter ersetzt. Ausserhalb der neu geschriebenen Klassen wurde an 27 Stellen minimalinvasiv in den MongoDB Quelltext eingegriffen. Welche strukturellen Änderungen nötig waren, wurde auf Seite 11 im Kapitel 3 Design im Abschnitt 3.2 Kommunikation beschrieben.

Im Allgemeinen ist der erwartete Portierungsaufwand für Projekte dann minimal, falls eine Netzwerkklasse existiert, die von der Transportschicht soweit abstrahiert, dass sie nicht mehr strikt an das Socketkonzept gebunden ist.

5.2.1 Softwareaktualisierungen

Der Aufwand, auf Aktualisierungen von MongoDB zu reagieren, hängt stark davon ab, welche Teile der Software betroffen sind. Im Allgemeinen sind keine Schwierigkeiten zu erwarten, falls die Netzwerkschicht nicht betroffen ist. Alle Aktualisierungen außerhalb der bei der Portierung geänderten Dateien, können automatisch durch eine geeignete Versionsverwaltung erfolgen. Verbleibende Änderungen sind dagegen manuell einzuarbeiten. Wenige Wochen vor der Fertigstellung dieser Arbeit erschien eine größere Aktualisierung von MongoDB, die nun hinsichtlich des zu erwartenden Aktualisierungsaufwands untersucht wird. Portiert wurde die Version 2.2.0, bei der Aktualisierung handelt es sich um die Version 2.4.1. Im offiziellen Repository von MongoDB sind zwischen diesen beiden Versionen allein 142 Änderungen verzeichnet, die mit Dateien aus dem Verzeichnis src/mongo/util/net zu tun haben. Insbesondere die Messageklasse und die Serverroutinen, in welchen die meisten Änderungen vorgenommen wurden, weisen 51 und 24 Änderungen auf. Da also offensichtlich sehr aktiv an der Netzwerkschicht gearbeitet wird, ist ein erhöhter Aktualisierungsaufwand zu erwarten.

5.3 Testumgebung

In diesem Abschnitt wird die Hard- und Softwareumgebung beschrieben, mit welcher die Funktionalität getestet und die Leistung gemessen wurde. Im Anschluss daran werden die verwendeten Testmethoden erläutert und schließlich die Ergebnisse vorgestellt und zur Diskussion gestellt.

5.3.1 Hardware

Getestet wurde auf einem vergleichsweise kleinen Infiniband Cluster mit sechs Knoten. Jeder Knoten ist mit einem Intel Xeon E31230 Quadcore CPU bestückt, welcher mit 3.20 GHz getaktet ist und über 8 MByte Cache verfügt. Jeder Knoten verfügt außerdem über 16 GByte DDR3 RAM, der mit 1.3 GHz betrieben wird. Das Infiniband Netzwerk besteht aus Mellanox ConnectX-3 QDR Adaptern, die an einem Mellanox InfiniBand QDR Switch angeschlossen sind und dem Netzwerk damit eine effektive Bandbreite von 40 GBit/s verleihen, die durch den Einsatz der PCI-Express 2.x8 Schnittstelle allerdings auf 32 GBit/s beschränkt werden. Auf den Knoten werden 64-bit Versionen von CentOS 6.2, basierend auf dem Linux-Kernel 2.6.32 eingesetzt.

5.3.2 Software

Zwar existieren einige Umgebungen, um die Leistung von MongoDB zu untersuchen, allerdings gab es zum Zeitpunkt der Messungen keine, die ausschliesslich auf dem C++-Treiber basierte. Die meisten verwendeten Java [18] [26] oder Python [16]. Da der Aufwand einen Treiber zu portieren, der innerhalb einer dynamischen Speicherumgebung wie einer virtuellen Maschine läuft, nicht unerheblich ist [12], wurde ein eigenes Testsystem auf Basis des portierten C++-Treibers entwickelt, das sich an der Funktionsweise anderer Testsysteme orientiert.

