Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

JULEA: A Flexible Storage Framework for HPC

Projektbericht

von Lars Thoms

Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Hamburg, 22. März 2018

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Rahmen des Projekts *Parallelrechnerevaluation* sollte das Speicherframework *JULEA* weiterentwickelt werden. *JULEA* ist ein in Softwareprojekt, welches Anderen ermöglicht, verschiedene Speichersysteme zum Testen oder Lernen anzubinden.

Dank der Abstraktion Programm + Speicherlösung wird kein spezielles Hintergrundwissen zu den unterschiedlichen *API*s der Backends benötigt, sondern man kann sich auf die eigentliche Verwendung fokusieren. Die Erweiterung, welche in diesem Projekt enstanden ist, ermöglicht es, ein weiteres Backend anzubinden und es wurde die Grundlage für ein Austausch des Netzwerkinterfaces geschaffen.

1.2 Projektziele

Der erste Meilenstein bestand aus der Implementation eines weiteren Speicherbackends – gewünscht wurde sich eine Unterstützung des Objektspeichers *Ceph*.

Des Weiteren sollte als zweiter Meilenstein die Netzwerkkomponente, die zwischen dem *JULEA*-Client und -Server für die Konnektivität sorgt, ausgetauscht werden. Momentan wird die `TCPSocket` Implementierung von *GLib* genutzt. Diese war gegen das Netzwerkframe *Mercury* auszutauschen.

Zum Schluss (3. Meilenstein) stand noch eine kleine Dokumentation in Form eines GitHub Wikis an, welche es Einsteigern ermöglicht, das Projekt noch besser zu nutzen.

1.3 Aufbau

Dieser Projektbericht ist in mehrere Etappen aufgeteilt. Zu Beginn werden die verwendeten Hauptprojektkomponenten und deren Verwendungszwecke vorgestellt. Es folgt eine Auflistung der für die Realisierung des Projekts genutzten Soft- und Hardware.

Der Einschub *Schnelles Netzwerken* knüpft an *Mercury* an und ist Bestandteil des *Integrierten Seminars*. Die darin vorgestellten Netzwerktechniken geben einen Einblick, weshalb eine Netzwerkabstraktion, wie *Mercury*, sinnvoll sein kann.

Im Abschnitt Realisierung des Projekts ist unterteilt in die Meilensteine. Enthalten sind Designs, Kommentare zur Implementation und die Kompileranweisungen. Der Bericht schließt mit Empfehlungen für weiterführende Arbeiten und einer Zusammenfassung der erreichten Ziele.

2 Die Projektkomponenten

2.1 JULEA

Das Framework *JULEA* ist die Hauptkomponente des Projekts. Es handelt sich hierbei um einen Adapter für verschiedene Speicherlösungen, mit dem Ziel, recht unkompliziert diese Backends in einem Forschungs- oder Lehreumfeld zu testen und zu debuggen. Dies wird weiterhin durch die Tatsache erleichtert, dass *JULEA* vollständig im Userspace (Ring 3), läuft und kein Kernelmodul ist.

JULEA ist komplett in `C` unter Zuhilfenahme der *GLib*-Bibliothek programmiert. Als *Buildsystem* kommt das in *Python* geschriebene *WAF* zum Einsatz. Bisher werden zwei verschiedene Typen von Speicherphilosophien unterstützt:

- Key-Value Speicher
- Objektspeicher

Die Unterschiede zwischen den beiden Konzepten sind je nach Implementation fließend. Bei einem *Key-Value*-Speicher werden kleine Datenmengen zu einem selbstdefinierten Schlüssel gespeichert. Dies sind öfters Meta- oder Konfigurationsdaten, welche ein Schlüsselpaar bilden und hervorragend in einer datenbankähnlichen Struktur gespeichert werden können.

Objektspeicher arbeiten mit erheblich größeren Datenmengen, wobei Objekte quasi Dateien sind, welche mit Hilfe eines eindeutigen Schlüssels in den Speicher abgelegt werden. Diese Systeme unterstützen zumeist die Replikation zur Lastverteilung und zur Ausfallverringerung, aber auch die Wiederherstellung von Daten im Fehlerfall.

JULEA unterstützte bisher die Speicherkomponenten *POSIX*, *LevelDB*, *MongoDB*, *SQLite*, *LMDB* und *GIO*. Einige der Komponenten im Client-, andere im Serverbetrieb. Dies bedeutet, dass die Datenübermittlung entweder vom Client direkt oder über den *JULEA*-Server an das Speicherbackend geht.

Das Framework wurde hauptsächlich von Michael Kuhn aus der Arbeitsgruppe *Wissenschaftlich Rechnen* der Universität Hamburg entwickelt und ist auf der Entwicklungsplattform *GitHub* zu finden:
`https://github.com/wr-hamburg/julea/`

Die wissenschaftliche Abhandlung über dieses Framework ist im entsprechenden Paper [5] zu finden.

2.2 Ceph

Ceph ist ein Objektspeicher, welches an der *University of California, Santa Cruz* von der Arbeitsgruppe um Sage Weil ab dem Jahre 2007 als Teil seiner Dissertation entwickelt wurde. Das Projekt gründete sich als *Inktank Storage* aus der Universität heraus und wurde 2014 von *Red Hat Inc.* übernommen.

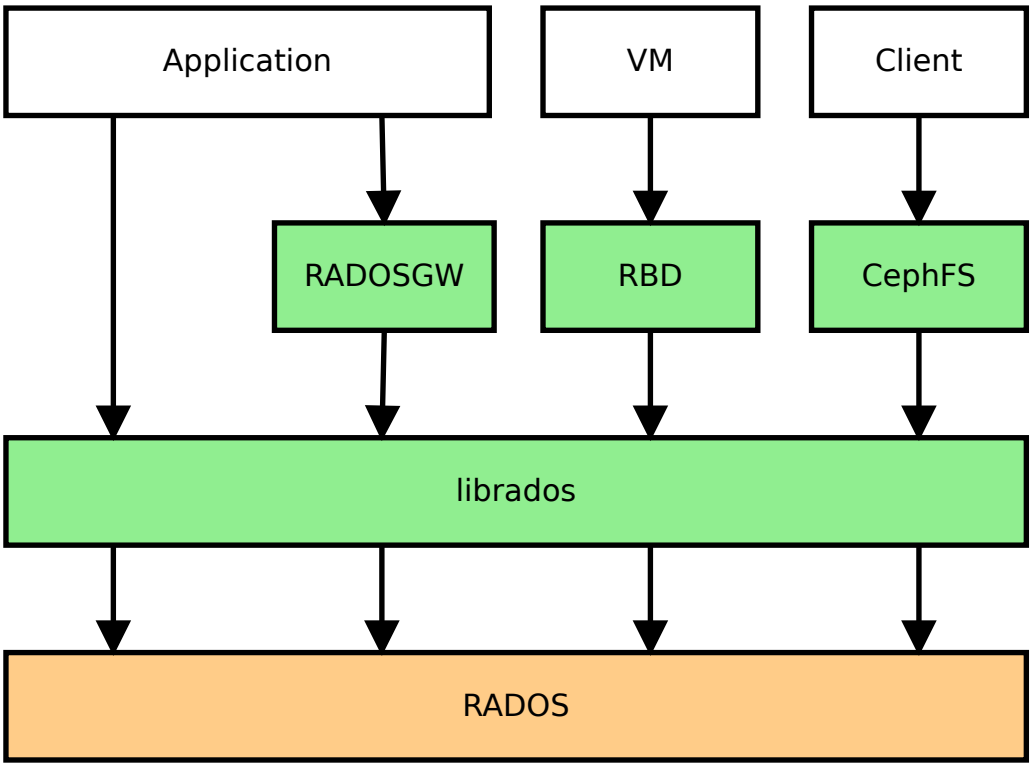


Abbildung 1: Architekturdiagramm von Ceph (Quelle: [7, Abbildung 8])

Es besteht aus mehreren Modulen (siehe Abbildung 1), welche im folgenden näher erläutert werden:

- **RADOS** – Das Herzstück des Projekts. Dies ist der Objektspeicher, wohin alle Daten abgelegt werden.
- **librados** – Eine `C++`-Bibliothek, welche Programmierern eine Schnittstelle zum Objektspeicher anzubietet. Vergleichbare Schnittstellen existieren auch für andere Programmiersprachen, beispielsweise für `C`, `Python` oder `Java`.
- **RADOSGW** – Ein Gateway für Programme, die eine RESTful API benötigen. Sie ist kompatibel zu anderen cloudbasierten Protokollen wie *Amazon S3* und *OpenStack Swift*.
- **RBD** – Ein Blockdevice, welches quasi als Festplatte in virtuelle Maschinen eingebunden werden kann.
- **CephFS** – Eine POSIX-Schnittstelle, welche jedoch einen Metadatenserver als weitere Komponente benötigt.

Im Projekt wird hauptsächlich die **librados**-Bibliothek verwendet, da diese eine API anbietet, welche in `C` genutzt werden kann.

Genauere Informationen zu Ceph finden sich hier: [7, Seite 15ff].

2.3 Mercury

Die dritte Projektkomponente ist ein *RPC*-Framework. *RPC* steht für *Remote Procedure Call* und kann als Übermittlung von kurzen Nachrichten, welche Anweisungen oder Parameter enthalten, genutzt werden. Es ist ebenfalls in `C` geschrieben und besitzt keine weiteren Abhängigkeiten. Des Weiteren existieren verschiedene Plugins, die unterschiedliche Protokolle zur Datenübermittlung anbieten.

Abgesehen vom *RPC* können auch größere Datenmengen als sogenanntes *Bulk Data* übermittelt werden. Dies geschieht in der Regel via *RDMA*, welches im Abschnitt *InfiniBand* näher erläutert wird.

3 Genutzte Software/Hardware

Für das Verfolgen und, falls nötig, Reviedieren von Änderungen wurde das Versionskontrollsystem *git* in Kombination mit der Entwicklungsplattform *Github* verwendet. Ausschlaggebend für die Wahl dieser Plattform war, dass das Projekt *JULEA* bereits dort gehostet worden ist. Dementsprechend wurde die Entwicklung auf einem Fork vorangetrieben und mittels *Pull Requests* (siehe Abbildung 2) in den bereits vorhandenen Hauptentwicklungsstrang eingepflegt.

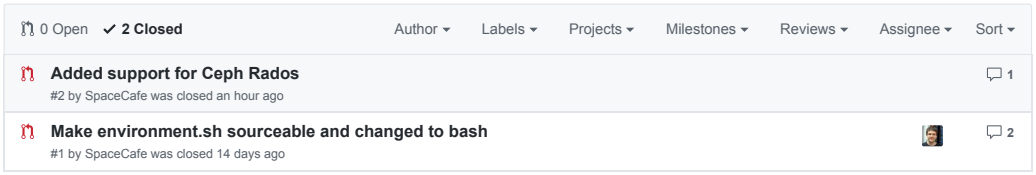


Abbildung 2: Pull Requests bei GitHub

Zum Programmieren, Dokumentieren und Bericht schreiben wurden die Textbearbeitungsprogramme *Sublime Text 3* und *vim* verwendet. Insbesondere wurde für die Erstellung des Berichts eine Kombination aus *pandoc* und *pdflatex (texlive)* genutzt, damit zwischen Markdown, der eigenen LaTex Vorlage und ein PDF konvertiert werden konnte.

Die Schaugrafiken und Zeichnungen wurden mit *Dia Diagram Editor* erstellt – Screenshots und weiterführende Grafiken wurden entsprechend mit *Inkscape* (vektorbasierend) oder *GIMP* (pixelbasierend) verarbeitet.

Auf Kommunikationsdienste, wie etwa E-Mail oder Jabber, wurde verzichtet, da an diesem Projekt nur eine Person beteiligt war. Daraus ergibt sich entsprechend, dass keine Synchronisationsdienste für Features oder Bugs vonnöten waren. Diese wurden analog und ohne Zuhilfenahme von elektronischen Hilfsmitteln auf einem Whiteboard festgehalten.

4 Einschub: Schnelles Netzwerken

4.1 Problemstellung HPC

HPC: *engl.: high-performance computing*
Berechnung eines komplexen Problems, das nicht mehr durch herkömmliche Computer gelöst werden kann. Dafür muss das Problem parallelisiert werden, damit es auf einem verteilten Cluster berechnet werden kann.

Das Hauptproblem heutiger Programme im wissenschaftlichen Umfeld ist deren Komplexität und die daraus folgende Berechnungsdauer. Diese Berechnungsprobleme müssen parallelisiert werden, damit auf mehreren Knoten (Computern) gleichzeitig an einem Teilproblem gearbeitet werden kann. Ansonsten würde die Lösung des Problems Jahre in Anspruch nehmen.

Wenn ein Programm parallelisiert werden soll, stellt sich die Frage: Wie lässt sich dieses Problem sinnvoll partitionieren? Im besten Fall wird eine große Matrix in kleinere Zellen aufgeteilt. Leider ist dies nicht immer möglich, da öfters ein Rechenungleichgewicht zwischen den Zellen einer Matrix herrscht. Auch sollten Überschneidungen zwischen Partitionen vermieden werden. Ansonsten sind temporäre Sperren und die Übermittlung des aktuellen Stands notwendig.

Dies würde dazu führen, dass eine vollständige Sicht auf die Daten essentiell ist. Denn dies führt dazu, dass alle Knoten den kompletten Datensatz geladen haben und ständig aktuell halten müssen. Dies ist ein sehr hoher, meist nicht notwendiger, Kommunikationsaufwand.

Ein weiteres Problem ist die räumliche Distanz zwischen Rechenknoten. Es gibt manchmal legitime Gründe, dass mehrere Rechenzentren an einem Problem rechnen, weil die Kapazität eines Zentrums nicht ausreicht oder ein größeres teurer / nicht verfügbar ist. Bestenfalls sollten diese dann exklusiv miteinander verbunden sein, was aber nicht immer gegeben sein muss. Hier gilt eine sorgfältige Auswahl der Dienstleister.

4.2 Kriterien zur Netzwerkbeurteilung

Um den Netzwerkverkehr und das -verhalten über die Zeit quantisiert beurteilen zu können, benötigt man vor der entgültigen Entscheidung für oder gegen eine Netzwerktechnologie eine Netzwerkanalyse über die Zeit. Die Abbildung 3 zeigt ein fiktives, mögliches Ergebnis einer Analyse des Netzwerkverkehrs über die Zeit von einem Knoten.

Die erste Phase des Programmes ist die Initialisierung. Dabei werden die benötigten Dateien über das Netzwerk von einem SAN kopiert. Während dieser Zeit finden keine Berechnungen statt. Erst in der nächsten Phase wird mit den vorhandenem Datenset gerechnet. Daraufhin findet ein Datenaustausch mit den benachbarten Knoten statt. Diese Kombination wiederholt sich, bis ein Abschluss der Berechnungen ansteht. Das Gesammtergebnis wird ebenfalls auf das SAN geschrieben.

