UNIVERSITÄT KASSEL
FACHBEREICH ELEKTROTECHNIK/INFORMATIK
FACHGEBIET ANGEWANDTE INFORMATIONSSICHERHEIT

Bachelorarbeit

Implementierung eines unstrukturierten Peer-to-peer-Netzwerkes für verteilte Berechnungen

Christopher Konze
Matrikelnummer: 29237498

UNIKASSEL
VERSITÄT
Fachgebiet Angewandte Informationssicherheit
Fachbereich Elektrotechnik/Informatik
Universität Kassel

13. April 2014

Prüfer:
Prof. Dr. Arno Wacker
Prof. Dr. Kurt Geihs
Betreuer:
M. Sc. Nils Kopal

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Rechenintensive Aufgaben sind sowohl für die Forschung als auch für die Allgemeinheit von steigendem Interesse. Die Spannweite dieser Aufgaben reicht von der Videocodierung, über die Vorhersage des Wetters bis hin zur Krebsforschung. Eine solche Aufgabe kann allerdings nicht von einem einzelnen Prozessorkern in akzeptabler Zeit berechnet werden. Eine Möglichkeit, die Berechnungszeit zu senken, wäre eine Erhöhung der Taktrate des Prozessorkerns. Dies ist jedoch nicht zu Letzt aufgrund der enormen Wärmeentwicklung sehr aufwändig und teuer. Daher besteht großer Bedarf nach einer Parallelisierung der Aufgabe. Zu diesem Zweck wurden Prozessoren mit mehreren Prozessorkernen entwickelt. Der nächste Schritt zur Erhöhung der Rechenkraft ist es, die Anzahl der parallel agierenden Prozessoren nochmals massiv zu erhöhen. Solche Hochleistungsrechner werden auch als Parallelrechner bzw. Supercomputer bezeichnet. Die Konstruktion und Wartung dieser Spezialhardware bedarf jedoch immenser finanzieller Mittel. Eine kostengünstigere Methode zu Erhöhung der Rechenkraft ist die Verwendung von Standardhardware. Hierbei werden mehrere Computer zu einem Rechnerverbund (Cluster) zusammengeschlossen und die Berechnung auf diese aufgeteilt.

1.1 Motivation

Für viele Anwendungen besteht zwar ein hohes wissenschaftliches und gesellschaftliches, jedoch kein ausreichendes monetäres Interesse. Prominente Beispiele hierfür sind etwa die Simulation des Klimawandels [Cli14] oder die Suche nach Gravitationswellen [Ein14]. Die Hardware- und Stromkosten für einen Rechnerverbund übersteigen meist den Finanzierungsrahmen. Eine mögliche Realisierung solcher Projekte ist durch das sogenannte *Volunteer Computing* möglich. Hierbei können freiwillige Teilnehmer die ungenutzte Rechenleistung ihres Computers und den zusätzlichen Stromverbrauch an das Projekt spenden.

Die meisten Volunteer Computing Netzwerke basieren auf einem zentralisierten Prinzip. Ein Teilnehmer bekommt von einer zentralen Instanz – einem Server – eine bestimmte Teilaufgabe zugewiesen. Beendet der Teilnehmer seine Berechnung, übermittelt er das Ergebnis an den Server. Dieses Vorgehen setzt bei einem großen Netzwerk eine entsprechende Leistung und Internetanbindung für die Verteilung der Aufgaben an die Teilnehmer voraus. Ein Volunteer Computing Netzwerk auf Peer-to-Peer (P2P) Basis benötigt dagegen keine zentrale Instanz, da die Aufgabenverteilung von den Teilnehmern selbst übernommen wird.

Die Kernfunktion des Volunteer Computing Netzwerkes ist das Berechnen der gemeinsamen Aufgabe. Damit möglichst viel an dem gemeinsamen Ziel gearbeitet werden kann, sollte die Verteilung der Aufgaben geringen Aufwand verursachen. Innerhalb eines strukturierten P2P-Netzwerkes existiert eine verteilte Struktur, die bei jedem Beitritt oder Verlassen des Netzwerkes angepasst werden muss. Dies erzeugt zusätzliche Arbeit für die Teilnehmer. Ein unstrukturiertes P2P-Netzwerk kommt zwar ohne diese aufwendige Struktur aus, besitzt aber einen höheren Kommunikationsaufwand. Deswegen ist es wichtig, die Kommunikation in einem unstrukturierten Netzwerk möglichst effektiv zu gestalten.