5.4 Versuchsaufbau

Die Laborbedingungen zeichnen sich dadurch aus, dass synthetische Daten und Operationen verwendet werden, die in einer realen Umgebung in dieser Form nicht oder nur sehr selten vorkommen. Sie liefern dafür aber reproduzierbare Messergebnisse und erlauben somit eine Einschätzung, inwieweit sich verschiedene Implementierungen voneinander hinsichtlich der Leistung unterscheiden. Diese Tests können auch dazu verwendet werden, sich an bestmögliche Bedingungen heranzutasten.

Um die Latenz zu testen, wurden Einfügeoperationen konstanter Länge mit randomisierten Daten an den Server geschickt und die Zeitstempel zwischen den Paketen aufgezeichnet. Die Differenzen dieser Zeitstempel wurden statistischen Verfahren wie einer Medianberechnung unterzogen, um aussagekräftige Werte zu erhalten. Die einzuspeisenden Daten wurden randomisiert, um mögliche Cache-Effekte sowohl auf Client-, als auch auf Serverseite ausschliessen zu können. Die Erzeugung der Testdaten, sowie die Berechnungen der Differenzen fanden ausserhalb der Zeitmessungen statt, um die Messergebnisse nicht zu verfälschen. Um den Durchsatz zu testen, wurden randomisierte Daten fester Länge in die Datenbank geschrieben und daraus gelesen und die Zeit gemessen, nach welcher die Daten vollständig übertragen waren. Dabei wurden kurze und lange Nachrichten sowohl separat als auch gemischt getestet, um die unterschiedlichen Eigenschaften von Short- und Longmessages von libRIPC zu berücksichtigen. Des Weiteren wurden während der Messungen auch die CPU-Auslastungen aller Threads in Prozent aufgezeichnet.

Die Arbeitsbelastungen vom YCSB bestehen jeweils aus verschiedenen Operationen, die in einem gegebenen Verhältnis zueinander auftreten sollen. Damit wird eine heterogene Folge von Operationen simuliert, wie sie auch in einer realen Anwendung anzutreffen ist .

5.5 Ergebnisse

5.5.1 Allgemeines

In diesem Kapitel werden jeweils drei verschiedene Netzwerkschnittstellen gegenübergestellt.

- Sockets, betrieben in einem Gigabit Ethernet Netzwerk (1G Eth)
- Sockets, betrieben in einem IP-over-Infiniband-Netzwerk (IPoIB)
- libRIPC, betrieben in einem Infiniband-Netzwerk (libRIPC)

Der Vergleich zum Gigabitnetzwerk dient in erster Linie als Referenz, da MongoDB für derartige Netzwerke entwickelt wurde. Wir vermuten, dass es in MongoDB hinsichtlich der Netzwerkgeschwindigkeit einige Annahmen und Optimierungen gibt.