Bei der Auslotung eines passenden Netzwerkes mit Hilfe folgender Kriterien sollten die zu verwendeten Programme betrachtet werden:

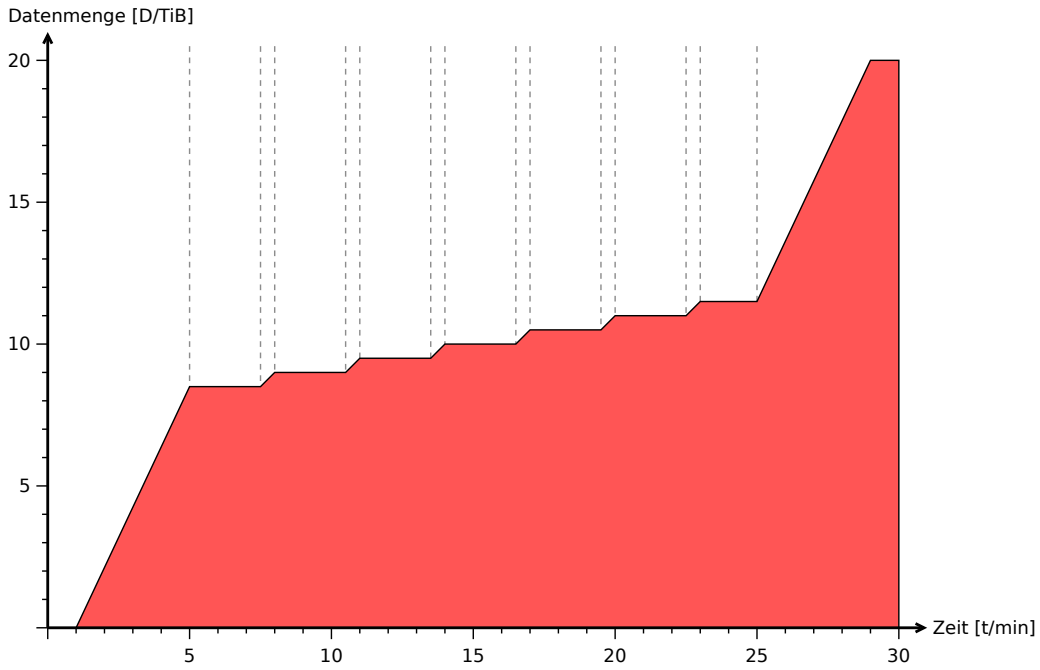


Abbildung 3: Datenmenge-Zeit-Diagramm eines fiktiven Programms

1. Häufigkeit (Intervall)
   Wie häufig kommuniziert das Programm? Geschieht dies in einem festen Intervall, oder eher sporadisch? Wenn ein Programm selten kommuniziert, können unter Umständen geringere Datenraten für das gesamte Netzwerk festgelegt werden, falls nicht alle gleichzeitig kommunizieren wollen oder mehrere Terabytes übertragen werden sollen.
   In dem Beispielprogramm wird sehr regelmäßig eine Datenmenge von 0.5 TiB übermittelt.

2. Regelmäßigkeit vorhanden?
   Es gibt nichts besseres als geplante Kommunikationsabläufe. Diese lassen sich recht einfach kalkulieren und einplanen. Mit Regelmäßigkeit ist nicht nur das zeitliche Intervall, sondern auch die Menge der zu übermittelnden Daten und die wiederkehrenden Kommunikationspartner gemeint.

3. Hohe Latenz ein Problem?
   Lädt das Programm nur zu Beginn der Berechnungsphase die Daten? Dann sind höhere Latenzen verschmerzbar. Falls jedoch ein Programm recht häufig während der Rechenphasen mit anderen Knoten kommunizieren muss, sind niedrigere Latenzen meistens relevant, je nachdem ob (a)synchron übertragen wird.
   Bei dem Beispiel finden sowohl am Anfang als auch am Ende der Programmlaufzeit größere Datentransfers statt. Während der Berechnung ist das Datenaufkommen recht niedrig. Ob die Daten asynchron geschrieben und gelesen werden lässt sich hieraus nicht ablesen. Dafür werden weitere Spezifikationen des Programms benötigt oder weitere Tests mit verschiedenen Latenzen.

4. Synchronizität wichtig?
   Bei synchroner Datenübertragung muss der Zielknoten seine Berechnungsphase pausieren, um die Kommunikation zu bearbeiten. Im Zweifel ist die Latenz hoch und die Partner reagieren nicht immer sofort. Dadurch entstehen recht hohe Wartezeiten für die Berechnung. Deshalb bieten sich für fast alle Fälle asynchrone Übertragungen an, deren Daten beim Zielknoten in einem Zwischenspeicher liegen, bis das Programm diese abrufen möchte – oder im Falle von *RDMA* (siehe Abschnitt 4.5.2) direkt im Zielspeicherbereich vorliegen.

5. Menge der übermittelnden Daten
   Je größer die Datenmenge ist, desto höher müsste theoretisch die mittlere Datenrate des Netzwerkes auslegt werden, wenn man einen Zeitrahmen t festlegt. Dies hängt aber stark davon ab, ob die Knoten hauptsächlich mit ihren Nachbarn oder mit einem zentralen SAN kommunizieren. Dadurch entscheidet sich die benötigte maximale Datenrate zwischen den Knoten und zu anderen Netzwerkkomponenten, wie beispielsweise das SAN.
   Die oberste Maxime sollte nämlich immer sein, die Transmissionszeit so kurz wie möglich zu halten, da in der Zeit normalerweise keine Berechnungen stattfinden können.

4.3 Netztopologien

Topologien auf der Netzebene beschreiben den Zusammenhang zwischen Nodes (in den folgenden Abbildungen rot gekennzeichnet) und Switches/Router (grün gekennzeichnet). Dies bedeutet, sie beschreibt die Art und Weise der Vernetzung zwischen diesen Knoten.

4.3.1 Punkt-zu-Punkt (P2P)

Dies ist die simpelste Art und Weise zwei Knoten miteinander zu verbinden. Dabei ist jeder Node mit einer dedizierten Leitung verbunden, ohne dass geswitched wird (siehe Abbildung 4). Diese Form besteht üblicherweise nur aus zwei Nodes.

Abbildung 4: Topologie: Punkt-zu-Punkt

Eine schnellere Verbindung als diese gibt es theoretisch nicht, da keine Umwege über weitere Knoten genommen werden müssen – was unter Anderem Caches, eine höhere Latenz und höhere elektrische Energieumwandlung implizieren würde.

Diese Topologie könnte mehr oder weniger auch als Subtopologie der folgenden Topologien betrachtet werden, da bei jeder Netztopologie Knoten miteinander verbunden sind.

4.3.2 Bus

Eine Abwandlung der *Punkt-zu-Punkt* Topologie ist der Bus. Dabei werden alle Knoten an einem Übertragungsmedium angeschlossen (siehe Abbildung 5).

Diese Technik ist recht günstig, da nur ein Kabel verlegt werden muss, an dem sich alle Beteiligten via *T-Stück* anklemmen können. Da es sich jedoch um ein *Shared-Medium* handelt kann nur einer pro Zeit senden, wodurch jeder Knoten seine zu sendenden Daten puffern muss. Das Ergebnis ist eine recht niedrige mittlere Datenrate pro Knoten. Des Weiteren ist die Ausfallwahrscheinlichkeit höher, denn sobald an einer Stelle des Busses das Kabel beschädigt wird, unterteilt sich der Graph in zwei Subgraphen. Dadurch wird temporär die Kommunikation zwischen den entstandenen Subgraphen unterbunden.

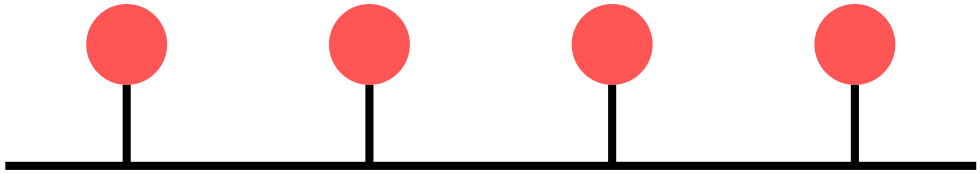


Abbildung 5: Topologie: Bus

4.3.3 Stern

Um die Probleme des Kommunikationsaufwandes für einzelne Knoten, der Unterteilung in Subgraphen und der unterschiedlichen hohen Anzahl an Sprüngen, zum Teil zu lösen, gibt es die *Sterntopologie* (siehe Abbildung 6). Sie ist eine der weit verbreitetsten, da sie beispielsweise in jedem Haushalt vorzufinden ist.

Dabei verbinden sich alle Nodes mit einem zentralen Switch, der für diese spezielle Aufgabe deutlich leistungsfähiger als ein Node ist. Dementsprechend wird die Anzahl der Sprünge auf **2** begrenzt und es ist nur noch die Ausfallwahrscheinlichkeit des Switches zu betrachten.

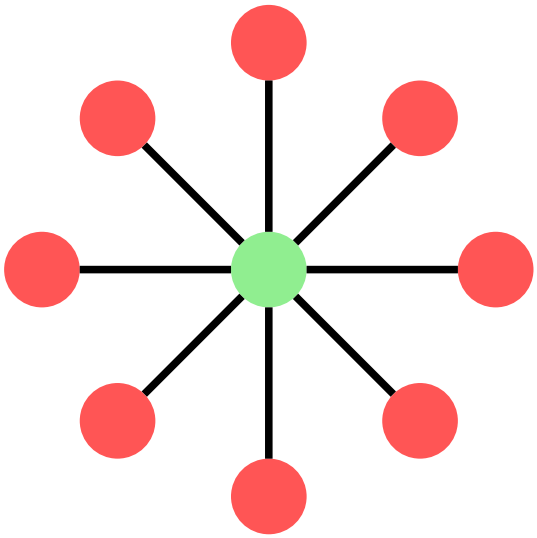


Abbildung 6: Topologie: Stern

4.3.4 Baum

Um mehrere *Sterntopologien* zu verbinden, gibt es die Möglichkeit die Switches wiederum in einer *Sterntopologie* zu verschachteln (siehe Abbildung 7). Dies führt jedoch leider dazu, dass die Datenrate der Verbindungen zwischen den beiden Subgraphen etwas geringer ist, als innerhalb der jeweiligen Subgraphen, wenn man annimmt, dass alle Knoten eines Subgraphs mit einem entsprechenden Partnerknoten des anderen Subgraphen kommunizieren möchten.

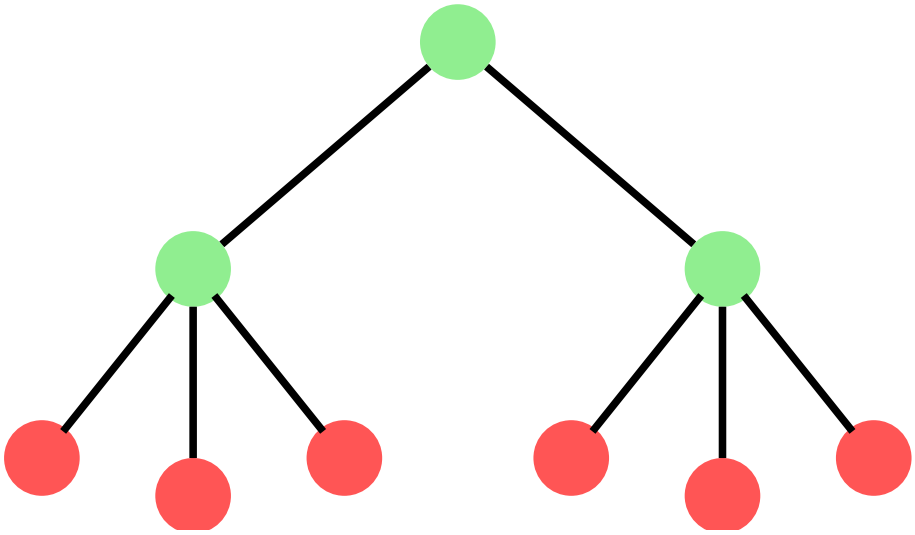


Abbildung 7: Topologie: Baum

4.3.5 FatTree

Das Problem des klassischen *Baums* mit der gleichen Datenrate im Subgraph der Switches wird beim *Fattree* in dem Sinne gelöst, dass bei den übergeordneten Switches stattdessen sehr leistungsfähige Coreswitches und Verbindungen mit einer deutlichen höheren Datenrate eingesetzt werden (siehe Abbildung 8). Möglich wäre beispielsweise, die Nodes in den Subgraphen via 1GbE (Gigabit Ethernet) zu verbinden, die Switches untereinander dagegen via 10GbE.

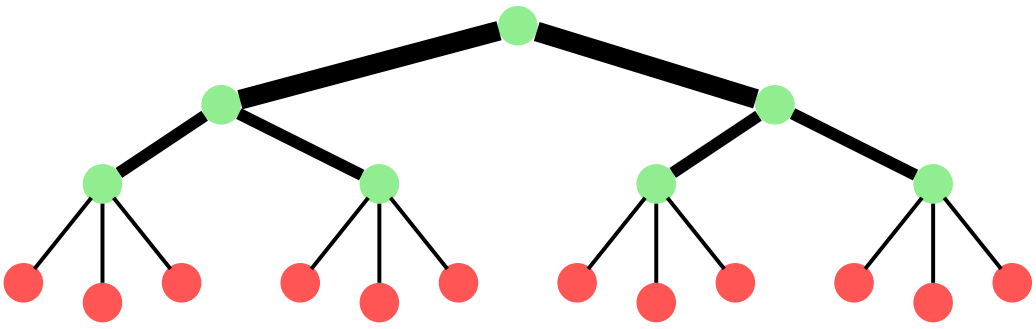


Abbildung 8: Topologie: FatTree

Dieser Typ wird in den meisten (Hochleistungs-)Rechenzentren verwendet, da die Kosten-/Nutzenrelation den meisten Anforderungen und Budgets entspricht. Es wird meistens darauf abgezielt, dass nicht alle Knoten zu jederzeit mit der maximalen Datenrate Informationen von A nach B verschicken müssen.

4.3.6 Dragonfly

Der *Dragonfly* ist eine recht besondere, im Bereich des HPCs jedoch durchaus eingesetzte Netztopologie. Ihre Struktur besteht aus mehreren Schichten und folgt bestimmten Regeln, welche im folgenden näher erläutert werden (siehe Abbildung 9).

Die erste Schicht besteht aus den Nodes und einem Switch. Diese sind in einer klassischen *Sterntopologie* angeordnet. Diese Gruppe gibt es genau m Mal je Untergruppe, welche eine zweite Schicht bilden. Die Anzahl Untergruppen in der zweiten Schicht ist n und es gilt: $n \mod 2 = 1$. Wichtig ist weiterhin, dass $n \mod m = 0$ gilt, sonst ist es nicht möglich, die Vermaschung gemäß der Regeln zu vollziehen.

Für die Switches in der zweiten Schicht ist eine Vollvermaschung vorgesehen, das bedeutet, alle Switches sind mit allen anderen verbunden.

Die dritte Schicht bildet die Vermaschung zwischen den Switches aus den unterschiedlichen Subgraphen, welche nach dem folgenden Regelset erfolgt:

1. $x = 1$ und gib den Knoten des Subgraphen an
2. Der x. Knoten des Subgraphen verbindet sich mit dem x. Knoten des nächsten, noch nicht verbundenen Subgraphen. Dies geschieht exakt $\frac{n}{m}$ Mal.
3. $x+ = 1$

Der Grund, weshalb diese Form gerne genutzt wird, ist, dass es maximal drei Sprünge bedarf, um jedes andere Subnetz zu erreichen. Dieser Vorteil führt dazu, dass keine speziellen Coreswitches, wie beim *FatTree*, benötigt werden, welche in der Anschaffung recht kostenintensiv sind.