1.2 Aufgabenstellung

Der exakte Wortlaut, der vom Fachgebiet ausgehändigten Aufgabenbeschreibung lautet wie folgt:
Moderne Peer-to-Peer-Netzwerke verbinden eine große Anzahl von Rechnern in einer strukturierten oder unstrukturierten Topologie und ermöglichen es, Rechenaufgaben auf mehrere Recheneinheiten zu verteilen. Strukturierte Peer-to-Peer-Netzwerke, welche z. B. in einer Ringstruktur angeordnet werden, ermöglichen das gezielte Adressieren und somit auch das geordnete Verteilen eines Berechnungsraumes auf die in ihnen verfügbaren Ressourcen. Die Aufrechterhaltung der Netzwerkstruktur und die eindeutige Zuordnung des Rechenraumes, z. B. durch den Einsatz von Distributed Hashtables, erfordert erheblichen Aufwand und führt so zu zusätzlicher Last im Peer-to-Peer-Netzwerk. Im Gegensatz dazu besitzen unstrukturierte Peer-to-Peer-Netzwerke, welche spontan gebildet werden, keine eindeutige Adressierbarkeit und machen eine direkte Verteilung einzelner Teilaufgaben größerer Berechnungen unmöglich. Unstrukturierte Peer-to-Peer-Netzwerke bieten weder Konsistenz, noch Partitionstoleranz noch Verfügbarkeit.

Im Zuge seiner Masterarbeit hat Nils Kopal zwei neue Verteilungsalgorithmen evaluiert, welche er zuvor am Fachgebiet für Angewandte Informationssicherheit zusammen mit Arno Wacker entwickelt hatte, die in ihren Grundzügen eine Verteilung mittels Fluten im unstrukturierten Netzwerk vornehmen. So errechnet jeder Knoten im unstrukturierten Netzwerk einen zufällig gewählten Teilbereich einer größeren Berechnung. Die Ergebnisse eines Knotens werden innerhalb des Peer-to-Peer-Netzwerks geflutet. Um das Gesamtergebnis zu erhalten, müssen alle Teilergebnisse miteinander kombiniert werden. Um die Größe der gefluteten Daten (Ergebnisse der Berechnungen) in einem technisch realisierbaren Rahmen zu halten, wurden mehrere Verfahren entwickelt. Zum einen wird der Berechnungsraum in sogenannte Epochen eingeteilt. Der Berechnungsfortschritt einer Epoche wird durch eine Bitmaske repräsentiert, welche jeder Peer für sich pflegt. Jedes Bit innerhalb der Maske steht für eine Teilberechnung (1 = Teilberechnung durchgeführt, 0 = Teilberechnung noch nicht durchgeführt). Es wird somit nicht der gesamte Ergebnisraum geflutet, sondern nur der Ergebnisraum (=Bitmaske) der aktuell zu berechnenden Epoche. In einem zweiten Ansatz wird ein Sliding-Window-Verfahren

genutzt, welches jeder Peer für sich pflegt. Das Fenster entspricht einem Unterraum, wieder repräsentiert durch eine Bitmaske, des gesamten Berechnungsraums. Erhält der Peer ein Teilergebnis, welches am Anfang seines Fensters steht, schiebt er das Fenster entsprechend weiter. Beide Verfahren terminieren lokal, sobald ein Peer alle Teilergebnisse lokal kombiniert hat und er keine weiteren Teilberechnungen durchführen kann.

Hauptgegenstand dieser Bachelorarbeit ist die Implementierung eines verteilten Peer-to-Peer-Netzwerkes, welches z.B. die verteilte Kryptoanalyse im lokalen Netzwerk durchführt (bzw. simuliert). Hierfür sollen folgende Dinge zunächst konzeptionell entwickelt und dann innerhalb einer Programmbibliothek implementiert werden: Ein eigenes Netzwerkprotokoll, welches folgendes unterstützt:

- *Einwahl und Auswahl aus dem Netz (regelmäßige Pings innerhalb des Netzwerks – „Broadcasting“ im lokalen Netzwerk)*
- *Kommunikationsschicht, welche mittels UDP Paketen im lokalen Netzwerk kommuniziert*
  - *Peer-Identifikation mittels Peer-ID und Zertifikat*
  - *Signierung aller Nachrichten*
  - *„Unterschiedliche Welten“ (Nur Benutzer in der selben Welt sehen Jobs der eigenen Welt)*
- *„Verwaltungsschicht“, welche „Jobs“, die gerade im Netzwerk gerechnet werden, verwaltet*
- *„Berechnungsschicht“, welche den Epochen-Verteilungsalgorithmus und den Sliding-Window- Verteilungsalgorithmus implementiert*

1.3 Ziele der Arbeit

Die folgenden Ziele wurden an die Bachelorarbeit gestellt und sind im Zuge dieser umgesetzt worden:

- (Z1) Die Entwicklung eines Netzwerkprotokolls zur verteilten Berechnung, basierend auf dem Epochen-Verteilungsalgorithmus und dem Sliding-Window-Verteilungsalgorithmus.
- (Z2) Die Entwicklung einer .NET Bibliothek, welche das entwickelte Protokoll implementiert.
- (Z3) Die Erstellung von Beispiel-Applikationen für die entwickelte Bibliothek.
- (Z4) Die Evaluierung der Bibliothek.

1.4 Aufbau der Arbeit

Die vorliegende Bachelorarbeit ist in sieben Kapitel gegliedert. Im Anschluss an das Einleitungskapitel werden zunächst die der Arbeit zugrunde liegenden Techniken und Begrifflichkeiten erläutert.