5.5.2 Laborbedingungen

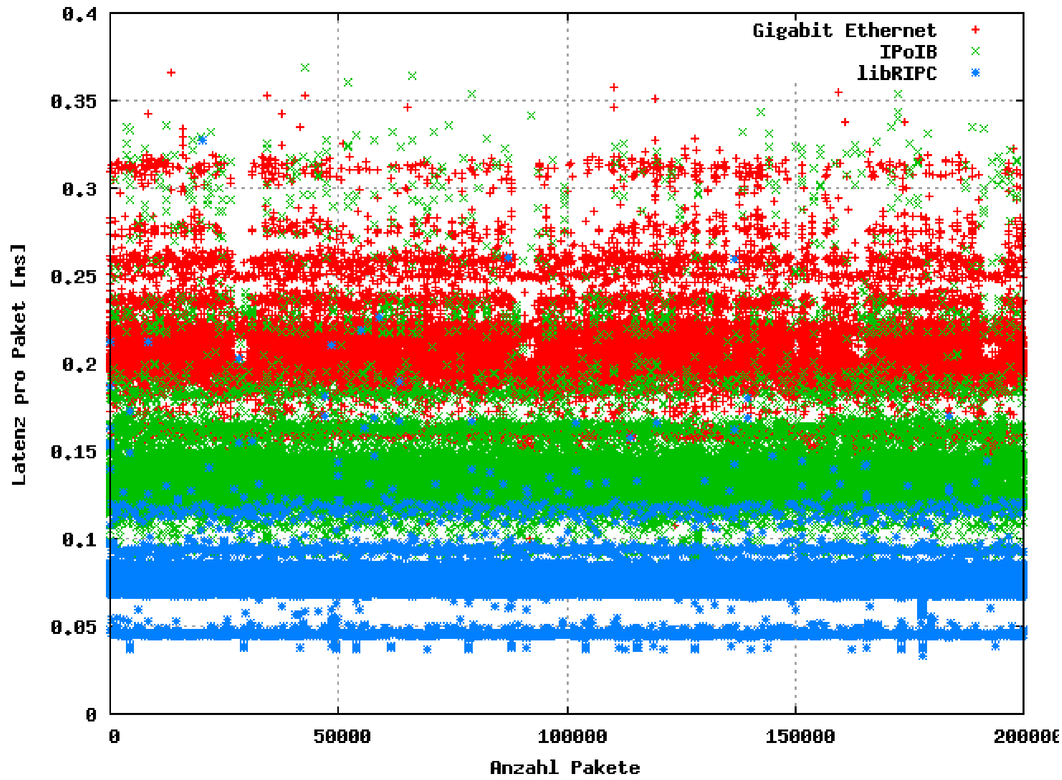

Abbildung 5.2: Latenz in ms beim Schreibvorgang unter Verwendung einer konstanten Nachrichtengröße von 100 Byte.

	Verbindungsaufbau [ms]
1G Eth	1.13
IPoIB	0.66
libRIPC	60.59

Tabelle 5.1: Zeit in Millisekunden, die zum Verbindungsaufbau benötigt wurde.

In Tabelle 5.1 sind die Messwerte für die Dauer des Verbindungsaufbaus dargestellt. Die gemessenen Zeiten sind bei libRIPC stets höher als mit Sockets. Wir vermuten, dass das an dem erhöhten Aufwand liegt, der für das Reservieren und Aushandeln der Receive-Windows benötigt wird. Für eine präzise Messumgebung ist es daher notwendig, sich ausschliesslich auf die einzelnen Paketlaufzeiten und nicht etwa auf die Gesamtdauer dividiert durch die Paketanzahl zu beziehen.

In Abbildung 5.2 ist die Latenz beim Schreibvorgang von Nachrichten mit einer Nutzlast von 100 Byte abgebildet. Durch die Größe der Nutzlast ist sichergestellt, dass ausschliesslich Shortmessages für den Transport per libRIPC verwendet werden und die Messwerte somit vergleichbar bleiben. In diesem Fall sind die Messwerte auch im erwarteten Bereich: Gigabit Ethernet ist langsamer als IPoIB, welches wiederum langsamer als libRIPC ist. Trotz der Streuungen und Ausreißer ist die Latenz in allen drei Fällen auch nach sehr vielen Nachrichten noch konstant. Da die Anzahl der Nachrichten die Größe des RipcBufferPools bei weitem übersteigt, kann man durch diesem Test auch verifizieren, dass das Wiederverwenden der RipcBuffer funktioniert. Durch eine Analyse des verwendeten Hauptspeichers konnte auch ein Speicherleck seitens der Portierung ausgeschlossen werden.

	100 B [ms]	1 KB [ms]	100 KB [ms]	1 MB [ms]	16 MB [ms]
1G Eth	0.18	0.23	2.63	10.43	190.85
IPoIB	0.13	0.18	0.50	2.07	23.41
libRIPC	0.09	0.16	0.78	2.39	20.57

Tabelle 5.2: Mittlere Zeit in Millisekunden, die zum Einfügen von Nachrichten verschiedener Größe benötigt wird.