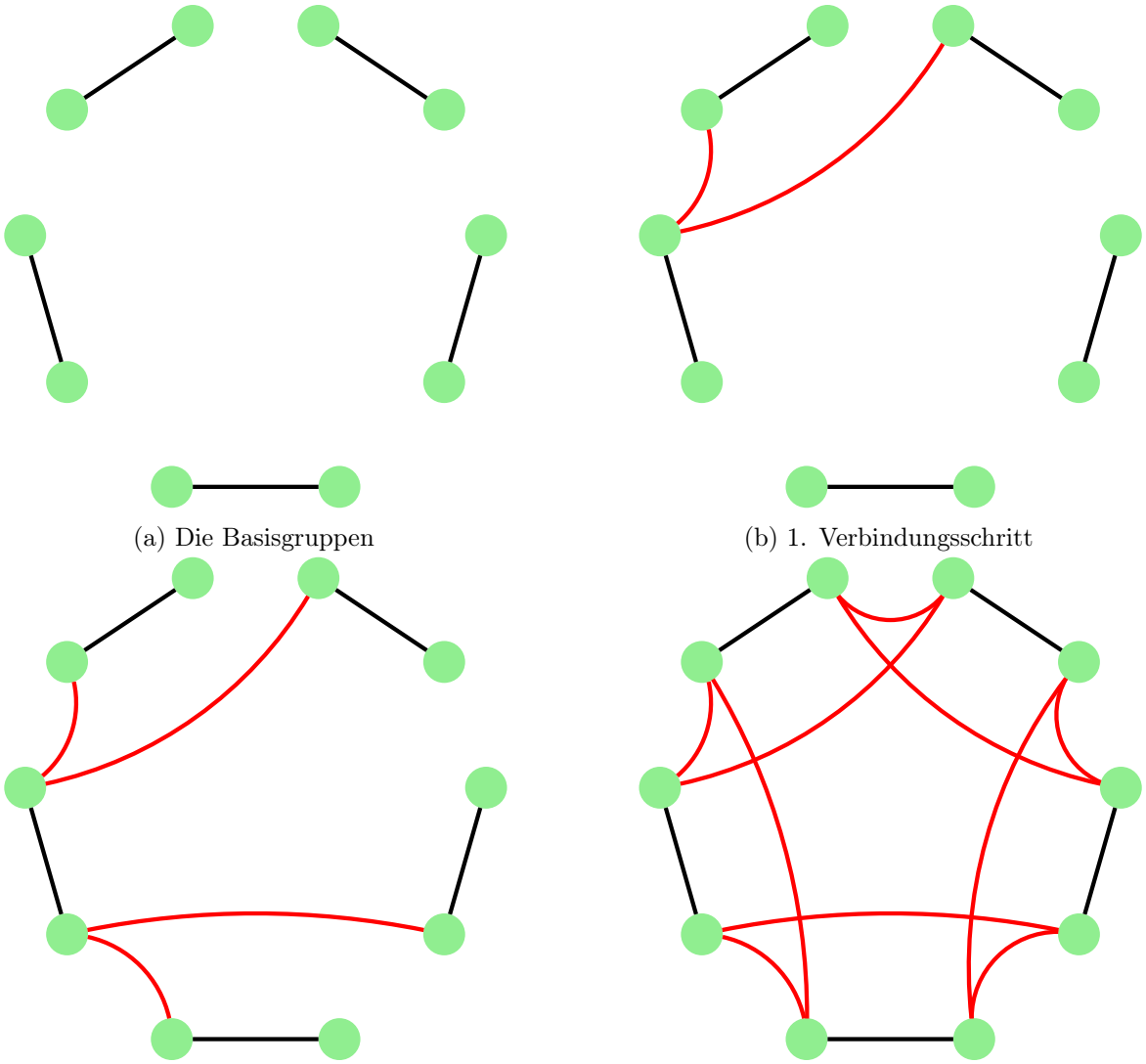
4.3.7 n-dimensionaler Torus

Die letzte vorzustellende Topologie ist der sogenannte n-dimensionale Torus. Der *Torus* ist in der Basisstruktur eine einfache Ringtopologie, welche in $n > 1$ Dimensionen in einer Gitterstruktur angeordnet wird (siehe Abbildung 10).

Bei der Betrachtung des zweidimensionalen Torus (siehe Abbildung 11) wird deutlich, dass diese Topologie sehr gut für Berechnungen im Gitter mit Kantenübergängen geeignet ist. Dies bedeutet, dass Probleme in quadratische Bereiche partitioniert werden, an dessen Kanten Knoten miteinander kommunizieren müssen.

Drei Dimensionen sind sehr gut dafür geeignet, um beispielsweise Klima- und Wetterdaten zu verarbeiten, da das Rechenproblem zusätzlich in verschiedenen Höhen unterteilt werden kann.

Leider sind in dem Fall durch den hohen Vernetzungsgrad der Knoten (sechs Netzwerkinterfaces pro Knoten in 3D) die Kosten pro Einheit sehr hoch und die Alternativen meistens ausreichend.



(a) Die Basisgruppen

(b) 1. Verbindungsschritt

(c) 2. Verbindungsschritt

(d) Vollständiger Dragonfly

Abbildung 9: Topologie: Dragonfly

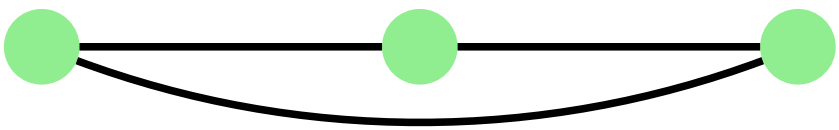
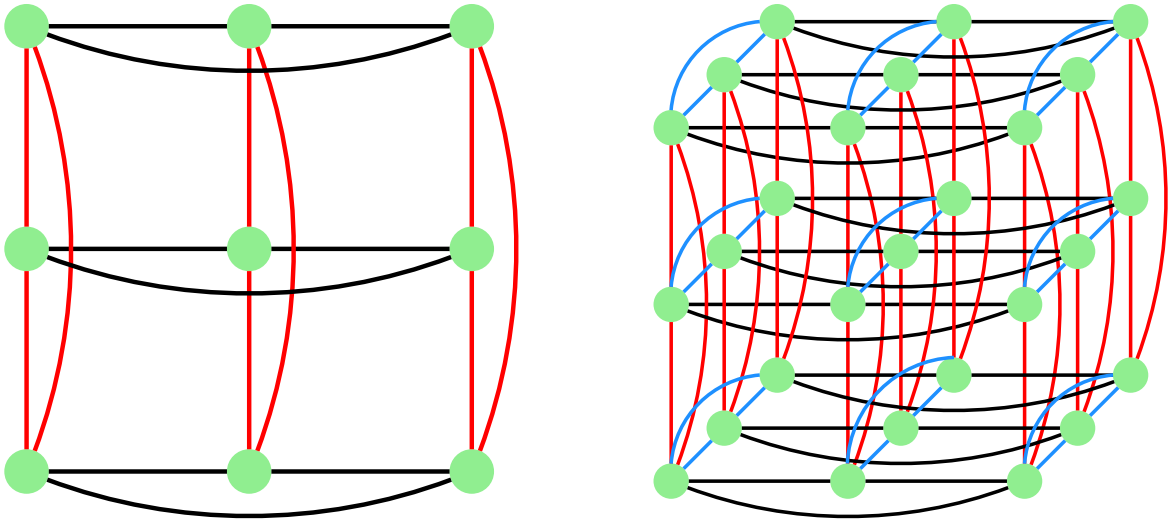


Abbildung 10: Topologie: 1D Torus



(a) 2D-Torus (b) 3D-Torus

Abbildung 11: Topologie: n-dimensionaler Torus

4.4 Nutzungsarten von Netzwerkinfrastruktur

In einem Rechenzentrum gibt es unterschiedliche Arten von Netzwerkanforderungen für verschiedene Zugriffs- und Nutzungsarten. Diese müssen nicht die selbe Infrastruktur oder Topologie teilen. Außerdem können die Verbindungen von unterschiedlicher Qualität sein.

Die Bedürfnisse können in drei Kategorien eingeteilt werden:

1. Daten- und Speicherzugriff
2. Managementzugriff
3. Interprozesskommunikation

4.4.1 Daten- und Speicherzugriff

Mit dieser Kategorie ist der Zugriff auf das zentrale Speichersystem gemeint. In Frage kommt hierfür in der Regel ein SAN *(Storage Attached Network)*. Dabei bilden mehrere Knoten/Racks gefüllt mit Speichermedien, wie beispielsweise Festplatten oder SSDs *(Solid State Disks)*, einen großen Speicherverbund, welcher nicht unmittelbar im selben Rechenzentrum stehen muss.

Programme müssen ihre umfangreichen Nutzungsdaten mit hoher Wahrscheinlichkeit initial laden. Da zu dieser Zeit keine Berechnungen stattfinden können, ist ein Uplink mit **hoher Datenrate** sinnvoll. Diese richtet sich danach, wie viele Knoten n Byte Daten laden müssen. Des Weiteren stellt sich die Frage, ob alle Knoten gleichzeitig und/oder die selben Daten laden müssen.

Müssen alle auf die gleichen Daten zugreifen, bietet sich ein *Multicast* fähiges Netzwerk an. In diesem Falle werden die Daten einmal vom Server an das Netzwerk gesendet. Multicastfähige Switches/Router können dann die entsprechenden Pakete an die anfordernden Knoten weiterleiten.

Eine niedrige Latenz ist bedingt wichtig: Müssen die Programme hauptsächlich initial Nutzdaten laden und wollen sie diese erst nach Abschluss ihrer zeitlich langen Berechnung wieder abspeichern, ist eine höhere Latenz vertretbar. Werden Daten während der Berechnungen geladen oder gespeichert, ist die Latenz nur vernachlässigbar, wenn diese I/O-Operationen asynchron von statten gehen. Ansonsten wird zu viel Zeit für das Warten verloren und der Rechenknoten ist in der Zeit nicht ausgelastet.

Über die Hochverfügbarkeit des Datennetzes lässt sich in der Hinsicht streiten, ob alle Rechenknoten intensiv Daten mit dem SAN austauschen müssen und ohne diesen nicht weiter rechnen können. Wenn nicht, lässt sich dies mit einem lokalen Zwischenpuffer für das Abspeichern umgehen.

4.4.2 Managementzugriff

Um die entsprechenden Rechenknoten und das Netzwerk zu verwalten und gegebenenfalls anzupassen, wird nur ein einfaches Netzwerk benötigt. Meistens werden Knoten via SSH *(Secure Shell)* und IPMI *(Intelligent Platform Management Interface)* ferngewartet.

Hierfür ist eine niedrige Latenz nicht zwingend notwendig. Daraus folgt, dass eine einfache Baumtopologie völlig ausreichend ist. Höhere Datenraten sind ebenfalls nicht unbedingt notwendig. Selbst wenn angenommen wird, dass die Knoten via PXE *(Preboot Execution Environment)* das Betriebssystem lokal von einem TFTP *(Trivial File Transfer Protocol)* Server beziehen, ist diese Last vernachlässigbar. Die Knoten starten selten neu und sollte dieser Fall dennoch eintreten, benötigt allein der Bootprozess des BIOS selbst schon mehrere Minuten. Ein paar Minuten mehr zum Beziehen des Images fallen dann nicht mehr ins Gewicht.

4.4.3 Interprozesskommunikation

Die Kommunikation zwischen den Knoten, während einer Berechnungsphase, ist bei den meisten Programmen sehr wichtig. Falls sie nicht asynchron ablaufen, ist eine sehr niedrige Latenz notwendig, da der Rechenprozess sowohl beim Sender als auch beim Empfänger anhält. Dadurch wird relativ viel Zeit für die Kommunikation verloren und die Recheneinheiten sind unausgelastet.

Wie üppig die Datenrate ausgestattet werden sollte, hängt sehr stark vom Kommunikationsverhalten des jeweiligen Programms ab.

Eine recht interessante Fragestellung, auch an das verwendete Protokoll, ist die optimale Paketgröße der Transmissionen. Wenn ein Programm alle paar Millisekunden sehr kleine Datenmengen asynchron überträgt, kann es sinnvoller sein, diese gesammelt in einem Paket zu übertragen, falls dies nicht zeitkritisch in einem Echtzeitsystem läuft.

4.5 Hardware

Ein wichtiger Schritt bei der Entscheidung für einen Netzwerkentwurf ist die Wahl einer geeigneten Hardware. Wurden alle obigen Punkte bedacht, besteht eine gute Entscheidungsgrundlage um zu bewerten, welche Kriterien als wichtig einzustufen sind. Dies kann beispielsweise der Preis, die Latenz,

die Datenrate oder auch die Kompatibilität zwischen verschiedenen Herstellern sein (offener Standard).

4.5.1 Ethernet

Das *Ethernet* ist verglichen mit den anderen beiden nachfolgenden Beispielen das älteste Protokoll. Es ist ursprünglich nicht auf Hochgeschwindigkeitsnetzwerken ausgerichtet gewesen, sondern auf einfache Vernetzung zwischen Computern in einem lokalen Netzwerk, welches Token Ring ablösen sollte.

Als Hardware gibt es hier sowohl elektrisch-galvanische Leiter, wie acht adrige Kupferleitungen, oder optische Leiter wie Glasfaser. Dabei kommen hauptsächlich *Multimode-* und *Singlemodefasern* zum Einsatz, welche sich durch die Anzahl an Moden und die damit verbundene maximale Leitungslänge unterscheiden.

Multimodefasern besitzen einen deutlich dickeren Kern (etwa Faktor 500 gegenüber *Singlemodefasern*), bei welchem die Totalreflektion des Kerns zum Mantel genutzt wird. Dadurch können verschiedene Modes (definierte Frequenzen) gleichzeitig durch den Lichtwellenleiter "durchreflektiert" werden. Leider ist durch die starke Reflektion der Laufweg des Licht deutlich länger als das Kabel und somit muss zum Teil eine höhere Laufzeit in Kauf genommen werden. Des Weiteren lassen sich solche Fasern nicht so lang ohne Repeater verlegen, wie eine *Singlemodefaser*, da die Dämpfung recht schnell ansteigt.

Singlemodefasern haben einen Kerndurchmesser, welcher sich aus einem kleinen Faktor der verwendeten Beleuchtungsfrequenz zusammensetzt. Die Dicke dieser Fasern ohne Mantel beträgt üblicherweise um die 1000 nm und es wird auch nur mit dieser einen Mode beleuchtet. Dadurch wird ein deutlich längerer Übertragungsweg mit geringerer Dämpfung erreicht.

Innerhalb von Rechenzentren sind *Multimodefasern* deutlich praktischer, da selten sehr lange Wege (teilweise bis zu 100 km) zu überbrücken sind. Dadurch kann mit bis zu tausend verschiedenen Modes eine einzelne Faser beleuchtet und dadurch extrem hohe Datenraten erreicht werden.

Sollen mehrere Standorte überbrückt werden, die beispielsweise einige Kilometer voneinander entfernt liegen, bietet sich eher die Verwendung mehrerer *Singlemodefasern* an. Um starke Leitungsdämpfungen zu vermeiden, sollten gegebenenfalls Repeater, also Signalverstärker, verwendet werden.

Die meisten Netzwerkinterfaces für Ethernet befinden sich schon auf der Hauptplatine, ansonsten lässt es sich via *PCIe*-Karten *(Peripheral Component Interconnect Express)* erweitern. Dadurch ist Ethernet auf jedem Rechner verfügbar. Heutzutage lassen sich mit Ethernet verschiedene Datenraten erreichen. Dabei gibt es eine Abhängigkeit von dem verwendeten Medium.

Elektrisch-galvanisch: $1\,\mathrm{Gbit\,s^{-1}}$, $10\,\mathrm{Gbit\,s^{-1}}$, $40\,\mathrm{Gbit\,s^{-1}}$ und $100\,\mathrm{Gbit\,s^{-1}}$
Optisch: wie bei *elektrisch-galvanisch*, zusätzlich $200\,\mathrm{Gbit\,s^{-1}}$ und $400\,\mathrm{Gbit\,s^{-1}}$

Die Latenzen bei Ethernet lassen leider etwas zu wünschen übrig. Die nachfolgenden Zahlen zeigen beispielhaft die Dimension, in der sich die Latenzen bewegen. Diese beruhen auf Messun-

gen mit jeweils TCP-Verbindungen Ende-zu-Ende via Switch mit der kleinsten Paketgröße (siehe [2, Tabelle 1]):

- $1\,\mathrm{Gbit\,s^{-1}}$: $29\,\mu s - 100\,\mu s$
- $10\,\mathrm{Gbit\,s^{-1}}$: $12.51\,\mu s$

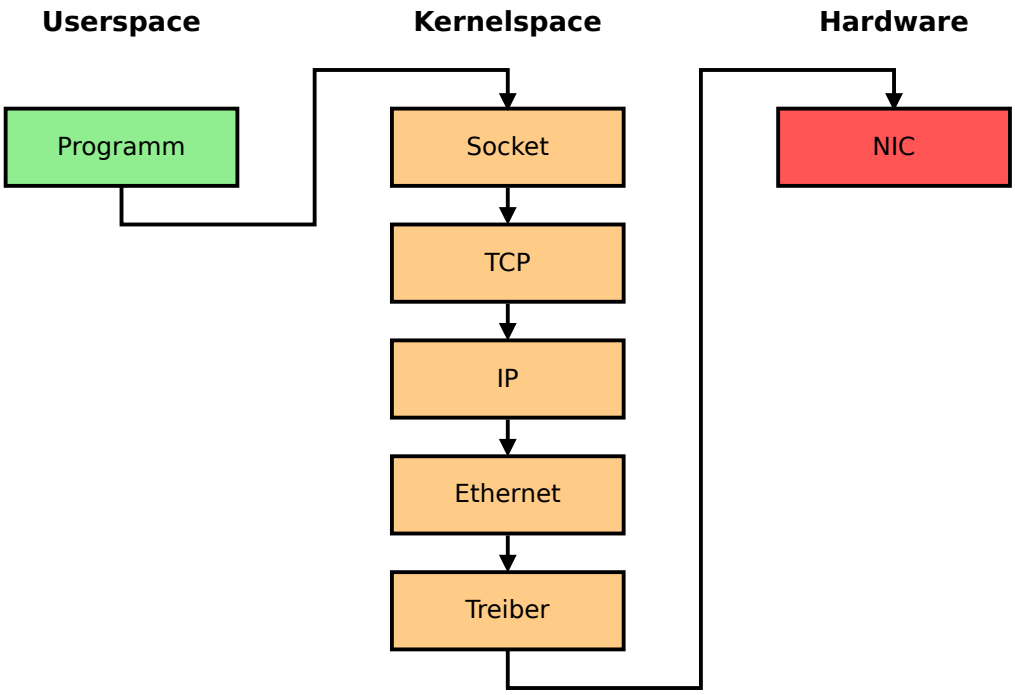


Abbildung 12: Netzwerkstack

Eine mögliche Ursache für diese recht hohen Latenzen ist der Netzwerkstack. Wie in Abbildung 12 erkennbar, agiert der klassische Ethernet-Netzwerkstack sowohl im *Userspace*, als auch im *Kernelspace*.