Darauf folgend wird das Verhalten und der Aufbau des Netzwerkprotokolls beschrieben. Zudem wird die Architektur der Bibliothek vorgestellt.

Das *„Implementierungs“*-Kapitel befasst sich mit der technischen Umsetzung der entworfenen Bibliothek. Anschließend werden Beispielanwendungen für diese vorgestellt.

Es erfolgt eine Auswertung der Qualität der Bibliothek. Diese Evaluierung umfasst dabei sowohl deren Funktionalität als auch deren Wartbarkeit bzw. Erweiterbarkeit.

Im Kapitel *„Verwandte Arbeiten“* wird zunächst das am weitesten verbreitete Volunteer Computing System BOINC und anschließend ein Peer-to-Peer-basiertes Volunteer Computing System vorgestellt.

Abschließend werden die wichtigen Ergebnisse nochmals zusammen gefasst. Außerdem wird ein Ausblick über die künftige Weiterentwicklung der Bibliothek gegeben.

2 Grundlagen

In diesem Kapitel wird auf die, für diese Arbeit notwendigen, Grundlagen eingegangen. Dazu werden zunächst kurz die Grundlagen des *„Distributed Comuting“* (Verteilte Berechnung) und des *„Volunteer Computing“* erläutert. Anschließend werden wesentliche Netzwerkarchitekturen beschrieben und danach ein Überblick über Broadcast und Multicast in Netzwerken gegeben. Im Anschluss werden die, innerhalb dieser Arbeit verwendeten, Verteilungsalgorithmen vorgestellt. Darauf folgend wird eine Übersicht, der verwendeten Entwickelungsmetodiken gegeben. Abschließend werden digitale Zertifikate insbesondere das X.509 Zertifikat kurz skizziert.

2.1 Distributed Computing

Beim Distributed Computing, oder auch verteiltem Rechnen, arbeitet eine Gruppe von eigenständigen Computern gemeinsam an einer Aufgabe. Diese Computer können dabei in einem Rechnerverbund (Cluster) oder – wie etwa beim Volunteer Computing – weltweit verteilt sein. Dabei wird die Aufgabe in viele kleine Teilaufgaben unterteilt und auf die Computer verteilt. Anschließend werden die Ergebnisse der Teilaufgaben zu einem Gesamtergebnis aggregiert.

2.2 Volunteer Computing

Innerhalb des Volunteer Computing werden nicht nur die eigenen Rechenkapazitäten für eine verteilte Berechnung verwendet. Vielmehr wird es Menschen auf der ganzen Welt ermöglicht, ihre ungenutzten Rechenkapazitäten einem Projekt beizusteuern (Sogenannte *Volunteers*). Dadurch können Rechenleistungen aggregiert werden, die an die Leistung von aktuellen Supercomputern heranreichen.

Eines der zurzeit leistungsfähigsten Volunteer Computing Netzwerke ist unter dem Namen Folding@home bekannt. Es erreicht eine Rechenleistung von 20,271 PFlop/s [Fol14] und verfügt damit über mehr Leistung als der aktuell zweitbeste Supercomputer Titan mit 17,59 PFlop/s [top14b]. Der aktuell beste Supercomputer Tianhe-2 verfügt über eine Leistung von 33,862 PFlop/s [top14a]. Die Rechenleistung des Folding@home Netzwerkes wird dazu verwendet, den Aufbau von Proteinen zu verstehen. Dies soll dabei helfen, Heilungsmethoden für bestimmte Krankheiten, wie u.a. Krebs und Parkinson, zu finden.

Die meisten Volunteer Computing Netzwerke agieren nach dem Client-Server-Prinzip. Ein Client, der eine Aufgabe berechnen möchte, meldet sich bei einem Server. Dieser Server übergibt dem Client eine Teilaufgabe. Diese Teilaufgabe wird außerdem für den Client eine gewisse Zeit lang reserviert. Das bedeutet, dass kein anderer Client die Aufgabe während dieser Zeit ausgehändigt bekommt. Eine weitere Übersicht der existierenden Volunteer Computing Lösungen wird in Kapitel 6 gegeben.

2.3 Netzwerk Architekturen

Innerhalb dieses Kapitels wird die Funktionsweise einer Peer-to-Peer-basierten Netzwerkarchitektur beschrieben. Als Referenzmodel wird ebenfalls die Client-Server-Netzwerkarchitektur erläutert.

2.3.1 Client-Server-Architektur

In der klassischen Client-Server-Architektur gibt es eine klare Trennung zwischen dem Anbieter eines Dienstes (Server) und dem Nutzer des Dienstes (Client). Die Spannweite der Dienste reicht dabei von der einfachen Datensenke, bei dem der Client lediglich Daten auf dem Server zwischenspeichert (Fat Client), bis hin zu kompletten serverseitigen Berechnung, bei dem vom Client lediglich die fertigen Ergebnisse dargestellt werden (Thin Client). Wie in Abbildung 2.1 zu sehen, stellt der Server eine zentrale Einheit da, auf den mehrere Clients zugreifen können. Dieses ermöglicht eine einfache Administrierung und Verwaltung des Dienstes, setzt aber auch eine entsprechende Netzwerkanbindung und Rechenkapazitäten des Servers voraus. Dabei muss der Server nicht nur alle Clients bedienen können, sondern ebenfalls eine gewisse Ausfallsicherheit gewährleisten. Das Erreichen dieser Ziele ist in der Regel sehr kostenintensiv.