In Tabelle 5.2 werden tabellarisch die durchschnittlichen Zeiten angegeben, die für Einfügeoperationen unter Verwendung verschiedener Nachrichtengrößen benötigt wurden. Wie zu erwarten war, unterscheiden sich die Werte je nach Nachrichtengröße stark. Besonders auffällig ist, dass libRIPC im Bereich 100 Byte bis 1 KByte besser als IPoIB abschneidet, danach aber wieder schlechter wird. Wir vermuten, dass dieses Verhalten auf die Grenze zwischen Short- und Longmessages zurückzuführen ist. Denn für Nachrichten größer als 1 KByte werden Longmessages statt Shortmessages verwendet, die erst mit größeren Nutzdaten effizient werden [12]. Derart selektive Tests erlauben einem zwar Einblicke in das Laufzeitverhalten bei der Verwendung bestimmter Paketgrößen, lassen aber keine Rückschlüsse auf die Bereiche dazwischen zu. Darum wurde im Folgenden auch ein Test durchgeführt, in dem die Nachrichtengröße sukzessive zunimmnt, um herauszufinden, ab welcher Größe die Vorteile von libRIPC signifikant zum Tragen kommen.

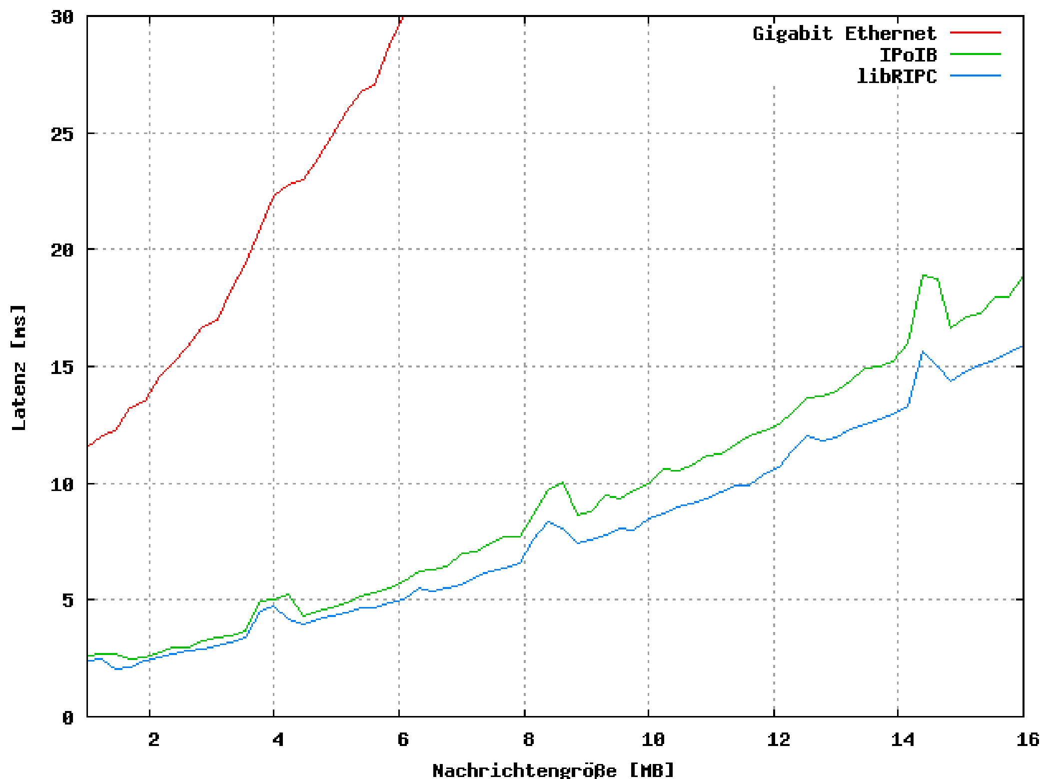

Abbildung 5.3: Latenz in ms beim Schreibvorgang in eine leere Datenbank in Abhängigkeit der Nachrichtengröße in MByte. Getestet wurde im Bereich zwischen 1 MByte und 16 MByte.

Ab etwa 7 MByte Nachrichtengröße ist der erste Vorteil durch libRIPC erkennbar, der auch stetig zunimmt. Allerdings wird der Leistungszuwachs stark durch MongoDB beschränkt, da die maximale Nachrichtengröße auf 16 MByte limitiert ist. Ein weiteres Merkmal, welches durch diesen Test zum Vorschein kommt, sind die Spitzen bei 1, 4, 8 und 14 MByte, die beim Betrieb mit Sockets im Gigabitnetz kaum erkennbar sind. Zu diesen Zeitpunkten hat MongoDB die Datenbankdateien vergrößert und mit Nullen überschrieben. Dieser Vorgang ist offenbar darauf optimiert, unter Gigabitbedingungen Clientseitig praktisch nicht aufzufallen. Da das InfiniBand-Netzwerk aber um Größenordnungen schneller ist, macht es sich in der Latenz deutlich bemerkbar.