Wieso dies zu Problemen mit der Latenz führen kann, soll an folgendem Beispiel verdeutlicht werden: Betrachtet wird ein Programm auf einer *x86-Architektur*, welcher Daten sowohl verschicken als auch empfangen möchte. Das Programm läuft normalerweise auf *Ring 3*, eine unprivilegierte Domain im Sicherheitskonzept eines Prozessors. Wenn ein Datensatz gesendet werden soll, meldet das Programm Zugriff auf ein *Socket* an. Da dieser sich im *Kernelspace* befindet, also im *Ring 0* läuft, muss ein Moduswechsel durchgeführt werden, welcher vom Scheduler veranlasst wird. Dabei werden alle Registerbänke gesichert und mit Kerneldaten überschrieben. Dies nimmt viel Zeit in Anspruch. Um zu verhindern, dass bei vielen kleinen hintereinander folgenden Datentransmissionen jedes Mal dieser Wechsel vollzogen werden muss, gibt es den sogenannten *Socket Buffer* (siehe Abbildung 13). Dieser sammelt für einen definierten Zeitabschnitt oder bis zu einer gewissen Speichergröße Transaktionen, welche darauffolgend weiterverarbeitet werden müssen.

Im nächsten Schritt werden mehrere Datentransmissionen in den *Kernelspace* kopiert und zu *TCP*-, *IP*- und *Ethernet*-Paketen mit ihren Prüfsummen und weiteren Metadaten erweitert. Diese landen in der *Transmit queue (qdisk)*, bis sie mit Hilfe des *Treibers* in das *Netzwerkinterface (NIC)* geschrieben werden. Dort werden die ausgehende Daten in den *TX Ringbuffer* abgelegt – eingehende Daten werden in den *RX Ringbuffer* geschrieben.

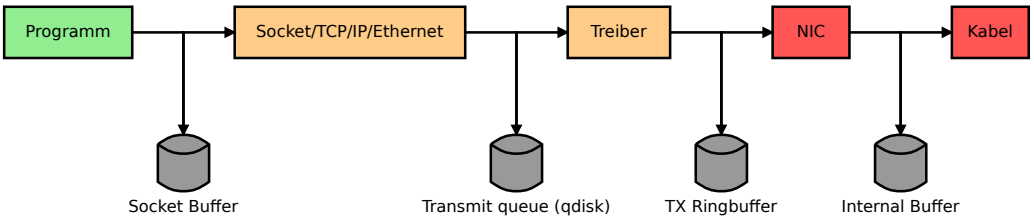


Abbildung 13: Traditionelles Puffersystem im Linux-Netzwerkstack

Nun werden die Daten von dem *Netzwerkinterface* via Kabel an die nächste Vermittlungsstelle des Netzwerks gesendet. Dies kann der Empfänger oder ein Switch sein. Diese speichern gerne auch kleinere Datenmengen zwischen, um den Netzwerkverkehr zu optimieren oder zu steuern.

Wenn Daten im *RX Ringbuffer* des Zielrechners ankommen, wird ein *Interrupt Request (IRQ)* an den Kernel gemeldet, welcher zeitnah bearbeitet werden muss. Dafür ist wieder ein *Moduswechsel* notwendig. Die restlichen Schritte bestehen aus der Überprüfung der Prüfsummen und dem Bereitstellen der Daten im *Socket Buffer*, damit das Programm sie lesen kann.

Moderne *Netzwerkstacks* und Betriebssysteme enthalten einige Verfahren, um einige der Zwischenspeicherungen zu umgehen. Im Nachfolgenden werden ein paar ausgewählte vorgestellt.

Die erste Methodik ist das *Zero-Copy*-Verfahren. Hierbei wird das Kopieren, also duplizieren von Nutzdaten, vermieden. Stattdessen werden die Adressen zu den jeweiligen Speicherbereichen weitergegeben.

Prozessoren mit der Befehlssatzerweiterung *Advanced Vector Extensions (AVX)* können zudem das *Scatter Gather* anwenden. Dabei wird ein Vektor als Pufferdeskriptor genutzt, welcher aus dem Datenpointer, der Länge der Nutzdaten und dem nächsten Deskriptor besteht. Dadurch wird erreicht, dass die Metadaten (TCP/IP- und Ethernetframes) zu den jeweiligen Paketen getrennt von den Nutzdaten im Speicher liegen.

Unterstützt das *Netzwerkinterface* zusätzlich *Direct Memory Access (DMA)*, können dementsprechend diese Vektoren direkt an das *NIC* durchgereicht werden, welcher alle Daten in der richtigen Reihenfolge aus dem Speicher lädt und über das angeschlossene Kabel verschickt.

Eine noch weitreichendere Technik ist der *Kernel Bypass*. Hierbei gibt es verschiedene Vorgehensweisen (`PACKET_MMAP`, `PF_RING`) und fertige Frameworks (*Snabbswitch*, *DPDK* und *Netmap*), um die Moduswechsel zum *Kernel* zu minimieren oder gar zu vermeiden. Für weitergehende Informationen siehe [1].

4.5.2 InfiniBand

Dieser Protokoll- und Hardwaretyp ist wie Ethernet ein offener Standard, was bedeutet, dass es ebenfalls mehrere Hersteller für Peripherie gibt. Weiterhin gibt es ebenfalls sowohl die Möglichkeit die Daten elektrisch-galvanisch über Kupfer als auch optisch via Glasfasern zu übertragen.

Die spezifizierten Datenraten bei *InfiniBand* sind recht ähnlich zu Ethernet und die hohen sind ebenfalls nur mit Glasfaser möglich: $10\,\mathrm{Gbit\,s^{-1}}$, $25\,\mathrm{Gbit\,s^{-1}}$, $40\,\mathrm{Gbit\,s^{-1}}$, $50\,\mathrm{Gbit\,s^{-1}}$, $56\,\mathrm{Gbit\,s^{-1}}$, $100\,\mathrm{Gbit\,s^{-1}}$ und $200\,\mathrm{Gbit\,s^{-1}}$

Im Gegensatz zum Netzwerkinterface von Ethernet ist der *Host Channel Adapter (HCA)* nicht auf der Hauptplatine verankert. Er muss fast immer mit Hilfe einer *PCIe*-Karte nachgerüstet werden.

Der größte äußerliche Unterschied zu Ethernet ist die niedrige Latenz, welche auch gleich als der große Vorteil von *InfiniBand* gilt. Beispielwerte sind unter anderem $1.72\,\mu\mathrm{s}$ bei *DDR* oder $1.67\,\mu\mathrm{s}$ mit *QDR* (siehe [2, Tabelle 1]).

Der Hauptgrund für diese niedrige Latenz ist die Verwendung von *Remote Direct Memory Access (RDMA)* (siehe Abbildung 14).

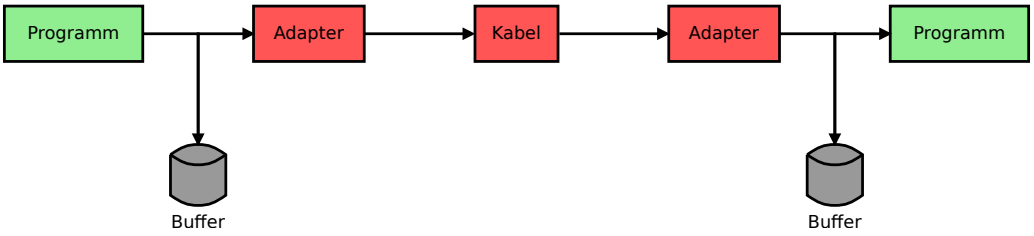


Abbildung 14: Remote Direct Memory Access bei Infiniband

Durch die Verwendung des *RDMA* wird erreicht, dass der *HCA* direkt in den Speicher des Zielrechners schreiben kann. Dies funktioniert recht ähnlich zum vorher beschriebenen *Scatter Gather* bei *Ethernet*. Zuerst werden die entsprechenden Datenpointer und Längen der jeweiligen Nutzdaten ermittelt und an den *HCA* übermittelt. Dieser meldet dem Zielrechner den Bedarf an n-Byte Speicherplatz im Hauptspeicher. Die entsprechenden freien Bereiche werden vom gegenüberliegenden *HCA* zurückgemeldet und der *HCA* schreibt die Nutzdaten direkt in diese freien Blöcke des fremden Hauptspeichers hinein.

4.5.3 Intel OmniPath

Intel OmniPath ist gegenwärtig eine proprietäre Lösung von Intel. Sie baut quasi auf dem *InfiniBand* Standard auf und verändert einige Bestandteile, um eine bessere mittlere Datenrate zu erzielen.

Als Übertragungsweg ist nur der optische mit einer Datenrate von $100\,\mathrm{Gbit\,s^{-1}}$ spezifiziert. Weiterhin wird das Kabel direkt am (neuen) Prozessor angeschlossen, um so die Laufwege zu verkürzen. Bisher existiert diese Schnittstelle an den *Intel Xeon Phi* und *Intel Xeon SP*. Für die anderen bietet Intel eine *PCIe* Karte, wie sie auch bei *InfiniBand* Verwendung findet.

Es sind mehr oder weniger drei wichtige Änderungen oder Verbesserungen gegenüber *InfiniBand* zu nennen:

1. *Forward Error Correction* wird simplifiziert durch einen einfachen 14 bit CRC
   Die Prüfsumme für die Datenübertragung wird durch ein schnelleres, aber schlechteres Verfahren ersetzt. Es wird davon ausgegangen, dass die Anzahl an Fehlern so gering sei, dass es wiederum sinnvoller sei diese wenigen Male einfach neu zu senden.

2. *Traffic Flow Optimization:* Mehrere *Flow Control Units (Flits)* werden in eine Transaktion unifiziert
   Dies spare Zeit, da auch bei *InfiniBand* oder *OmniPath* ein minimaler Overhead an Verwaltung benötigt werde. Zum Beispiel bei der Vereinbarung der freien Speicherbereiche.
3. *Priorisierte Transaktionen*
   Es gibt nun die Möglichkeit, größere Transaktionen zu unterbrechen/pausieren, um eine wichtige Mitteilung zu übermitteln. Dies ist bei *InfiniBand* bisher leider nicht möglich.

5 Realisierung

5.1 Unterstützung von Ceph

Ceph ist ein verteiltes Objektspeichersystem und der Projektteilnehmer hat durch vorangegangene Forschungen Expertise mit diesem System gesammelt. Dementsprechend ist es ideal als weiteres Speicherbackend für *JULEA*. Für die Implementierung wird die Bibliothek `librados` benötigt, die eine direkte Verbindung mit dem Speichersystem aufnehmen kann. Dadurch benötigt der Benutzer nur diese Bibliothek und keine weiteren Abstraktionen.

5.1.1 Installieren von `librados`

Ein Vorteil von `librados` ist, dass es in den meisten Linuxdistributionen in den Repositories vorhanden ist. Dadurch ist ein manuelles kompilieren nicht mehr notwendig.

Unter Debian wird einfach `apt` benutzt,

```
1 apt install librados-dev
```

oder unter Fedora das Verwaltungswerkszeug `dnf`.

```
1 dnf install librados-devel
```

5.1.2 Design

JULEA unterstützt zwei verschiedene Arten von Zugriffen auf die Speichersysteme (siehe Abbildung 15):

1. Direkt
   Hierbei wird ohne eine Serverkomponente von *JULEA* auf das System zugegriffen. Der Vorteil hierbei ist, dass kein Umweg über einen weiteren Knoten genommen werden muss. Nachteilig ist, dass der Client die entsprechenden Bibliotheken des jeweiligen Speicherbackends auf dem eigenen Computer installiert haben muss.

2. Indirekt
   Die Verbindung zum Speicher wird über einen *JULEA* Server abgewickelt. Der Client braucht keine weiteren Bibliotheken und die Daten werden quasi durch einen Proxy geschleust.

Wichtig bei *Ceph* ist, dass dem Client eine Konfigurationsdatei `ceph.conf` und der entsprechende Schlüssel vorliegen. Diese werden dem *RADOS*-Client übergeben, damit dieser eine Verbindung zum Speichersystem aufnehmen und sich authentifizieren kann. Diese Dateien werden normalerweise von der Benutzerverwaltung des Speicherbackends ausgegeben.

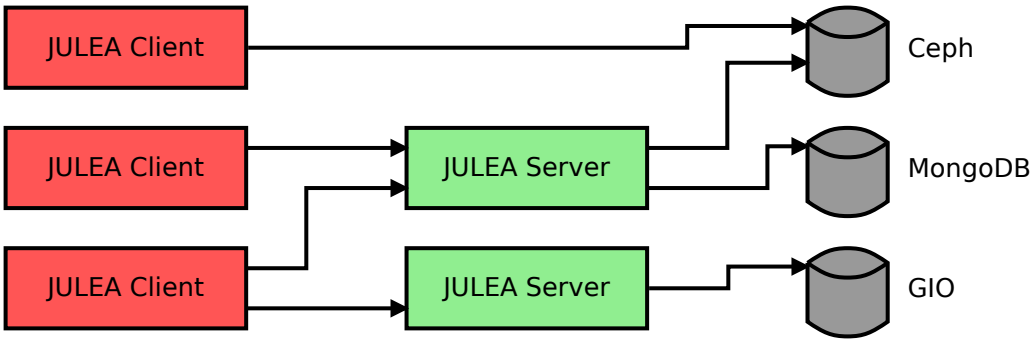


Abbildung 15: Verbindungsmöglichkeiten von *JULEA* – es sind sowohl direkte, als auch indirekte Verbindungen möglich. Ebenfalls können verschiedene Clients auf verschiedene Server zugreifen: eine m-n Beziehung

5.1.3 Implementation

Ceph wurde sowohl für eine direkte, als auch eine indirekte Verbindungsvariante implementiert. Die Implementation beider Varianten ist identisch, es unterscheidet sich nur der Ordner, in welcher der Quellcode zur Kompilierzeit liegt.

Der Quellcode liegt vollständig in der Datei `rados.c` und befindet sich in den beiden Ordnern `backend/client/`. Der entsprechende Callgraph ist in Abbildung 16 zu sehen.

Damit das System das neue Backend erkennt, gibt es eine Standardfunktion, welche bei jeder Implementation aufgerufen wird: `backend_info()`. Hierbei wird übermittelt, um welchen Speichertyp es sich handelt und die Liste der implementierten Funktionen.

```
1  G_MODULE_EXPORT JBackend* backend_info (JBackendType type)
2  {
3      JBackend* backend = NULL;
4
5      if(type == J_BACKEND_TYPE_OBJECT)
6      {
7          backend = &rados_backend;
8      }
9
10     return backend;
11 }
```

Diese Liste an implementierten Funktionen wird in einem `struct` als Funktionspointer übergeben. Dadurch entsteht ein Mapping der entsprechenden I/O-Operation auf die zu nutzende Funktion. Außerdem kann darüber ermittelt werden, welche Operationen überhaupt unterstützt werden.

```
1  static JBackend rados_backend = {
2      .type = J_BACKEND_TYPE_OBJECT,
3      .object = {
```

```
        .init = backend_init,
        .fini = backend_fini,
        .create = backend_create,
        .delete = backend_delete,
        .open = backend_open,
        .close = backend_close,
        .status = backend_status,
        .sync = backend_sync,
        .read = backend_read,
        .write = backend_write
    }
};
```

Die Initialisierungsfunktion `backend_init()` benötigt eine korrekt gesetzte Pfadinformation. Sie setzt sich aus dem Pfad zur Konfigurationsdatei und den Namen des zu verwendenden Pools zusammen. Das Schema ist `{Pfad zur ceph.conf}:{Pool}`.