2.3.2 Peer-to-Peer-Architektur

Bei einer Peer-to-Peer-Architektur wird versucht – im Gegensatz zur klassischen Client-Server-Architektur – auf zentrale Elemente zu verzichten. Die Teilnehmer eines Netzwerkes (sogenannte *Peers*), sollen gleichberechtigt sein. Jeder Peer ist sowohl Nutzer als auch Anbieter des gemeinsamen Dienstes und muss daher neben Aufgaben des Clients auch Aufgaben des Servers übernehmen.

Peer-to-Peer-Netzwerke sind logische Netzwerke, die auf einem existierenden Netzwerk (z.B. dem Internet) aufbauen, um Verbindungen zwischen den Peers herzustellen. Aus diesem Grund werden sie auch als *Overlay*-Netzwerke bezeichnet.

Aufgrund ihrer Architektur bringen Peer-to-Peer-Netzwerke auch einige Nachteile mit sich. Zum einen kann sich nicht auf die Verfügbarkeit eines bestimmten

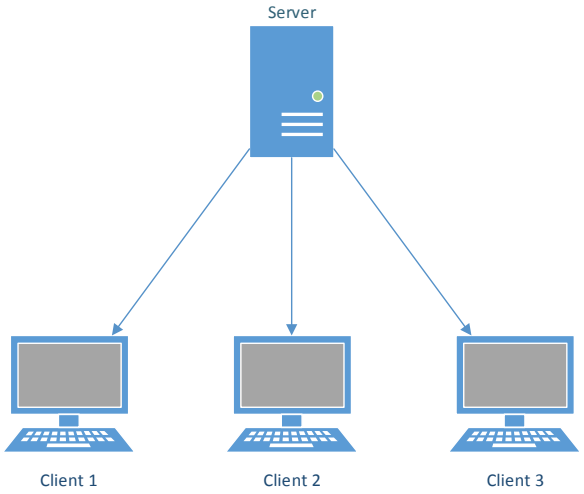

Abbildung 2.1: Client Server

Teilnehmers verlassen werden. Zum anderen ist auch die Effektivität der Hardwarenutzung in der Regel geringer als bei einer Client-Server-Architektur. Der Erfolg der meisten Peer-to-Peer-Anwendungen hängt aber oft direkt von den Vorteilen ab. So ermöglicht es die Voice-Over-IP-Software „Skype“, aufgrund geringer Serverkosten eine kostenlose Bereitstellung des Dienstes. Dies ist der Fall, weil die Kommunikation der Teilnehmer innerhalb eines Peer-to-Peer-Netzwerkes stattfindet. Peer-to-Peer-Netzwerke ermöglichen aber auch Anwendungen wie die dezentrale anonyme Kryptowährung Bitcoin, die ohne eine zentrale Instanz auskommt [Nak08].

Peer-to-Peer-Architekturen können grob in strukturierte und unstrukturierte Netzwerke unterteilt werden. Innerhalb eines strukturierten Netzwerkes wird eine Struktur für die Adressierung von Teilnehmern oder von Daten aufgebaut. Ein Beispiel für ein strukturiertes P2P-Netzwerk stellt Chord dar [FK14]. Hierbei ordnen sich, wie in Abbildung 2.2 zu sehen ist, die Teilnehmer in einer Ringstruktur an. Dabei ist jeder Teilnehmer für einen bestimmten Bereich der gesamten Datenmenge zuständig. Für die Suche ist jeder Teilnehmer mit mehreren anderen Teilnehmern verbunden. Diese Verbindungen (sogenannte *Fingers*) durchqueren dabei den Ring, wobei sich der Abstand zwischen diesen jedesmal verdoppelt. Innerhalb der Struktur kann dadurch mit eine Komplexität von $O(log(n))$ nach Daten gesucht werden. Wenn ein Teilnehmer dem Netzwerk beitritt oder es verlässt, muss der Ring entsprechen modifiziert werden.

In einem unstrukturierten Peer-to-Peer-Netzwerk hingegen werden die Nachrichten an eine Anzahl beliebig gewählter Nachbarn versendet. Diese Vorgehensweise erfordert eine Selbstorganisierung der Teilnehmer, welche durch Verteilungsalgorithmen umgesetzt wird. Die innerhalb dieser Arbeit verwendeten Verteilungsalgorithmen werden im Kapitel 2.5 im Einzelnen beschreiben.