5.5.3 Arbeitsbelastungen

Im YCSB sind sechs *workloads* angegeben, die jeweils mit 10 Feldern und 100 Bytes Daten pro Feld arbeiten, was einem Datenbankeintrag von 1 KByte entspricht.

Pro Test ist ausserdem die Wahrscheinlichkeit der beteiligten Operationen definiert. Im Detail sind das folgende Tests [27]:

- a) Lesen/Schreiben im Verhältnis 50 zu 50
- b) Lesen/Schreiben im Verhältnis 95 zu 5
- c) Lesen
- d) Lesen/Einfügen im Verhältnis 95 zu 5
- e) Suchen/Einfügen im Verhältnis 95 zu 5
- f) Lesen/Lesen-Ändern-Schreiben im Verhältnis 50 zu 50

Da MongoDB zum Zeitpunkt der Messungen keine effiziente Volltextsuche unterstützte [1], wurde der Test Suchen/Einfügen nicht durchgeführt. Dafür wurde das initiale Einfügen der Testdaten ebenfalls gemessen. Die Testserie wurde außerdem noch um einen reinen Schreibtest ergänzt. In Anlehnung an YCSB wurden folgende Tests durchgeführt: Einfügen, Lesen/Schreiben 50/50, Lesen/Schreiben 95/5, Lesen, Schreiben, Lesen/Lesen-Ändern-Schreiben 50/50

Zwischen den Tests wurde jeweils für fünf Sekunden pausiert, damit eventuell verzögerte Schreiboperationen auf dem Server, die anderen Tests nicht beeinflussen. Anlass zu dieser Entscheidung, waren Einträge in der Logdatei des Servers, in der sowohl bei IPoIB als auch bei libRIPC einzelne Schreiboperationen als *slow-queries* bezeichnet wurden, obwohl sämtliche Anfragen sowohl die gleiche Länge als auch einen identischen Aufbau hatten. Beim Betrieb mit der Gigabit-Schnittstelle kam das nicht vor. Wir vermuten daher, dass der Server eingehende Anfragen verzögert verarbeitet, um kurze Lastspitzen überbrücken zu können. Falls die Daten jedoch dauerhaft schneller eingehen, als sie verarbeitet werden können, kommt es periodisch immer wieder zu Verzögerungen. Diese Beobachtungen erschweren sowohl die Messungen als auch die Interpretation der gemessenen Daten.

	100 B [ms]	1 KB [ms]	100 KB [ms]	1 MB [ms]	16 MB [ms]
1G Eth	0.52	0.56	2.41	11.42	55.79
IPoIB	0.40	0.43	0.70	3.24	24.58
libRIPC	0.57	1.35	0.92	4.04	18.78

Tabelle 5.3: Mittlere Zeit in Millisekunden, die im YCSB pro Nachricht unter Verwendung verschiedener Nachrichtengrößen benötigt wird.

In Tabelle 5.3 werden tabellarisch die durchschnittlichen Zeiten in Millisekunden aufgeführt, die für einzelne Nachrichten im YCSB unter Verwendung

[1] Wird ab Version 2.4 unterstützt: http://docs.mongodb.org/manual/release-notes/2.4-overview/

verschiedener Nachrichtengrößen benötigt wurden. Hier ist libRIPC bis zur Nachrichtengröße 1 MByte durchgehend langsamer als IPoIB und teilweise sogar langsamer als Gigabit. Da bei diesen Tests auch gelesen wird und beim Lesen viele Zusatzinformationen wie zum Beispiel das interne Feld *_id* mitversendet werden, welches aus einer 24-Byte langen Hexadezimalen Zeichenkette, sowie einer textuellen Beschreibung besteht, kommt es bereits bei kleinen Nutzdatenmengen zum Einsatz von Longmessages, die jedoch erst ab einer größeren Menge Nutzdaten effizient werden.