Erst dann kann eine Verbindung zum Cluster aufgebaut werden. Des Weiteren wird ein Kontext für die Verbindung gesetzt und somit der Pool festgelegt wird. Im Problemfall wird die Verbindung terminiert und das Programm wirft ein Fehler.

```
rados_create(&backend_connection, NULL);
rados_conf_read_file(backend_connection, backend_config);
rados_connect(backend_connection);
rados_ioctx_create(backend_connection, backend_pool, &backend_io);
```

Jede initialisierte Verbindung muss mit einem `backend_fini()` beendet werden.

```
rados_ioctx_destroy(backend_io);
rados_shutdown(backend_connection);
```

Die restlichen Funktionen sind mehr oder weniger ein reines Mapping von Funktionen, eingebettet in einem Wrapper. Dies betrifft soweit folgende Funktionen:

- `backend_create()` – `rados_write_full(backend_io, full_path, "", 0);`
- `backend_delete()` – `rados_remove(backend_io, bf->path);`
- `backend_status()` – `rados_stat(backend_io, bf->path, size, modification_time);`
- `backend_read()` – `rados_read(backend_io, bf->path, buffer, length, offset);`
- `backend_write()` – `rados_write(backend_io, bf->path, buffer, length, offset);`

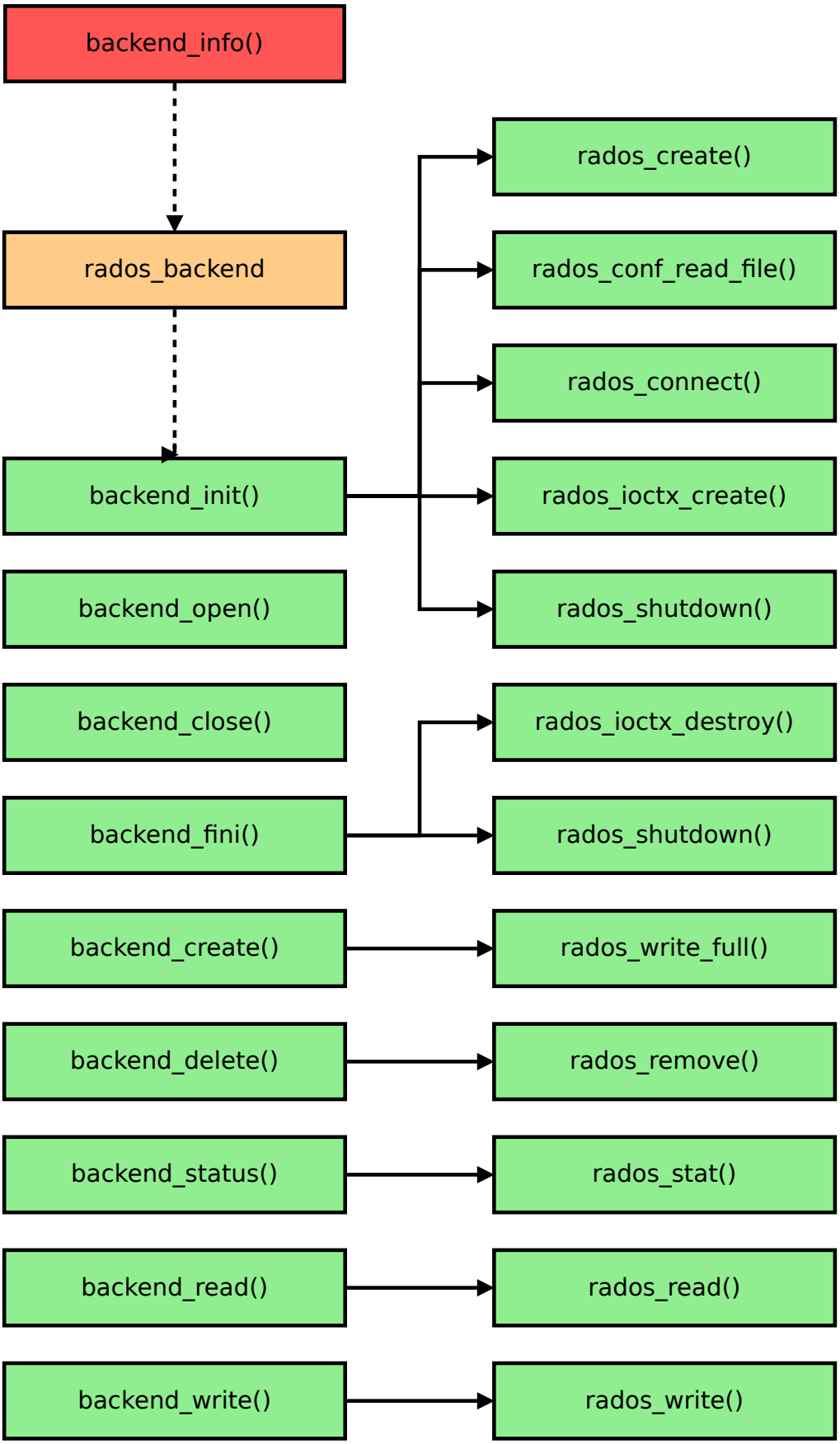


Abbildung 16: Callgraph von `rados.c` (gestrichelte Linien bilden das Mapping als Funktionspointer ab)

5.1.4 Einbau in das Buildsystem

Als Buildsystem wird das in Python geschriebene *WAF* verwendet. Damit *Ceph* als mögliche Komponente von *JULEA* kompiliert werden kann, muss die Datei `wscript` modifiziert werden. Dort stehen alle Kompileranweisungen und Umgebungsvariablen, welche für den Prozess benötigt werden.

Zu Beginn wird folgender ein weiterer Parameter dem Konfigurationsassistenten hinzugefügt, der die Möglichkeit eröffnet, *Ceph* in die Programme einzuschließen.

```
ctx.add_option('--librados', action='store', default=None, help='librados driver
↪  prefix')
```

Weiterhin wird die Umgebungsvariable `JULEA_LIBRADOS` eingeführt, die entsprechende Bibliotheken als Abhängigkeiten hinzufügt.

```
ctx.env.JULEA_LIBRADOS = \
ctx.check_cc(
    lib = 'rados',
    uselib_store = 'LIBRADOS',
    mandatory = False
)
```

Damit *Ceph* auch als Speicherbackend bekannt ist, muss dies dem System noch mitgeteilt werden.

```
if ctx.env.JULEA_LIBRADOS:
    backends_client.append('rados')

for backend in backends_client:
    use_extra = []

    if backend == 'mongodb':
        use_extra = ['LIBMONGOC']
    elif backend == 'rados':
       use_extra = ['LIBRADOS']
```

5.1.5 Kompilieren

Das Programm lässt sich mit dem Buildsystem recht komfortabel erstellen. Sobald die nötigen Abhängigkeiten vorhanden sind, werden diese in die Applikation hinzugefügt.

```
./configure.sh --debug --sanitize
./waf
```

5.1.6 Test

Nach dem Bauen kann der Code mit dem enthaltenen Testskript geprüft werden. Hierfür müssen erst einmal die Umgebungsvariablen in eine Bash Session geladen werden.

```
. ./scripts/environment.sh
```

Als nächstes muss eine Konfiguration für *JULEA* erstellt werden. Dies kann mit Hilfe des Programms `julea-config` erledigt werden. Das dadurch erstellte Dokument befindet sich üblicherweise im `~/.config/julea` Ordner.

Da *JULEA* sowohl einen Objektspeicher für die Dateien, als auch einen KeyValue Speicher benötigt, wurde hier zum Testen *Ceph* mit *SQLite* verwendet.

```
julea-config --user \
    --object-servers="$(hostname)" \
    --object-backend=rados \
    --object-component=client \
    --object-path=/tmp/ceph/ceph.conf \
    --kv-servers="$(hostname)" \
    --kv-backend=sqlite \
    --kv-component=server \
    --kv-path=/tmp/julea
```

Die dadurch erstellte Konfiguration sieht folgendermaßen aus:

```
[clients]
max-connections=0

[servers]
object=localhost;
kv=localhost;

[object]
backend=rados
component=client
path=/tmp/ceph/ceph.conf:data

[kv]
backend=sqlite
component=server
path=/tmp/julea
```

Essentiell ist zudem, dass der Keyring von *Ceph* in der `ceph.conf` korrekt verlinkt ist. Falls es relativ verlinkt werden sollte, muss der Pfad relativ zum ausführenden Programm liegen – nicht relativ zur Konfigurationsdatei! Im Zweifel bietet sich eine absolute Verlinkung an.

```
keyring = /tmp/ceph/ceph.keyring
```

Der Server kann nun via `julea-server` gestartet werden. Nach dem Ausführen des Testskripts `julea-test` sollte die Ausgabe optimalerweise folgendermaßen aussehen:

```
/background_operation/new_ref_unref: OK
/background_operation/wait: OK
/batch/new_free: OK
/batch/semantics: OK
/batch/execute: OK
/batch/execute_async: OK
/cache/new_free: OK
/cache/get: OK
/cache/release: OK
/configuration/new_ref_unref: OK
/configuration/new_for_data: OK
/configuration/get: OK
/distribution/round_robin: OK
/distribution/single_server: OK
/distribution/weighted: OK
/list/new_free: OK
/list/length: OK
/list/append: OK
/list/prepend: OK
/list/get: OK
/list-iterator/new_free: OK
/list-iterator/next_get: OK
/lock/new_free: OK
/lock/acquire_release: OK
/lock/add: OK
/memory-chunk/new_free: OK
/memory-chunk/get: OK
/memory-chunk/reset: OK
/message/new_ref_unref: OK
/message/header: OK
/message/append: OK
/message/write_read: OK
/semantics/new_ref_unref: OK
/semantics/set_get: OK
/item/collection/new_free: OK
/item/collection/name: OK
/item/item/new_free: OK
/item/item/name: OK
/item/item/size: OK
/item/item/modification_time: OK
/item/uri/new_free: OK
```

```
/item/uri/valid: OK
/item/uri/invalid: OK
/item/uri/create: OK
/item/uri/get: OK
```

Zur Evaluation der implementierten Schnittstelle wurde das interne Programm `julea-benchmark` verwendet, welches in mehrmaligen Iterationen jeden IO-Befehl an die angeschlossene Speicherlösung sendet und die benötigte Zeit ermittelt.

Getestet wurde die *RADOS*-Implementation und in Vergleich gestellt mit der *POSIX*-Schnittstelle.

Der *Ceph*-Cluster – bestehend aus einem Monitor und zwei OSDs – befand sich auf einer einzigen Hardware, um einen direkten Vergleich mit *POSIX* zu ermöglichen. Des Weiteren lagen die Metadaten in einer *SQLite*-Datenbank als *Key-Value-Speicher* in einer *RAM-Disk*.

Zur Hardware: Getestet wurde auf einem *AMD*-Knoten mit zwei *AMD Opteron 6168* (12 Kerne, kein Hyperthreading), einer *Seagate Barracuda* (ST3500418AS) und 32GB Arbeitsspeicher. Der Knoten wurde exklusiv genutzt. Performanceeinbußen sind möglich durch die entsprechenden CPU-Bugfixes:

- AMD Erratum 383 (`tlb_mmatch`) [3]
- `FXSAVE` leakt `FOP` , `FIP` und `FOP` (`fxsave_leak`)
- `SYSRET` leakt versteckte `SS` Attribute (`sysret_ss_attrs`)
- Spectre Variante 1 – Angriffe mit bedingten Sprüngen (`spectre_v1`)
- Spectre Variante 2 – Angriffe mit indirekten Sprüngen (`spectre_v2`)

Zur Software: Auf dem Knoten lief ein *Ubuntu 16.04.5 LTS* mit der Kernelversion *4.4.0-133-generic x86_64*. Alle benötigten Bibliotheken wurden mit Hilfe des Paketmanagers *Spack* installiert. Hierfür kam der *GCC* Kompiler in der Version *7.3.0* aus dem *Spack* Repository zum Einsatz.

Sowohl bei der *SQLite*-Datenbank, als auch bei dem Speicherbereich des *POSIX*-Test kam *ext4* als Dateisystem zum Einsatz. Beim *Ceph*-Cluster hingegen wurde *xfs* genutzt, da sowohl *btrfs* als auch der neue, native *BlueStore* noch nicht als stabil eingestuft worden sind [4].

Getestet wurden alle Schritte, welche das Tool `julea-benchmark` unter dem Namensraum `/object/` anbietet. Dabei gibt es sechs grundsätzliche Tests: `create` , `delete` , `status` , `read` , `write` und `unordered-create` . Es wird zudem zwischen `distributed-object` versus `object` und zwischen einzelne versus `batch` -Aufrufe variiert.

Im entsprechenden Quellcode des Benchmarks (`benchmark/object/distributed-object.c` und `benchmark/object/object.c`) kann die genaue Anzahl der Iterationen für jeden Schritt nachvollzogen werden, woraus ein gemittelter Wert berechnet wird. Minimal werden jedoch immer 1000 Wiederholungen eines Testsets durchgeführt.

Um eine Basislinie für den Benchmark zu bekommen, wurde vorab ein Test ohne IO-Load durchgeführt (*null*-Backend). Dadurch ist die maximale Leistung des Benchmarks auf der Hardware ohne IO-Zugriffe ablesbar.

Alle drei Ergebnisse sind visuell in der Abbildung 17 und tabellarisch im folgenden dargestellt. Die Vergleichgröße ist die Anzahl IO-Operationen pro Sekunde *(IOPS)* und die Werte werden auf Grund der starken Streuung logarithmisch notiert.

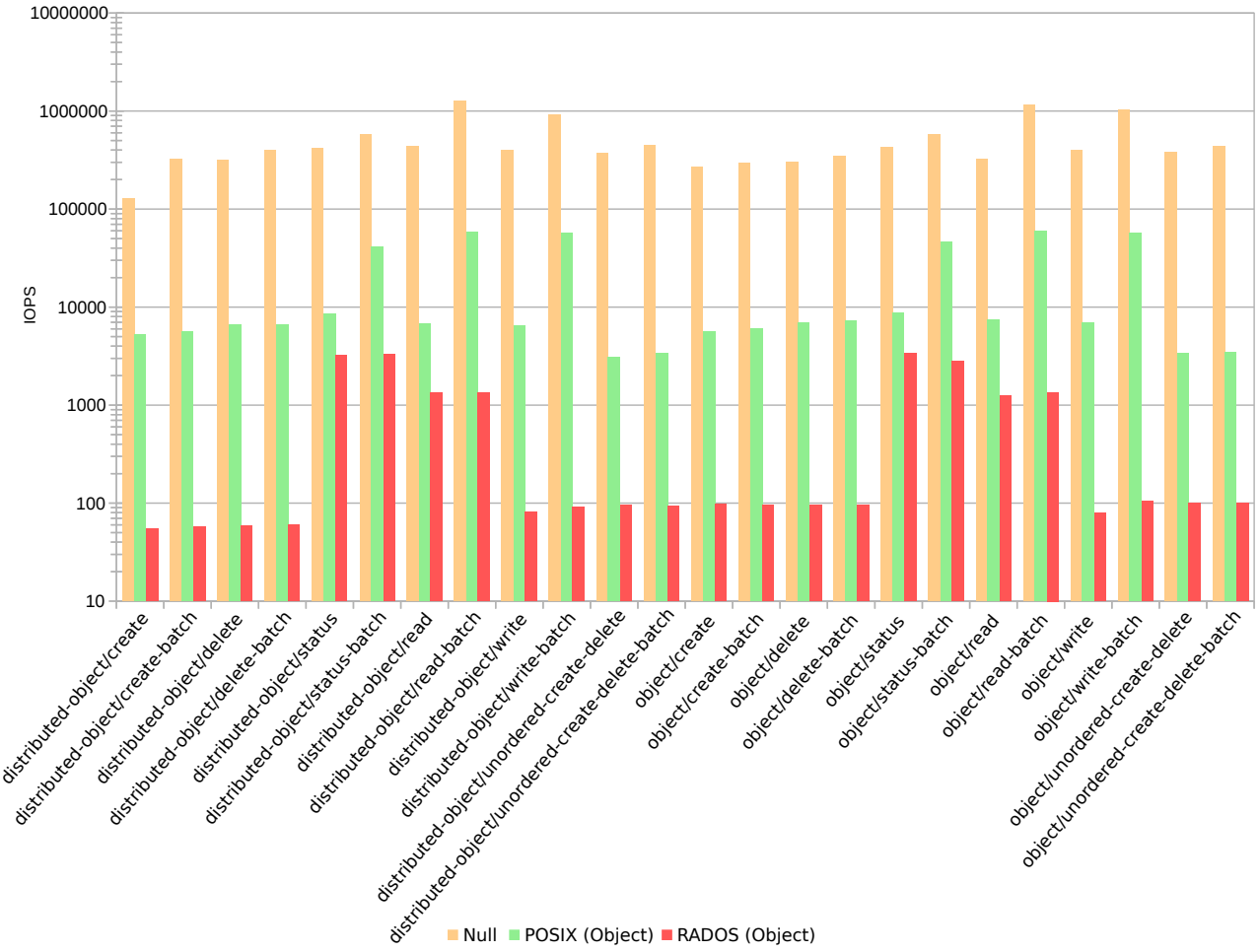


Abbildung 17: Benchmarks zwischen *POSIX* und *RADOS*

Name of NULL tests	Elapsed	Operations	Total elapsed
`distributed-object/create`	0,007904	126518,218623	0,009619
`distributed-object/create-batch`	0,307746	324943,297395	0,488758
`distributed-object/delete`	0,031946	313028,235147	0,066812
`distributed-object/delete-batch`	0,024913	401396,861077	0,051628
`distributed-object/status`	0,002401	416493,127863	0,002519
`distributed-object/status-batch`	0,001731	577700,751011	0,001744
`distributed-object/read`	0,057385	435653,916529	0,085008
`distributed-object/read-batch`	0,019626	1273820,442270	0,044141
`distributed-object/write`	0,062506	399961,603686	0,062639
`distributed-object/write-batch`	0,027251	917397,526696	0,027270
`distributed-object/unordered-create-delete`	0,027062	369521,838741	0,027135