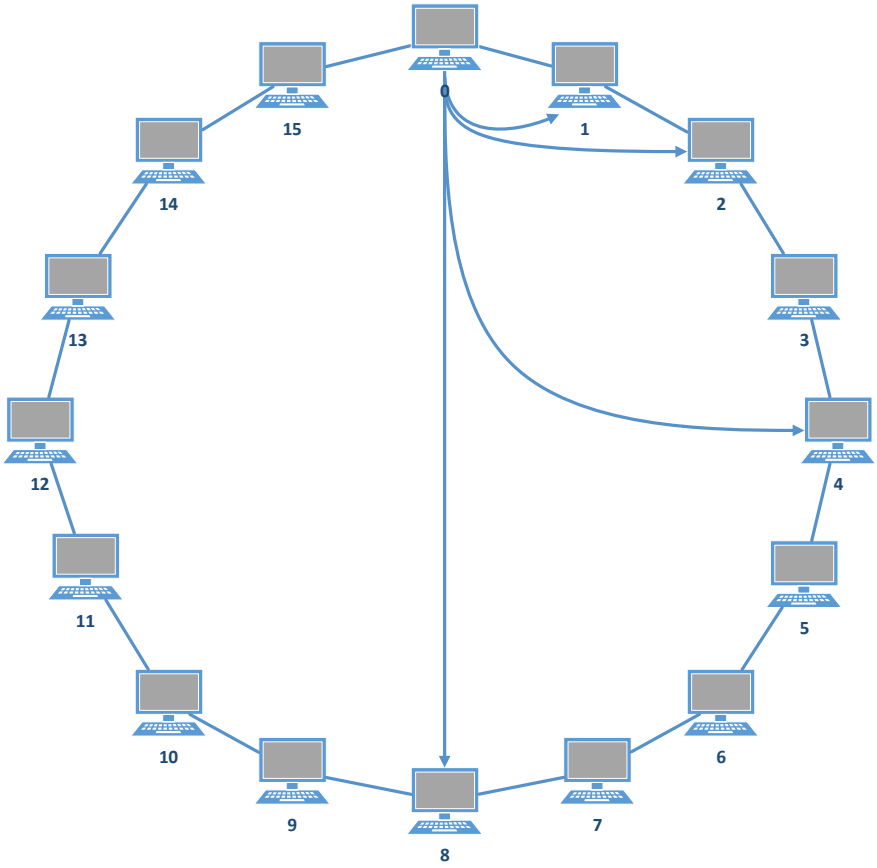

Abbildung 2.2: Chord Netzwerk (Basiert auf Abbildung in [Kop12])

2.4 Übertragungprotokolle

Im Folgenden werden die Grundlagen und die technische Umsetzung von Broadcasts in IPv4 und von Multicast in IPv4 und IPv6 dargelegt [Pos81, Dee98].

2.4.1 Broadcast

Ein IP-Broadcast ist eine spezielle Form einer IPv4-Nachricht, die an alle Teilnehmer eines Netzwerkes gerichtet ist (*One-to-Many*). Ein Broadcast wird unter anderem benötigt, wenn Informationen von einem noch unbekannten Teilnehmer des lokalen Netzwerk gesucht werden. Ein Beispiel dafür ist das Dynamic Host Configuration Protocol (DHCP) [Dro97], welches zur automatischen Einwahl in ein Netzwerk verwendet wird. Der beitretende Teilnehmer sendet einen Broadcast um einen DHCP-Server zu finden, mit dem er dann die Konfiguration für das Netzwerk austauschen kann.

Im Idealfall sollte für einen Broadcast in das lokale Netzwerk nur eine Nachricht verschickt werden müssen. Dies kann allerdings nicht auf IP-Ebene direkt statt-

finden, sondern muss von dem darunterliegenden Protokoll des Link-Layers unterstützt werden. Da zumindest die gängigen Link-Layer Protokolle (z.B.: Ethernet, IEEE 802.11) dies nativ unterstützen, sind Router in einer solchen Umgebung dazu angehalten, IP-Broadcasts an das lokale Netzwerk als Broadcast des Link-Layers weiterzuleiten [Bak95].

Für das Versenden von lokalen Broadcasts ist die IP-Adresse 255.255.255.255 vorgesehen. IP-Broadcasts können auch an Teilnehmer eines entfernten Netzes gesendet werden [Mog84]. Diese sollen allerdings aus Sicherheitsgründen nicht mehr weitergeleitet werden [Sen99]. In IPv6 existiert diese Art des Broadcasts nicht mehr, sondern wird durch das Prinzip von Multicast abgelöst.

2.4.2 Multicast

Ein Multicast ist ebenfalls eine Nachricht, die an mehrere Empfänger gerichtet ist. Die Empfänger bilden dabei eine logische Gruppe (Multicast-Gruppe), die beliebig betreten oder verlassen werden kann. Die Multicast Gruppe wird durch eine IP-Adresse identifiziert. Hierfür sind, unter IPv4 der Adressbereich von 224.0.0.0 bis 239.255.255.255 und unter IPv6 der Adressraum ff00::/8 reserviert. In beiden IP-Versionen gibt es gewisse Restriktionen für das Weiterleiten von Multicast Nachrichten. So werden unter IPv4 der Adressraum 224.0.0.0/24 und unter IPv6 der Adressraum ff02::/8 nur im lokalen Netzwerk weitergeleitet [Dee88, Dee98].