Da die Latenz für Shortmessages vielversprechend aussieht, betrachten wir, wie libRIPC unter realen Bedingungen mit kleinen Datenmengen zurechtkommt.

	T 1 [ms]	T 2 [ms]	T 3 [ms]	T 4 [ms]	T 5 [ms]	T 6 [ms]
1G Eth	0.18	0.56	0.53	0.56	0.37	0.71
IPoIB	0.13	0.41	0.38	0.38	0.30	0.54
libRIPC	0.09	0.52	0.69	0.79	0.22	0.90

Tabelle 5.4: Mittlere Zeit in Millisekunden, die für einzelne Tests mit 100 Bytes Nutzdaten benötigt wird.

In Tabelle 5.4 sind die Messwerte dargestellt, die im Umgang mit kleinen Nachrichten in der Größe von 100 Bytes gemessen wurden. Es fällt auf, dass libRIPC beim Schreiben besser, beim Lesen allerdings schlechter abschneidet. Das ist in erster Linie darin begründet, dass beim Schreiben vorreservierte Puffer benutzt werden können, beim Lesen allerdings neue Speicherstellen reserviert und wieder freigegeben werden müssen. Des weiteren kommen durch die Verwendung von Shortmessages noch gelegentliche Paketverlust hinzu, die sich in der Latenz ebenfalls negativ auswirken.

	T 1 [ms]	T 2 [ms]	T 3 [ms]	T 4 [ms]	T 5 [ms]	T 6 [ms]
1G Eth	0.45	0.66	0.54	0.59	0.45	0.67
IPoIB	0.18	0.48	0.47	0.49	0.35	0.63
libRIPC	0.16	0.64	0.98	1.11	0.32	4.99

Tabelle 5.5: Mittlere Zeit in Millisekunden, die für einzelne Tests mit 1 KBytes Nutzdaten benötigt wird.

In Tabelle 5.5 sind die Messwerte dargestellt, die im Umgang mit Nachrichten in der YCSB-Standardgröße 1 KBytes gemessen wurden. Auch hier sind die reinen Schreibzugriffe mit libRIPc schneller als mit IPoIB, die Lesezugriffe allerdings wesentlich langsamer als in Tabelle 5.4. Das liegt wie oben bereits beschrieben, an dem Umstand, dass beim Lesen etwas mehr Daten übertragen werden, was

zu einem Wechsel zu Longmessages führt, die jedoch erst ab einer größeren Menge Nutzdaten effizient werden.

	T 1 [ms]	T 2 [ms]	T 3 [ms]	T 4 [ms]	T 5 [ms]	T 6 [ms]
1G Eth	190.85	160.58	146.03	146.02	185.08	146.63
IPoIB	23.41	34.32	21.79	19.44	34.54	17.18
libRIPC	20.57	26.41	29.12	28.88	29.25	22.06

Tabelle 5.6: Mittlere Zeit in Millisekunden, die für einzelne Tests mit 16 MByte Nutzdaten benötigt wird.

In Tabelle 5.5 sind die Messwerte dargestellt, die im Umgang mit Nachrichten mit der maximalen Größe 16 MByte gemessen wurden.

5.5.4 Durchsatz

	100 B[MB/s]	1 KB[MB/s]	0.1 MB[MB/s]	1 MB[MB/s]	16 MB
1G Eth	0.53	4.35	38.02	95.88	83.84
IPoIB	0.77	5.56	200.09	483.09	683.47
libRIPC	1.11	6.25	128.21	418.41	777,83

Tabelle 5.7: Mittlerer Durchsatz in MByte/s, der für Nachrichten verschiedener Größe erreicht wird.

In Tabelle 5.7 ist der Durchsatz in MByte/s aufgeführt. Wie erwartet, können mehr Daten pro Zeit transportiert werden, wenn wenige große, statt viele kleine Datenbankeinträge verwendet werden. Aus der Tabelle ist ersichtlich, dass das Gigabitnetzwerk unter Verwendung von großen Nachrichten ausgelastet wird, im Infinibandnetzwerk dagegen noch reichlich Kapazitäten verfügbar sind. Im folgenden Abschnitt 5.5.5 wird jedoch klar, dass das Netzwerk nicht den limitierenden Faktor darstellt.