Name of NULL tests	Elapsed	Operations	Total elapsed
distributed-object/unordered-create-delete-batch	0,022532	443813,243387	0,022535
object/create	0,371218	269383,488947	0,543768
object/create-batch	0,338311	295586,014052	0,540358
object/delete	0,332727	300546,694437	0,631174
object/delete-batch	0,285695	350023,626595	0,570590
object/status	0,465720	429442,583527	0,465788
object/status-batch	0,346314	577510,582881	0,346387
object/read	0,006172	324044,069994	0,008822
object/read-batch	0,001733	1154068,090017	0,003706
object/write	0,004997	400240,144086	0,005054
object/write-batch	0,001958	1021450,459653	0,001968
object/unordered-create-delete	0,525903	380298,267932	0,525919
object/unordered-create-delete-batch	0,461002	433837,597234	0,461007

Name of POSIX tests	Elapsed	Operations	Total elapsed
distributed-object/create	0,190546	5248,076580	0,333956
distributed-object/create-batch	17,600810	5681,556701	31,517433
distributed-object/delete	1,519186	6582,472456	3,246155
distributed-object/delete-batch	1,498167	6674,823301	3,158252
distributed-object/status	0,118080	8468,834688	0,119196
distributed-object/status-batch	0,024321	41116,730398	0,024954
distributed-object/read	3,670445	6811,163224	4,108263
distributed-object/read-batch	0,424323	58917,381335	0,887296
distributed-object/write	3,860368	6476,066530	3,886827
distributed-object/write-batch	0,434953	57477,474578	0,468572
distributed-object/unordered-create-delete	1,606294	3112,755199	1,606319
distributed-object/unordered-create-delete-batch	1,481533	3374,882638	1,481553
object/create	17,729183	5640,417835	31,091742
object/create-batch	16,715343	5982,527550	30,120829
object/delete	14,355469	6965,986273	30,900797
object/delete-batch	13,760039	7267,421262	30,216120
object/status	23,093631	8660,396453	23,094688
object/status-batch	4,280419	46724,397775	4,281390
object/read	26,781904	7467,728956	30,445906
object/read-batch	3,359936	59524,943332	6,946889
object/write	28,876584	6926,026984	29,055464
object/write-batch	3,498198	57172,292706	3,702356
object/unordered-create-delete	29,845294	3350,611993	29,845309
object/unordered-create-delete-batch	28,852928	3465,852755	28,852941

Name of RADOS tests	Elapsed	Operations	Total elapsed
`distributed-object/create`	18,042894	55,423481	34,309599
`distributed-object/create-batch`	1740,763832	57,446046	3466,537650
`distributed-object/delete`	170,859983	58,527455	349,757838
`distributed-object/delete-batch`	166,631815	60,012549	335,166921
`distributed-object/status`	0,307710	3249,813136	0,412630
`distributed-object/status-batch`	0,299518	3338,697507	0,366230
`distributed-object/read`	18,523312	1349,650646	406,213694
`distributed-object/read-batch`	18,476285	1353,085861	284,604293
`distributed-object/write`	307,890962	81,197577	307,926163
`distributed-object/write-batch`	274,342416	91,126995	274,420241
`distributed-object/unordered-create-delete`	105,700521	94,606913	105,700626
`distributed-object/unordered-create-delete-batch`	107,795664	92,768110	107,795703
`object/create`	1026,666052	97,402656	2081,478425
`object/create-batch`	1040,519882	96,105804	2092,115530
`object/delete`	1055,622237	94,730858	2089,722324
`object/delete-batch`	1043,094244	95,868615	2074,098597
`object/status`	59,614570	3354,884553	59,659264
`object/status-batch`	71,142728	2811,250083	71,174192
`object/read`	1,586757	1260,432442	21,514124
`object/read-batch`	1,477197	1353,915558	28,504275
`object/write`	25,021628	79,930850	25,065163
`object/write-batch`	19,118640	104,609951	19,170336
`object/unordered-create-delete`	2021,920268	98,915869	2021,920299
`object/unordered-create-delete-batch`	2018,881506	99,064754	2018,881514

Zur Auswertung: Der Unterschied zwischen den `distributed-object` und `object` Operationen ist nicht signifikant. Sowohl bei der Basismessung mit *null*, als auch bei den beiden Speicherlösungen. Bei *POSIX* ist erkennbar, dass die `status`, `read` und `write` Operationen im *Batchmodus* deutlich effizienter sind, als ihr Pendant. *RADOS* ist im Vergleich zu *POSIX* in der Messung merklich langsamer. Abgesehen von `status` und `read` Operationen ist das Backend um den Faktor 10^3 langsamer. Die beiden genannten Operationen dagegen *nur* um den Faktor 2-3.

In absoluten Zahlen kommt das *POSIX* Backend auf einen Median von rund 6869 IOPS, *RADOS* nur auf 97 IOPS.

Der Grund für die erheblichen Performanceunterschiede könnte in den unterschiedlichen Einsatzgebieten liegen. *POSIX* bietet Schnittstellen für lokale Speichermedien an und ist nicht geeignet für einen verteilten Zugriff – abgesehen von der fehlenden Netzwerkfähigkeit. *RADOS* ist eine Lösung für verteiltes Speichern, somit muss immer mit einem Netzwerkoverhead gerechnet werden. Weiterhin lässt die Konfiguration von *RADOS* mit zwei Speichernodes auf einer physikalischen Maschine und nicht optimal eingestellter Anzahl an *Placement Groups* recht viel Potential offen.

5.2 Mercury als Netzwerkframework

Um *Mercury* als Framework für den Netzwerkverkehr benutzen zu können, muss sich für mindestens eine Erweiterung entschieden werden. Für dieses Projekt wurde *CCI* ausgewählt, da es gegenüber den anderen mehrere Vorzüge bietet:

1. Die Implementation des Plugins benötigt laut den Beschreibungen der Entwickler im Idle sehr geringe CPU Zeiten (außer bei *TCP*-Verbindungen), da sie kein sogenanntes *Busy Spinning* betreiben. Dies bedeutet, dass kein Konstrukt existiert, welches dem folgenden Prinzip entspricht:

```
while(1)
{
    if(is_data_received())
    {
        process_data();
    }
    else
    {
        sleep(1);
    }
}
```

2. Für die sogenannten *Bulk* Operationen, also die Übertragung von größeren Datenmengen, wird *Remote Memory Access (RMA)* verwendet, wodurch die Daten effizient direkt in den Zielpuffer geschrieben werden können.

3. Es werden recht viele Transportprotokolle unterstützt, deren Implementation im Vergleich zu den anderen stabiler und performanter seien. Die Unterstützung reicht von *Verbs* (*RDMA* Protokol), über *Shared Memory*, *TCP* bis zu *GNI* (*Generic Network Interface* von *Cray*).

5.2.1 Kompilieren von CCI

Bevor *Mercury* kompiliert werden kann, werden die Bibliotheken und Headerdateien zu der gewünschten Erweiterung benötigt. Dies ist in diesem Fall *CCI*, welches selber gebaut werden muss, da es nicht in den Repositories der gängigen Linuxdistributionen zu finden ist. Dafür ist die aktuelle Version von der Webseite des Projektes zu beziehen: `http://cci-forum.com`. Für dieses Projekt wurde mit der zu dem Zeitpunkt aktuellen Version 2.1 gearbeitet.

Um die Wartezeiten bei I/O-Operationen zu verkürzen, wurden alle folgenden Komponenten in einer RAMDisk kompiliert:

```
curl -o cci-2.1.tar.gz
↪  "http://cci-forum.com/wp-content/uploads/2017/05/cci-2.1.tar.gz"
tar xf cci-2.1.tar.gz
cd cci-2.1/
```

CCI baut weitestgehend auf Standardbibliotheken auf und verwendet zum Bauen einen Workflow bestehend aus `./configure` Skripten und *Makefiles*.

```
./configure --prefix=/tmp/cci-2.1-build/
make -j 4 //Anzahl Kerne
make install
```

Nach Ausführung genannter Schritte befinden sich die benötigten Headerdateien und Bibliotheken nun unter `/tmp/cci-2.1-build/`

5.2.2 Kompilieren von Mercury

Das Bauen von *Mercury* verläuft etwas anders. Zu Beginn sollte das Projekt aus dem *GitHub*-Repository geklont werden, um den aktuellen Stand zu beziehen. Es sei anzumerken, dass das Releasebundle in der Version 0.9 nicht empfohlen werden kann, da dieses unvollständig gepackt worden ist (es fehlen beispielsweise essentielle Bestandteile des Beispielcodes).

```
git clone git@github.com:mercury-hpc/mercury.git
cd mercury/
```

Als Buildsystem wurde *Cmake* verwendet. Zusätzlich gibt es den Konfigurationsassistenten `ccmake`, welchen man mit diesem System verwenden muss.

```
mkdir build/
cd build/
ccmake ../
```

Bei der folgenden Konfiguration ist zu beachten, dass für die Unterstützung von *CCI* die zugehörigen Pfade extra noch angegeben werden müssen, falls sie nicht in den Systempfaden installiert wurden. Des Weiteren wird für die Implementierung *Boost* benötigt, da die *Mercury High Level API* mit den speziellen Makros nur von *Boost* prozessiert werden können.

```
BUILD_DOCUMENTATION              OFF
BUILD_EXAMPLES                   OFF
```

```
BUILD_SHARED_LIBS                ON
BUILD_TESTING                    OFF
CCI_INCLUDE_DIR                  /tmp/cci-2.1-build/include
CCI_LIBRARY                      /tmp/cci-2.1-build/lib/libcci.so
CMAKE_ARCHIVE_OUTPUT_DIRECTORY   /tmp/mercury/build/bin
CMAKE_BUILD_TYPE                 RelWithDebInfo
CMAKE_INSTALL_PREFIX             /usr/local
CMAKE_LIBRARY_OUTPUT_DIRECTORY   /tmp/mercury/build/bin
CMAKE_RUNTIME_OUTPUT_DIRECTORY   /tmp/mercury/build/bin
MERCURY_ENABLE_COVERAGE          OFF
MERCURY_ENABLE_STATS             OFF
MERCURY_ENABLE_VERBOSE_ERROR     ON
MERCURY_USE_BOOST_PP             ON
MERCURY_USE_CHECKSUMS            OFF
MERCURY_USE_EAGER_BULK           ON
MERCURY_USE_OPA                  OFF
MERCURY_USE_SELF_FORWARD         OFF
MERCURY_USE_SM_ROUTING           OFF
MERCURY_USE_SYSTEM_MCHECKSUM     OFF
MERCURY_USE_XDR                  OFF
NA_USE_BMI                       OFF
NA_USE_CCI                       ON
NA_USE_MPI                       OFF
NA_USE_OFI                       OFF
NA_USE_SM                        OFF
```

Einige Konfigurationen tauchen erst nach dem Aktivieren anderer Optionen auf bzw. benötigen eine Konfliktbehebung. Es muss mit der Taste `c` so lange die Konfiguration getestet werden, bis keine Konflikte mehr auftreten. Sobald dies der Fall ist, kann mit der Taste `g` eine *Cmake* Konfiguration angelegt werden.

```
make -j 4 //Anzahl Kerne
```

Die benötigten Dateien liegen nun sowohl in `/tmp/mercury/`, als auch in `/tmp/mercury/build/` und müssen von den jeweiligen Kompilern, wie beispielsweise *GCC*, inkludiert werden.

5.2.3 Design

Das Projekt *JULEA* baut bis dato auf der *GLib*-Implementation von TCPStreams auf. Dabei gibt es mehrere Komponenten:

1. Verbindungspool
   Es existiert ein Pool an offenen, initialisierten Client-zu-Server-Verbindungen. Um Daten an einen der *JULEA*-Server zu versenden, wird eine von diesen Verbindungen aus dem Pool herausgenommen. Implementiert ist dies durch eine einfache `pop()` Funktion.

Sobald die Transferoperationen zum Ende geführt worden sind, wird diese Verbindung wieder zur Verfügung gestellt, indem sie via `push()` wieder zu dem Pool hinzugefügt wird.
Dieser Ablauf kann mit *Mercury* ebenfalls genutzt werden. Ein Kontexthandler, welcher in etwa einer initialisierten Verbindung entspricht, könnte in einem Pool gesammelt und bei Bedarf abgerufen werden.

2. Nachrichten
   Die Datentransfers werden in einem `struct` relativ gut abstrahiert, sodass diese auch für *Mercury* genutzt werden könnten. Eine Nachricht hat keine feste Größe, da die Daten referenziert werden, enthält aber alle mögliche Steuerinformationen. Dementsprechend müsste diese nicht neu implementiert werden.

Für die Initialisierung des *Bulk Transfers*, welches *RMA* verwendet, muss vorab die exakt nötige Puffergröße bekannt sein. Die Referenzimplementation für *JULEA* sieht vor, dass diese Größe einfach mit dem *RPC* initial übertragen wird – dies ist ein `int32` und als Beispiel wird `4096` übertragen, welches in *Byte* interpretiert wird. Danach startet der *Bulk Transfer* und es werden entweder vom Client oder zum Client Daten übertragen. In Abbildung 18 ist der vollständige Ablauf der Kommunikation abgebildet.