Zur Verwaltung der Mitgliedschaften einer Multicast-Gruppe innerhalb eines lokalen Netzwerkes werden Protokolle wie das Internet Group Management Protocol (IGMP) oder IGMPv6 verwendet [Fen97]. Hierbei kann der Router abfragen, welche Gruppen in seinem Netzwerk beigetreten wurden. Zudem können die Computer des Netzwerks aktiv einer Gruppe beitreten bzw. diese verlassen.

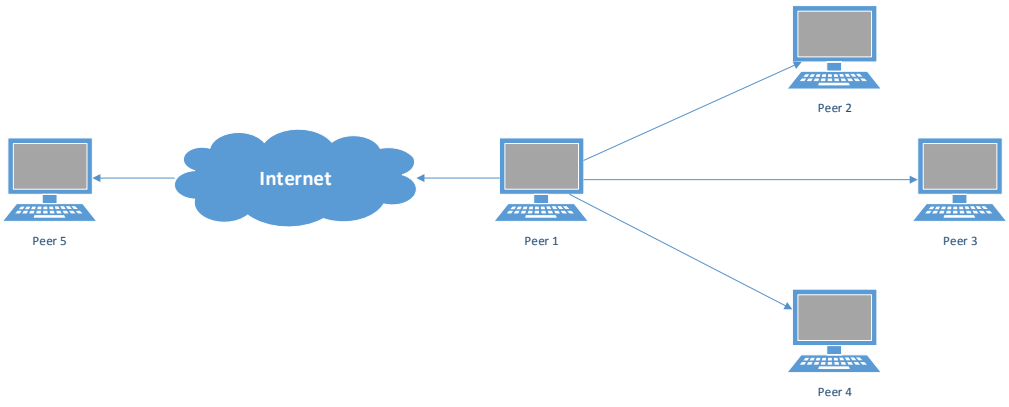

Abbildung 2.3: Multicast über Netzwerkgrenzen

Unter IPv4 ist Multicast eine optionale Erweiterung. Daher kann sich nicht darauf verlassen werden, dass ein Router einen Multicast außerhalb des lokalen Netzwer-

kes weiterleitet. Um eine, wie in Abbildung 2.3 dargestellte, netzwerkübergreifende Struktur zu erhalten, werden in IPv4 spezielle Multicast Backbones (MBONE) für das Routing verwendet. Unter IPv6 hingegen muss jeder Router multicastfähig sein.

Für das Senden an eine Multicast Gruppe muss der Absender selbst nicht der Gruppe beitreten. Er sendet seine Nachricht direkt an die IP-Adresse der Gruppe woraufhin die Router diese entsprechend weiterleiten. Hierbei gilt Ähnliches wie bei dem in 2.4.1 beschriebenen Broadcast. Wenn der Link-Layer One-to-Many-Verbindungen unterstützt, wird innerhalb eines lokalen Netzwerkes die Nachricht nur einmal versendet.

2.5 Verteilungsalgorithmen

Ein essenzieller Bestandteil der verteilten Berechnungen ist die gleichmäßige Verteilung der Aufgaben auf die Teilnehmer. Hierzu gibt es grundsätzlich zwei Ansätze: Die zentrale und die dezentrale Aufgabenverteilung. Im Gegensatz zu der zentralen Verteilung, bei der eine zentrale Instanz die Aufgaben an die Teilnehmer verteilt, erfordert eine dezentrale Aufgabenverteilung eine Selbstorganisation innerhalb des Netzwerkes.

Im Kontext dieser Bachelorarbeit werden zwei dezentrale Verteilungsalgorithmen verwendet, der Epochen-Verteilungsalgorithmus und der Slinding-Window-Verteilungsalgorithmus. Die Algorithmen wurden am Fachgebiet Angewandte Informationssicherheit der Universität Kassel entwickelt und in [Kop12] evaluiert. Beide Verteilungsalgorithmen verfolgen einen ähnlichen Ansatz. Zunächst wird vorausgesetzt, dass die zu berechnende Aufgabe C in n Unteraufgaben $C_1, C_2, \ldots, C_n$ aufgeteilt werden kann und das Ergebnis R sich mittels einer Verknüpfung $\circ$ aus den Teilergebnissen $R_1, R_2, \ldots, R_n$ ergibt. Die Verknüpfung $\circ$ muss dabei sowohl assoziativ, kommutativ sowie idempotent sein.