5.5.5 Prozessorbelastung

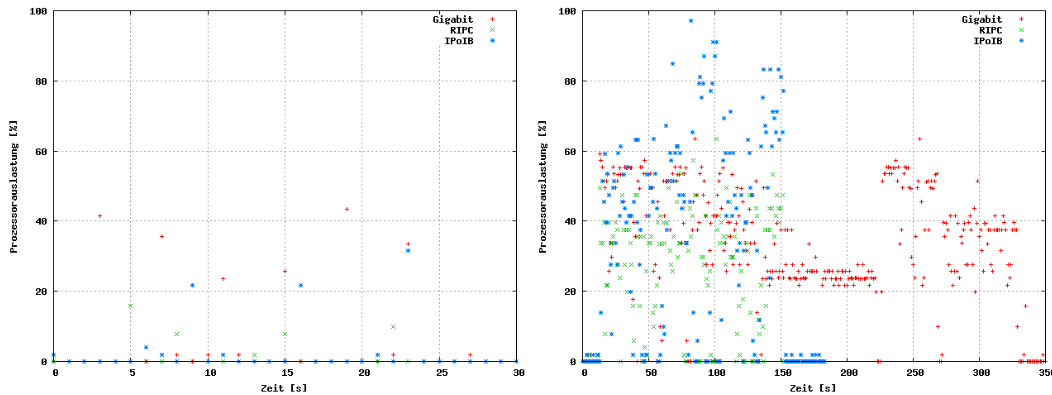

Abbildung 5.4: Serverseitige CPU Auslastungen bei einem YCSB mit Einträgen zu 1 MByte bzw. 16 MByte.

In Abbildung 5.4 sind die Prozessorbelastungungen aller Threads auf dem Server dargestellt. Wie erwartet, hängt die Belastung mit der Nachrichtengröße zusammen. Die Maximalwerte von Gigabit Ethernet, IPoIB und libRIPC betragen bei der Verwendung von 16 MByte Nachrichten 63.4, 97.1 und 63.4 Prozent. Der Median beträgt 33.6, 38.6 und 27.7 Prozent.
Wir sehen, dass die Prozessorbelastung mit libRIPC geringer ausfällt als mit IPoIB und können die Messergebnisse aus [12] somit bestätigen.

Auffällig ist, dass die Prozessorbelastung von MongoDB über IPoIB derart hoch ist, sobald die maximale Nachrichtengröße verwendet wird. Da die maximale Größe einer BSON-Struktur in MongoDB auf Nutzeranfragen hin, erst nachträglich von 4 MByte auf 16 MByte erhöht wurde [4], wurde die Datenbank dementsprechend ursprünglich für kleinere Einträge konzipiert, was die massive CPU Auslastung beim Betrieb mit 16 MByte Einträgen erklärt.

5.5.6 Bewertung

In diesem Abschnitt haben wir uns mit der Messung von Latenz, Prozessorbelastung und Durchsatz unter verschiedenen Bedingungen beschäftigt. Dabei wurden sowohl erwartete, als auch unerwartete Werte festgestellt und unter den gegebenen Bedingungen genauer betrachtet.

Zusammenfassend kann festgestellt werden, dass MongoDB unter Verwendung von libRIPC beim Schreiben von kleinen oder großen Nachrichten besser abschneidet als beim Betrieb über IPoIB, bei gemischten Operationen allerdings erst mit zunehmender Nachrichtengröße Vorteile bietet. Wir haben auch herausgefunden, dass MongoDB hinsichtlich der Prozessorbelastung, mit vielen kleinen Nachrichten wesentlich effizienter umgehen kann, als mit wenigen großen. Unter Berücksichtigung dieser Randbedingungen ist der Einsatz von libRIPC in MongoDB vorteilhaft.

5.5.7 Zukünftige Arbeit

Da Datenbanken wie MongoDB offensichtlich mit vielen kleinen Nachrichten zu tun haben, wäre es hinsichtlich libRIPC interessant, Netzwerkanwendungen zu portieren, die auch mit vergleichsweise großen Daten arbeiten, wie zum Beispiel FTP-, NFS- oder RSYNC- Server.