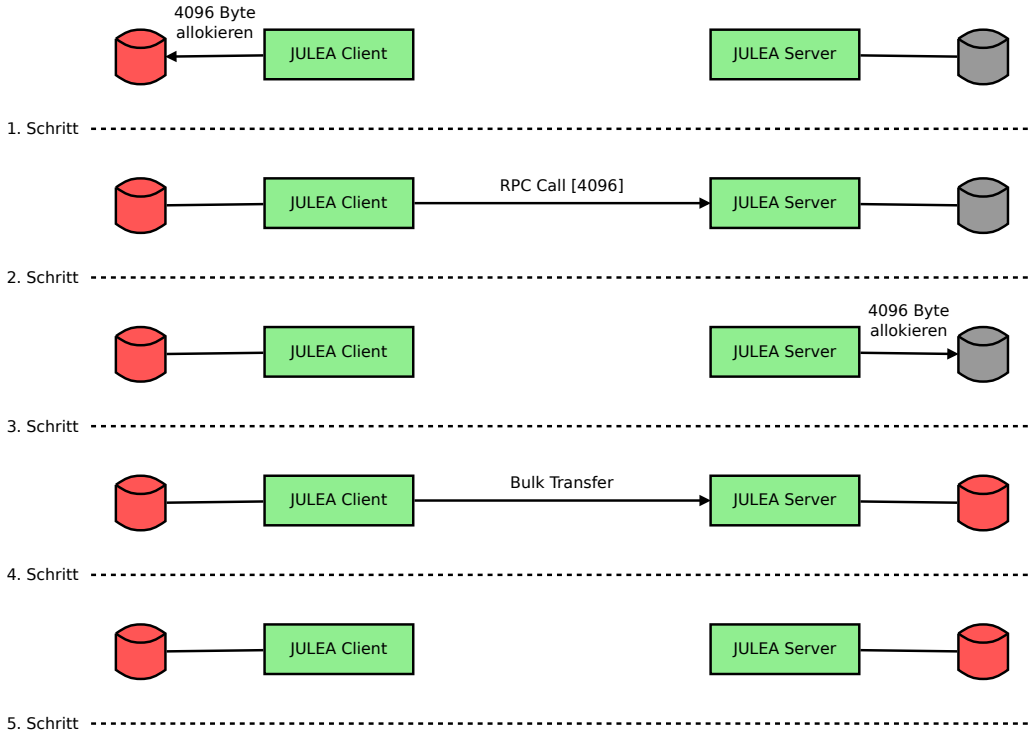


Abbildung 18: Kommunikationsschritte der Referenzimplementation via Mercury (Grundidee aus [6, Abbildung 1])

5.2.4 Implementierung

Die Referenzimplementation wurde unter Zuhilfenahme des im Projekt enthaltenen Beispielcodes erstellt, da eine sinnvolle Dokumentation seitens *Mercury* zu dem Zeitpunkt leider nicht vorlag – was eine Umsetzung in der zur Verfügung stehenden Zeitspanne schwierig machte.

Sie besteht sowohl aus einer Client- als auch aus einer Serverkomponente, schickt die Puffergröße via *RPC* und initiiert einen *Bulk Transfer* von einem `string` mit einer Zufallszahl, um nachzuweisen, dass es sich um einen tatsächlichen, einmaligen Datentransfer handelt.

Der erstellte Code wurde vollständig nach dem Codingschema von *JULEA* erstellt. Im folgenden werden die verwendeten Projektdateien aufgelistet und beschrieben:

- cci.ini
  Aufgrund eines Fehlers in der Implementation von CCI in Mercury kann der gewünschte Hostname bzw. die gewünschte IP-Adresse nicht in den Initialisierungsfunktionen angegeben werden. Deshalb wird diese externe Konfigurationsdatei benötigt, die zur Laufzeit des Programms via Umgebungsvariablen eingebunden sein muss.

- server.c
  Eine einfache Implementation eines Servers, der einen *RPC* registriert und einfach nur zuhört und wartet.

- client.c
  Der Client baut eine Verbindung auf und führt 20-mal eine Netzwerkoperation durch, wobei jedes Mal ein *Bulk Transfer* mit einem `string` und einer Zufallszahl durchgeführt wird.

- j_mercury.c, j_mercury.h
  Alle benötigten Funktionen, von der Initialisierung einer Verbindung, über der Registrierung des *RPCs*, bis hin zu den Callback Funktionen der asynchronen Übertragung liegen in diesen Dateien.

In der folgender Konfigurationsdatei **cci.ini** wird eine Verbindung via *TCP* zum Server aufgebaut, der auf dem gleichen Rechner läuft.

```
1 [lo]
2 transport = tcp
3 ip = 127.0.0.1
4 default = 1
```

Diese Datei muss mit Hilfe der Umgebungsvariable `CCI_CONFIG` zur Laufzeit verlinkt werden. Der Befehl, um den korrekten Pfad zu setzen, lautet mit Bash `CCI_CONFIG=cci.ini` oder mit Fish `set -x CCI_CONFIG cci.ini`, um den korrekten Pfad zu setzen. Vorsicht bei relativen Angaben: Sie müssen relativ zum ausführenden Programm liegen!

Die Serverkomponente liegt in der Datei `server.c` und ist recht trivial aufgebaut. Der entsprechende Callgraph ist in Abbildung 19 zu sehen. Zu Beginn wird eine Verbindung mit Hilfe des Frameworks hergestellt (folgender Codeblock, Zeile 3-4).

Der Verbindungsparameter, das erste Argument von `j_mercury_init()`, gibt sowohl das zu verwendende Plugin, das Protokoll, die Adresse auf der gelauscht werden soll als auch den entsprechenden Port an. Das Schema sieht folgendermaßen aus: `{Plugin}+{Protokoll}://{Hostname/IP}:{Port}`. Das zweite Argument gibt an, ob diese Verbindung an die angegebene Adresse und Port lauschend binden, oder zum Senden verwendet werden soll. Mit Hilfe dieser Verbindung wird der sogenannte *RPC* registriert (folgender Codeblock, Zeile 5).

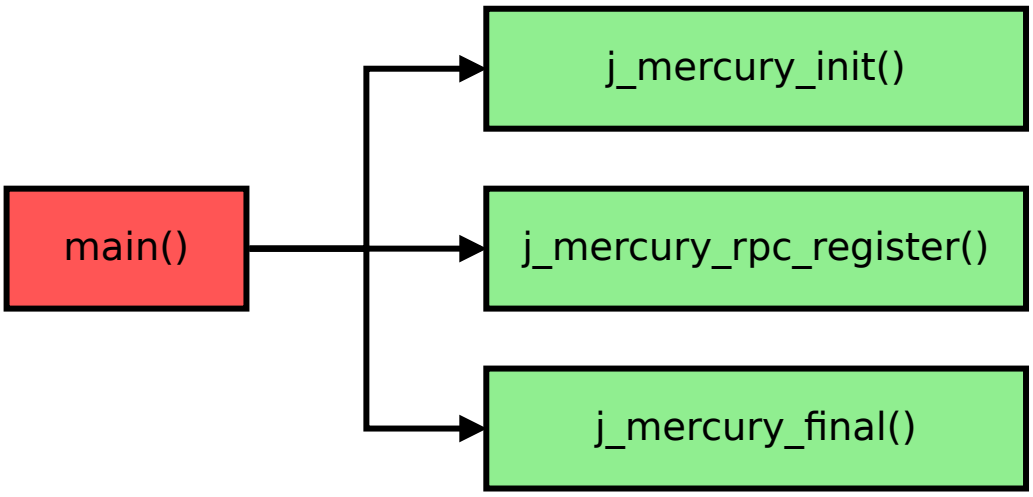


Abbildung 19: Callgraph von `server.c`

Da *Mercury* asynchron mit Callbacks arbeitet, muss der Server im Grunde nichts machen, außer die Terminierung zu verhindern, was mit Hilfe einer einfachen Schleife realisiert worden ist (folgender Codeblock, Zeile 7-10). Diese sollte jedoch für spätere Implementationen ausgebaut und ein Signalhandler hinzugefügt werden, damit bei gewünschter Termination die Verbindung finalisiert werden kann.

```
1  int main(void)
2  {
3      JMercuryConnection* connection;
4      connection = j_mercury_init("cci+tcp://127.0.0.1:4242", TRUE);
5      j_mercury_rpc_register(connection);
6
7      while(1)
8      {
9          sleep(1);
10     }
11
12     j_mercury_final(connection);
13     return(0);
14 }
```

Der Client befindet sich in `client.c` und stellt n-Anfragen an den Server. Diese Anfragen beginnen mit einem *RPC*, welche die Größe des Puffers für den folgenden *Bulk Transfer* enthält, beispielsweise `4096` (Byte). Der entsprechende Callgraph ist in Abbildung 20 zu sehen. Daraufhin wird der Transfer gestartet und ein String mit einer Zufallszahl als Proof-of-Concept an den Server übertragen. Der Client terminiert, wenn alle Anfragen, welche parallel mit Hilfe von `pthreads` abgeschickt werden, abgeschlossen worden sind. Die Anzahl an Anfragen kann über die Variable `number_of_rpcs` festgelegt werden.

Auch beim Client muss eine Verbindung initialisiert werden. Der Unterschied jedoch liegt im Verbindungsparameter: es müssen weder Hostname noch IP-Adresse oder Port angegeben wer-

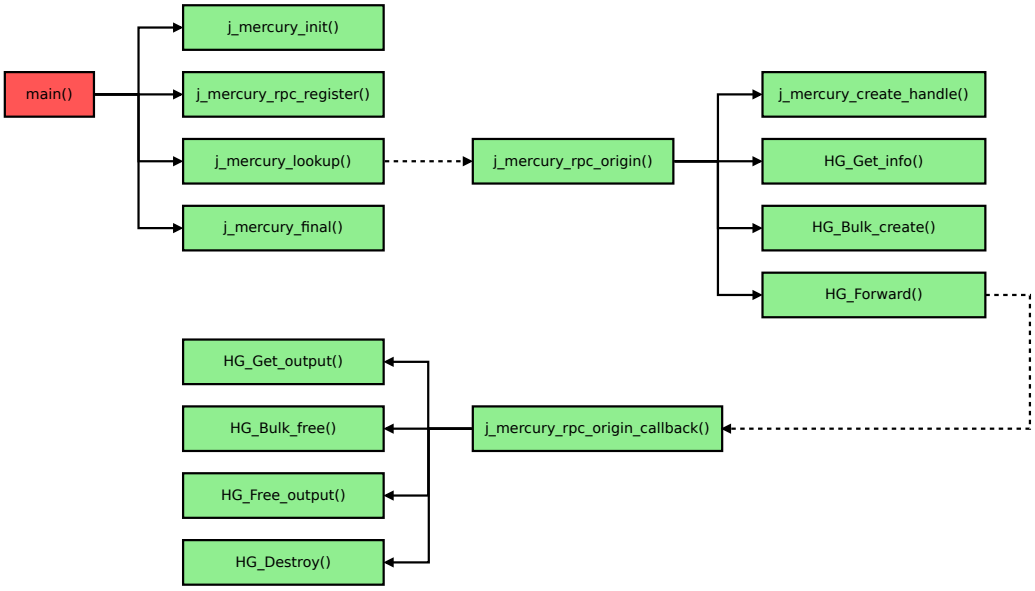


Abbildung 20: Callgraph von `client.c` (gestrichelte Linien bilden einen Aufruf via Callback ab)

den. Die RPC ID wird später für den Handler noch benötigt (folgender Codeblock, Zeile 3-4).

Die n-Anfragen werden asynchron in einer Schleife hintereinander gestartet (folgender Codeblock, Zeile 6-13). Dafür muss die exakte Adresse des Servers mit Hilfe des Verbindungsparameters ermittelt werden. Außerdem wird die Payload des *RPCs* angehängt (folgender Codeblock, Zeile 11).

Sobald alle Anfragen terminiert sind und der Mutex entsperrt werden kann (folgender Codeblock, Zeile 14-19), kann die Netzwerkverbindung geschlossen werden (folgender Codeblock, Zeile 21).

```
1  int main(void)
2  {
3      connection = j_mercury_init("cci+tcp", FALSE);
4      rpc_id     = j_mercury_rpc_register(connection);
5
6      for(gint i = 0; i < number_of_rpcs; i++)
7      {
8          gint* rpc_value = g_malloc(sizeof(*rpc_value));
9          *rpc_value = 4096;
10
11         j_mercury_lookup(connection, "cci+tcp://127.0.0.1:4242",
           ↪  j_mercury_rpc_origin, rpc_value);
12     }
13
14     pthread_mutex_lock(&thread_mutex);
15     while(thread_counter < number_of_rpcs)
16     {
17         pthread_cond_wait(&thread_condition, &thread_mutex);
18     }
19     pthread_mutex_unlock(&thread_mutex);
```

```
20
21      j_mercury_final(connection);
22      return(0);
23  }
```

Die Callback Funktion `j_mercury_rpc_origin()`, welche in der vorhin erwähnte *Lookup-Funktion* (vorheriger Codeblock, Zeile 12) verwendet worden ist, initialisiert einen *Bulk Transfer* – die Puffergröße wurde vorher bereits übermittelt. Es muss weiterhin ein RPC Handler erstellt und angemeldet werden (folgender Codeblock, Zeile 15).

Darauffolgend wird der *Bulk Transfer* angemeldet (folgender Codeblock, Zeile 17). Erst danach können die initialisierten Puffer und Attribute (folgender Codeblock, Zeile 7-11) übermittelt werden (folgender Codeblock, Zeile 21).

```
1   static hg_return_t j_mercury_rpc_origin(const struct hg_cb_info* info)
2   {
3       JMercuryRPCState* rpc_state;
4       JMercuryRPCIn in;
5       const struct hg_info* handle_info;
6
7       rpc_state         = g_malloc(sizeof(*rpc_state));
8       rpc_state->size   = *((gint*) info->arg);
9       rpc_state->value  = *((gint*) info->arg);
10      rpc_state->buffer = g_malloc0(rpc_state->size * sizeof(gchar));
11      g_sprintf((gchar*) rpc_state->buffer, "Lorem ipsum dolor sit amet! %d\n",
        ↪  g_rand_int(g_rand_new()));
12
13      g_free(info->arg);
14
15      j_mercury_create_handle(connection, info->info.lookup.addr, rpc_id,
        ↪  &rpc_state->handle);
16      handle_info = HG_Get_info(rpc_state->handle);
17      HG_Bulk_create(handle_info->hg_class, 1, &rpc_state->buffer, &rpc_state->size,
        ↪  HG_BULK_READ_ONLY, &in.bulk_handle);
18      rpc_state->bulk_handle = in.bulk_handle;
19
20      in.input_val = rpc_state->value;
21      HG_Forward(rpc_state->handle, j_mercury_rpc_origin_callback, rpc_state, &in);
22
23      return(NA_SUCCESS);
24  }
```

Sobald der Transfer abgeschlossen worden ist, wird wiederum eine Callback-Funktion aufgerufen (vorheriger Codeblock, Zeile 21). In diesem Fall handelt es sich um `j_mercury_rpc_origin_callback()`. Die Antwort des Servers kann nun ausgelesen werden, um eventuell eine Fehlerbehandlung durchzuführen (folgender Codeblock, Zeile 6-7).

Hauptsächlich werden jedoch nur die Verbindungen und Handler geschlossen (folgender Codeblock, Zeile 9-11) und die Sperren von `pthread` inkrementiert (folgender Codeblock, Zeile 15-18), damit das Programm irgendwann terminieren kann.

```
1  static hg_return_t j_mercury_rpc_origin_callback(const struct hg_cb_info* info)
2  {
3      JMercuryRPCOut out;
4      JMercuryRPCState* rpc_state = info->arg;
5
6      HG_Get_output(info->info.forward.handle, &out);
7      g_printf("[ RPC  ]  Got response: %d\n", out.ret);
8
9      HG_Bulk_free(rpc_state->bulk_handle);
10     HG_Free_output(info->info.forward.handle, &out);
11     HG_Destroy(info->info.forward.handle);
12     g_free(rpc_state->buffer);
13     g_free(rpc_state);
14
15     pthread_mutex_lock(&thread_mutex);
16     thread_counter++;
17     pthread_cond_signal(&thread_condition);
18     pthread_mutex_unlock(&thread_mutex);
19
20     return(HG_SUCCESS);
21 }
```

In `j_mercury.c` befinden sich alle notwendigen Hilfsfunktionen, um das Framework erfolgreich zu verwenden. Dabei handelt es sich meistens um Initialisierungs- und Finalisierungsfunktionen, aber auch kleine Wrapper um die *Mercury-API*. Der entsprechende Callgraph ist in Abbildung 21 zu sehen.