Es wird eine Bitmaske B der Größe n angelegt, in der das i-te Bit den Berechnungszustand der Teilaufgabe C_i repräsentiert. Jedes Bit besitzt zunächst den Wert 0. Sobald eine Teilaufgabe C_i berechnet wurde und das Teilergebnis R_i, mit den bereits vorhandenen Ergebnissen verknüpft wurde ($R_{tmp} = R_{tmp} \circ R_i$), wird das entsprechende Bit der Bitmaske auf 1 gesetzt. Die Bitmaske sowie die Teilergebnisse werden im Netzwerk mittels Fluten – also dem Senden an möglichst viele Teilnehmer – ausgetauscht. Zur Bestimmung der zu berechnenden Teilaufgabe wählt jeder Teilnehmer zufällig eine noch nicht berechnete Teilaufgabe aus und beginnt mit der Berechnung. Sobald die Berechnung abgeschlossen ist, setzt der Teilnehmer das entsprechende Bit der Bitmaske auf 1 und flutet diese im Netzwerk.

In der Praxis stellt sich allerdings das Problem, dass eine Bitmaske der Teilaufgaben zu groß wäre, um sie effektiv im Netzwerk zu fluten. Daher verwenden die

beiden Algorithmen unterschiedlich Methoden zu Verkleinerung der zu flutenden Daten. Diese werden im Folgenden erläutert.

2.5.1 Epochen-Verteilungsalgorithmus

Der Epochen-Verteilungsalgorithmus löst die in 2.5 beschriebene Problematik durch die Einführung von festen Epochen. Wie in Abbildung 2.4 zu sehen ist, wird die Bitmaske für die Gesamtmenge der Teilaufgaben (Vollständiger Berechnungsraum) in kleinere Bitmasken, den Epochen, gleichmäßig aufgeteilt. Die Epochen werden dabei durch einen eindeutigen Index voneinander unterschieden. Wobei hier nur die EpochenBitmaske und der EpochenIndex geflutet werden muss.

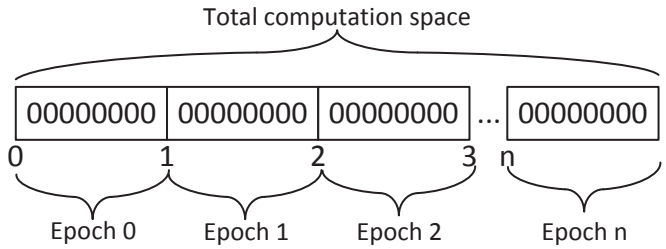

Abbildung 2.4: Epochenverteilung im Berechnungsraum [Kop12]

Nach der Initialisierung des Epochen-Algorithmus und dem Beitritt in das Netzwerk startet jeder Teilnehmer mit einer leeren Epochen-Bitmaske B_{local} in der nullten Epoche $I_{local} = 0$ und einer leeren Ergebnismenge R_{local}. Der Hauptteil des Algorithmus arbeitet wie in 2.5 beschreiben. Wobei die Epoche die Bitmaske darstellt. Sobald alle Teilaufgaben der aktuellen Epoche berechnet wurden, wird geprüft, ob es sich bei dieser um die letzte Epoche gehandelt hat. Ist dies nicht der Fall, wird B_{local} mit Nullen gefüllt, I_{local} erhöht und mit der Berechnung der neuen Epoche begonnen. Wenn die aktuelle Epoche die Letzte war, terminiert der Algorithmus.

Asynchron zum Hauptteil des Algorithmus kann ein Teilnehmer Nachrichten von anderen Teilnehmern empfangen. Diese Nachrichten bestehen aus einer Bitmaske $B_{neighbor}$, einem Epochenindex $I_{neighbor}$ und den bisherigen kumulierten Ergebnissen $R_{neighbor}$. Zunächst werden die Epochenindizes verglichen. Ist $I_{neighbor}$ kleiner als I_{local}, sind die empfangenden Daten veraltet und werden deswegen komplett ignoriert. Sollte $I_{neighbor}$ größer als I_{local} sein, wird vor dem nächsten Schritt die eigene Bitmaske B_{local} vollständig mit Nullen gefüllt und $I_{local} = I_{neighbor}$ gesetzt. Anschließend wird die eigene Bitmaske und die empfangenen Bitmaske mittels der logischen OR-Operation verknüpft und das Ergebnis als neues B_{local} gesetzt ($B_{local} = B_{local} \mid B_{neighbor}$). Weiterhin werden die Ergebnisse zusammengefügt ($R_{local} = R_{local} \circ R_{neighbor}$). Abschließend wird überprüft, ob es Änderungen am eigenen Zustand $\{I_{local}, B_{local}, R_{local}\}$ gegeben hat. Sollte dies der Fall sein, wird der neue Zustand an die eigenen Nachbarn gesendet.

2.5.2 Sliding-Window-Verteilungsalgorithmus

Der Sliding-Window-Algorithmus verfolgt eine andere Strategie zur Reduzierung der zu sendenden Daten. Dieser schiebt, wie Abbildung 2.5 zeigt, ein Fenster über den gesamten Berechnungsraum. Der Start des Fensters wird durch einen Fenster-Offset angegeben. Innerhalb des Fensters findet die Berechnung statt, somit muss nur die Bitmaskte des Fensters und der Fenster-Offset im Netzwerk geflutet werden.