Kapitel 6

Fazit

In dieser Arbeit wurde die Portierung einer verteilten Datenbank auf einen Hochleistungscluster behandelt. Dazu haben wir uns erst die Softwarearchitektur von MongoDB angeschaut und versucht, alle Schwierigkeiten im Vorfeld zu erkennen und durch ein passendes Design zu bewältigen. Daraufhin haben wir die Designentscheidungen im Rahmen der Implementierung umgesetzt und die Lösung im Anschluss unter Laborbedingungen erfolgreich getestet. Nach einigen Messungen mit unterschiedlichsten Rahmenbedingungen, wurde die Implementierung auch noch realistischeren Tests unterzogen. Wir konnten durch diese Arbeit sowohl bekannte Tendenzen aus anderen Veröffentlichungen bestätigen, als auch sehr viel Neues bezüglich MongoDB aufzeigen.

Literaturverzeichnis

[1] 10gen. bsonspec. http://bsonspec.org.

[2] 10gen. Mongodb. http://mongodb.org.

[3] 10gen. Mongodb customers. http://www.10gen.com/customers.

[4] 10gen. Mongodb: Increase the 4mb bson object limit to 16mb. https://jira.mongodb.org/browse/SERVER-431.

[5] F. Baiao. Horizontal fragmentation in object dbms: new issues and performance evaluation. University of Rio de Janeiro, February 2000.

[6] Sitha Bhagvat. Designing and enhancing the sockets direct protocol (sdp) over iwarp and infiniband. Ohio State University, 2006.

[7] Daniel P. Bovet and Marco Cesati. The claremont report on database research. pages 63–76, November 2013.

[8] Sanjay Ghemawat Wilson C. Hsieh Fay Chang, Jeffrey Dean and Robert E. Gruber. Bigtable: A distributed storage system for structured data. Harvard University, November 2006.

[9] Jian Huang. High-performance design of hbase with rdma over infiniband. pages 774–785, May 2012.

[10] infinibandta. infiniband. http://www.infinibandta.org.

[11] J. Jose. Memcached design on high performance rdma capable interconnects. pages 743–752, September 2011.

[12] Jens Kehne. Light-weight remote communication for high-performance cloud networks. Master's thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, May 2012.

[13] N. Leavitt. Will nosql databases live up to their promise? pages 12–14, February 2010.

[14] J. Liedtke. Microkernels must and can be small. pages 152–155, October 1996.

[15] Mehul Shah Alistair Veitch Marcos K. Aguilera, Arif Merchant and Christos Karamanolis. Dynamo: Amazon's highly available key-value store. Harvard University, October 2007.

[16] Wisdom Omuya. Benchmark script for the mongodb server. `https://github.com/mongodb/mongo-perf`.

[17] openfabrics. libiverbs. `http://www.openfabrics.org`.

[18] PolePosition project. Poleposition: the open source database benchmark. `http://www.polepos.org/`.

[19] Thomas Robertaskwi. Basics of computer networks. pages 25–28. Springer, 2012.

[20] Han S., Marshall S., B. Chun, and Ratnasamy S. Megapipe: A new programming interface for scalable network i/o. http://research.yahoo.com, 2012.

[21] Inc. Scientific Toolworks. Understand: Source code analysis and metrics. `http://www.scitools.com/`.

[22] Sayantan Sur. Scalable and high-performance mpi design for very large infiniband clusters. The Ohio State University, 2007.

[23] Andrew S. Tanenbaum. Computernetzwerke. Pearson Studium, 2003.

[24] Andrew S. Tanenbaum und Marten van Steen. Verteilte systeme. pages 17–19. Pearson Studium, 2003.

[25] Jim / James C. Warner. Procps: The /proc file system utilities (top). `http://procps.sourceforge.net/`.

[26] Yahoo. Yahoo cloud service benchmark. `http://research.yahoo.com/node/3202`.

[27] Yahoo. Yahoo cloud service benchmark workloads. `https://github.com/brianfrankcooper/YCSB`.