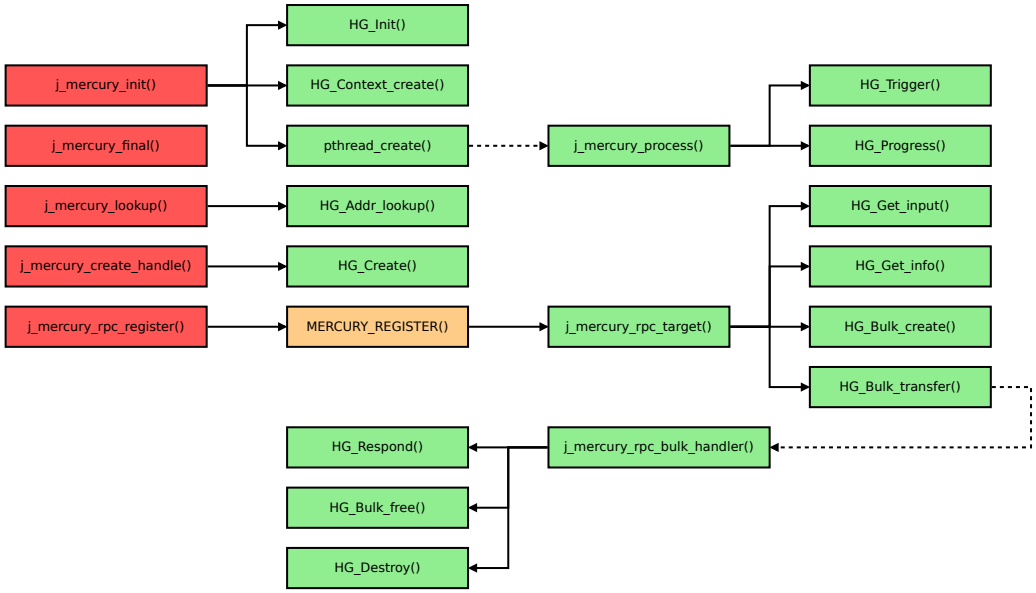


Abbildung 21: Callgraph von `j_mercury.c` (gestrichelte Linien bilden einen Aufruf via Callback ab)

Die erste Funktion `j_mercury_init()` initialisiert eine neue *Mercury*-Verbindung und erstellt einen neuen Kontext (folgender Codeblock, Zeile 3-6). Daraufhin wird ein Thread gestartet, welcher `j_mercury_process()` mit der neuen Verbindung als Payload aufruft (folgender Codeblock, Zeile 8).

```
1  JMercuryConnection* j_mercury_init (const char* local_addr, gboolean listen)
2  {
3      JMercuryConnection* connection = g_slice_new(JMercuryConnection);
4      connection->shutdown_flag      = FALSE;
5      connection->class              = HG_Init(local_addr, (na_bool_t) listen);
6      connection->context            = HG_Context_create(connection->class);
7
8      pthread_create(&connection->thread_id, NULL, j_mercury_process, connection);
9
10     return connection;
11 }
```

Jede offene Verbindung muss am Ende geschlossen werden. Dadurch wird under Anderem vermieden, dass zu viele Netzwerkports blockiert werden.

```
1  void j_mercury_final (JMercuryConnection* connection)
2  {
3      connection->shutdown_flag = TRUE;
4      pthread_join(connection->thread_id, NULL);
5      g_slice_free(JMercuryConnection, connection);
6      return;
7  }
```

Um den Server mittels eines Verbindungsparameters zu ermitteln, gibt es diese Lookup-Funktion. Durch die Asynchronität muss immer eine Callback Funktion im Parameter `callback` mitgegeben werden. Genutzt wird sie hauptsächlich in der Hauptfunktion des Clients.

```
1  void j_mercury_lookup (JMercuryConnection* connection, const char* name, hg_cb_t
   ↪  callback, void* arg)
2  {
3      HG_Addr_lookup(connection->context, callback, arg, name, HG_OP_ID_IGNORE);
4      return;
5  }
```

Zum Erstellen eines *RPC* Handlers gibt es die folgende Wrapperfunktion. Sie wird für die Serverkomponente `j_mercury_rpc_origin()` benötigt.

```
1  void j_mercury_create_handle (JMercuryConnection* connection, na_addr_t address,
   ↪  hg_id_t rpc_id, hg_handle_t* handle)
2  {
3      HG_Create(connection->context, address, rpc_id, handle);
```

```
4       return;
5   }
```

`j_mercury_process()` ist eine Callback Funktion, welche in `j_mercury_init()` aufgerufen und in einem eigenen Thread gestartet worden ist. Diese Funktion gibt erst `NULL` zurück, sobald die Verbindung terminiert worden ist. Somit ist demnach auch der Thread beendet, da die Schleife (folgender Codeblock, Zeile 6-18) durch den veränderten *Shutdown-Flag* unterbrochen wird.

```
1   static void* j_mercury_process (void* connection)
2   {
3       hg_return_t ret;
4       guint count;
5
6       while(!((JMercuryConnection*) connection)->shutdown_flag)
7       {
8           do
9           {
10              ret = HG_Trigger(((JMercuryConnection*) connection)->context, 0, 1,
                ↪  &count);
11          }
12          while((ret == HG_SUCCESS) && count && !((JMercuryConnection*)
            ↪  connection)->shutdown_flag);
13
14          if(!((JMercuryConnection*) connection)->shutdown_flag)
15          {
16              HG_Progress(((JMercuryConnection*) connection)->context, 100);
17          }
18      }
19
20      return (NULL);
21  }
```

Um einen *RPC* senden zu können, muss dieser vorher registriert werden. Sowohl die Client- als auch Serverkomponente benötigen diese Funktionalität zu Beginn der Initialisierung. Es wird auf die High-Level API-Schnittstelle von *Mercury* zurückgegriffen. Diese sieht ein Aufruf des Makros `MERCURY_REGISTER()` vor, welches mit Hilfe des *Boost Preprozessors* umgewandelt wird.

```
1   hg_id_t j_mercury_rpc_register (JMercuryConnection* connection)
2   {
3       return MERCURY_REGISTER(connection->class, "j_mercury_rpc", JMercuryRPCIn,
        ↪  JMercuryRPCOut, j_mercury_rpc_target);
4   }
```

Die gerade beschriebene Funktion ruft in ihrem Callback (vorheriger Codeblock, Zeile 3) die folgende Funktion auf. Dort werden die Handler für die Transfers aufgerufen oder erstellt.

Sobald die lokalen Handels erstellt worden sind (folgender Codeblock, Zeile 6-7) kann die *RPC*-Eingabe dekodiert werden (folgender Codeblock, Zeile 8). Dieser gibt in dieser Implementation die Größe des Puffers für den *Bulk-Transfer* vor (folgender Codeblock, Zeile 9).

Ist der Puffer allokiert (folgender Codeblock, Zeile 13) und registriert (folgender Codeblock, Zeile 14) kann der lokale Handle global annonciert werden (folgender Codeblock, Zeile 16). Der *Bulk-Transfer* startet daraufhin und die Inhalte werden übermittelt (folgender Codeblock, Zeile 17).

```
1  static hg_return_t j_mercury_rpc_target (hg_handle_t handle)
2  {
3      struct JMercuryRPCState* rpc_state;
4      const struct hg_info* handle_info;
5
6      rpc_state         = g_malloc0(sizeof(*rpc_state));
7      rpc_state->handle = handle;
8      HG_Get_input(handle, &rpc_state->in);
9      rpc_state->size   = rpc_state->in.input_val;
10
11     g_printf("[ RPC  ]  Bulk transfer size: %d\n", rpc_state->size);
12
13     rpc_state->buffer = g_malloc0(rpc_state->size * sizeof(gchar));
14     handle_info = HG_Get_info(handle);
15
16     HG_Bulk_create(handle_info->hg_class, 1, &rpc_state->buffer, &rpc_state->size,
       ↪  HG_BULK_WRITE_ONLY, &rpc_state->bulk_handle);
17     HG_Bulk_transfer(handle_info->context, j_mercury_rpc_bulk_handler, rpc_state,
       ↪  HG_BULK_PULL, handle_info->addr, rpc_state->in.bulk_handle, 0,
       ↪  rpc_state->bulk_handle, 0, rpc_state->size, HG_OP_ID_IGNORE);
18
19     return 0;
20 }
```

Für den Transfer wird ebenfalls eine Callback Funktion angegeben, welche im folgenden beschrieben wird. Sie sendet ein `ACK` zum Client (folgender Codeblock, Zeile 9), um die erfolgreiche Übertragung zu quittieren und räumt daraufhin den Speicher wieder frei (folgender Codeblock, Zeile 11-14).

```
1  static hg_return_t j_mercury_rpc_bulk_handler (const struct hg_cb_info* info)
2  {
3      struct JMercuryRPCState *rpc_state = info->arg;
4      JMercuryRPCOut rpc_out;
5      rpc_out.ret = 42;
6
7      g_printf("[ DATA ]  %s", rpc_state->buffer);
8
```

```
    HG_Respond(rpc_state->handle, NULL, NULL, &rpc_out);

    HG_Bulk_free(rpc_state->bulk_handle);
    HG_Destroy(rpc_state->handle);
    g_free(rpc_state->buffer);
    g_free(rpc_state);

    return 0;
}
```

5.2.5 Kompilieren

Das Programm benötigt jeweils sowohl von *Mercury*, als auch von *CCI* eine Shared Library und die entsprechenden Headerdateien. Diese Pfade müssen dem Compiler mitgeteilt werden, falls diese nicht in dem Systempfad liegen. In den vorherigen Abschnitten wurden diese in der RAMDisk unter `/tmp/` erstellt.

Ebenfalls wird *Boost*, die *GLib* und ein aktuelles GCC benötigt. Der Client wird äquivalent zum Server kompiliert. Es müssen nur der Parameter `-o server` und die Datei `server.c` entsprechend ersetzt werden.

```
gcc -o server \
    j_mercury.c server.c \
    -I/tmp/mercury/src \
    -I/tmp/mercury/build/src \
    -I/tmp/mercury/src/na \
    -I/tmp/mercury/build/src/na \
    -I/tmp/mercury/src/util \
    -I/tmp/mercury/build/src/util \
    -L/tmp/mercury/build/bin \
    -lmercury -lmercury_util \
    -L/tmp/cci-2.1-build/lib/ \
    -lcci -lpthread -lrt \
    -I/usr/include/glib-2.0 \
    -I/usr/lib64/glib-2.0/include \
    -lglib-2.0
```

5.2.6 Starten der Referenzimplementation

Vor dem Start der beiden Komponenten müssen einige Umgebungsvariablen gesetzt werden. Wie weiter oben beschrieben, muss `CCI_CONFIG` gesetzt werden. Außerdem muss der Pfad zu der Shared Library von *Mercury* und *CCI* gelinkt werden. Dies funktioniert je nach Shell normalerweise mit der Variable `LD_LIBRARY_PATH`.

Unter Bash sehen die Befehle folgendermaßen aus:

```
CCI_CONFIG=cci.ini
LD_LIBRARY_PATH=/tmp/cci-2.1-build/lib/:/tmp/mercury/build/bin/
```

Oder wahlweise bei Fish:

```
set -x CCI_CONFIG cci.ini
set -x LD_LIBRARY_PATH /tmp/cci-2.1-build/lib/:/tmp/mercury/build/bin/
```

Es bietet sich an, den Server zuerst zu starten. Dieser bedarf wie der Client auch keiner weiteren Parameter. Ansonsten meldet der Client einen Fehler und stürzt ab:

```
cci:2176:tcp_conn_set_closing_locked: tconn->queued has 1
fish: ‘‘./client’’ terminated by signal SIGSEGV (Address boundary error)
```

Die typische Ausgabe des Servers wäre die Mitteilung der gewählten Puffergröße und des übertragenen Inhalts. Die Nachricht besteht aus einem klassischen `Lorem ipsum`, gepaart mit einer Zufallszahl.

```
[ RPC  ]  Bulk transfer size: 4096
[ DATA ]  Lorem ipsum dolor sit amet! 264785317
```

Der Client gibt entsprechend den vom Server zurückgegebenen Code aus. Im Beispiel handelt es sich hierbei natürlich um eine `42`.

```
[ RPC  ]  Got response: 42
```

6 Future Work

Die Integration von *Mercury* in *JULEA* gestaltete sich, wie im vorigen Abschnitt bereits angedeutet, leider schwieriger als gedacht, weshalb zum Schluss eine an *JULEA* angepasste Referenzimplementation als Ergebnis herauskam.

Um *Mercury* gut zu implementieren, müsste *JULEA* in der Hinsicht umbaut werden, dass es mit asynchronen Handles arbeiten kann. Ansonsten würde mit sogenanntem *Busy Spinning* nur dafür gesorgt werden, dass der Vorteil von Asynchronizität nicht mehr vorhanden ist.

Mit dem *Bulk Transfer* können recht große Datenmengen in einem Durchlauf übertragen werden. Das bedeutet, die *JMessages* sollten im besten Fall nicht mehr mit einer `send_list` arbeiten, sondern die Nachrichten als großen Transfer direkt über das Netzwerk verschicken. Was es ebenfalls zu testen gilt, ist die Möglichkeit einen Rückkanal via *Bulk Transfer* zu initiieren – also quasi einen zweiten Transfer zu starten, welcher in die andere Richtung schreibend wirkt.

Zu guter Letzt müssten überall die `GSocket` Verbindungen durch ein abstraktes Konstrukt ersetzt werden, welches *Mercury* initiierte Verbindungen vorhält. Das Konzept mit den Verbindungspools kann jedoch beibehalten werden.

Weiterhin sollte der Performanceunterschied zu klassischem TCP gemessen werden, um zu ermitteln, ob eine Steigerung messbar ist. Vor allem mit Hinblick auf andere Hardwarekomponenten, deren Verwendung nun ermöglicht worden ist.

7 Zusammenfassung

JULEA ist nun in der Lage einen weiteren Objektspeicher als Speicherlösung zu verwenden. *Ceph* ist sogar der erste echte Objektspeicher, zu dem der Client selber eine Verbindung aufbauen kann – also clientseitig läuft. Dadurch wird eine weitere Serverinstanz eingespart und die Latenz sollte deutlich geringer sein, als bei einem zusätzlichen Hop.

Die vollständige Implementation von *Mercury* war leider, wie bereits erwähnt, nicht möglich. Jedoch wurde ein großer Teil der Vorarbeit und Evaluation durch das Erstellen einer funktionierenden Referenzimplementation absolviert. Dieses Demo ist in der Lage eine Client-Server Beziehung aufzubauen, einen *RPC* zu empfangen und einen *Bulk Transfer* durchzuführen. Diese Komponenten müssten in *JULEA* in abgewandelter Form eingebaut werden.

Der Quellcode des Projektes ist zu finden auf *GitHub* in dem Repository von *wr-hamburg* unter der Adresse `https://github.com/wr-hamburg/julea`.

Zu Beginn des Projekts wurde ein Fork erstellt, um später mittels *Pull Requests* die Änderungen zu publizieren. Die Commitversion des Forks lautet **3389190ec5e1728ab52acd3837cc84b9f9639940** – die letzte lautet **c44692a99b33878d5179d7066617e7426e44f223**.

Abbildungsverzeichnis

Literatur

[1] Kernel bypass. *Marek Majkowski*. 7. Sep. 2015. URL: https://blog.cloudflare.com/kernel-bypass/ (besucht am 04. 03. 2018) (siehe S. 19).

[2] HPC Advisory Council. *Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing*. URL: http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf (besucht am 04. 03. 2018) (siehe S. 18, 20).

[3] Advanced Micro Devices. *Revision Guide for AMD Family 10h Processors*. URL: http://support.amd.com/TechDocs/41322_10h_Rev_Gd.pdf (besucht am 27. 09. 2018) (siehe S. 29).

[4] Inc Inktank Storage. *Hard Disk and File System Recommendations*. URL: http://docs.ceph.com/docs/jewel/rados/configuration/filesystem-recommendations (besucht am 05. 10. 2018) (siehe S. 29).

[5] Michael Kuhn. "JULEA: A Flexible Storage Framework for HPC". In: *High Performance Computing*. Lecture Notes in Computer Science 10524. Frankfurt, Germany: Springer International Publishing, Nov. 2017. ISBN: 978-3-319-67629-6. DOI: https://doi.org/10.1007/978-3-319-67630-2 (siehe S. 4).

[6] J. Soumagne; D. Kimpe; J. Zounmevo; M. Chaarawi; Q. Koziol; A. Afsahi; R. Ross. "Mercury: Enabling remote procedure call for high-performance computing". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. Sep. 2013, S. 1–8. DOI: 10.1109/CLUSTER.2013.6702617 (siehe S. 36).

[7] Lars Thoms. *Suitability analysis of object storage for HPC workloads*. März 2017. URL: https://kataloge.uni-hamburg.de/DB=1/XMLPRS=N/PPN?PPN=895196646 (siehe S. 5).