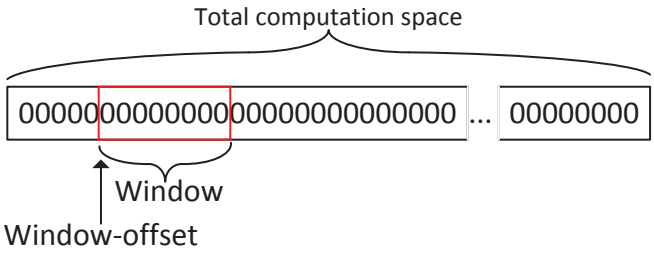

Abbildung 2.5: Sliding-Window Verteilung [Kop12]

Nachdem die Teilnehmer dem Netzwerk beigetreten sind und der Sliding-Window-Algorithmus initialisiert wurde, beginnen die Teilnehmer mit einem Fenster-Offset o_{local} von 0, einer mit Nullen gefüllten Fenster-Bitmaske W_{local} und einer leeren Ergebnismenge R_{local}. Die Berechnung der Aufgaben erfolgt nach dem in 2.5 beschriebenen Verfahren, wobei W_{local} die Bitmaske darstellt. Nachdem ein Ergebnis berechnet wurde, wird W_{local} solange nach links verschoben, bis das niedrigwertigste Bit 0 ist. o_{local} wird dabei jedes Mal um 1 erhöht. Sollte o_{local} der Größe des vollständigen Berechnungsraumes entsprechen, terminiert der Algorithmus.

Beim Empfang eines Zustandes $\{o_{neighbor}, W_{neighbor}, R_{neighbor}\}$ werden die beiden Fenster $W_{neighbor}$ und W_{local} entsprechend der Fenster-Offsets angeglichen. Dazu wird das Fenster mit dem kleineren Offset solange verschoben, bis die beiden Offsets gleich sind. Nun können die beiden Fenster $W_{neighbor}$ und W_{local} mittels OR verknüpft werden und die beiden Ergebnismengen $R_{neighbor}$ und R_{local} kombiniert werden. Sollte sich nach dem Zusammenfügen der beiden Zustände eine Änderung ergeben haben, werden die Nachbarn über diese informiert.

2.6 Entwicklungsmetodiken

Im Folgenden werden die Methodiken beschrieben, die innerhalb dieser Arbeit Verwendung fanden. Dabei wird zunächst das Konzept des Test-driven-developement vorgestellt und anschließend wird der verwendete Softwareentwickelungsprozess skizziert.

2.6.1 Test-Diven-Development

Die in dieser Arbeit entwickelte Bibliothek (VoluntLib) wurde konsequent nach der Test-Driven-Development (TDD) Methodik entwickelt. TDD ist eine agile Entwickelungsmethodik bei der – im Gegensatz zu klassischen Entwicklungsmethoden – die Tests vor der Funktionalität selbst erstellt werden. Dabei wird wie folgt vorgegangen:

1. Zunächst wird ein Unit-Test erstellt, der die neue Funktionalität validiert. Dieser Test wird vorerst nicht lauffähig sein, da die Funktionalität selbst noch nicht existiert.
2. Als nächstes wird genau soviel produktiver Quellcode geschrieben um alle Tests zu erfüllen.
3. Anschließend wird der Quellcode allgemeinen Clean-Code Regeln angepasst und abschließend noch einmal alle Tests durchgeführt.

Durch diese Vorgehensweise entstehen kurze Entwicklungszyklen, wodurch der Entwickler ein schnelles Feedback darüber bekommt, ob seine Funktionalität die Aufgabe erfüllt. Zudem werden häufig Werkzeuge zur automatisierten Ausführung der Tests verwendet. Dies verkürzt das Intervall zwischen den Feedbacks zusätzlich.

Durch die richtige Verwendung von TDD entsteht zwangsläufig eine modular aufgebaute Software mit loser Koppelung der Module untereinander, da jede Funktionalität isoliert testbar sein muss.

Da die komplette Funktionalität des Programms durch Tests abgedeckt ist, kann das Programm nach Belieben verändert werden, ohne das Gefahr besteht, versteckte Fehlerquellen einzubauen. Dies erleichtert sowohl das Aufräumen und Umstrukturieren des Programms (Wartbarkeit) als auch dessen Erweiterbarkeit. Diese Eigenschaften werden zusätzlich durch den modularen Aufbau und die lose Koppelung innerhalb der Software verstärkt.

2.6.2 Continuous Delivery

Continuous Delivery (CD) beschreibt einen Softwareentwickelungsprozess, der gezielt auf viele und kurze Veröffentlichungszyklen einer Anwendung setzt. Dabei wird zunächst eine kleinstmögliche funktionsfähige Version der Software erstellt und diese dann in weiteren Versionen um Funktionalitäten erweitert. Durch diese Vorgehensweise kann bereits früh im Entwicklungsprozess eine Rückmeldung des Anwenders eingeholt werden. Dies vermindert zum einen das Risiko einer Fehlentwicklung aber es ermöglicht auch die Anwendung früh in realen Szenarien testen zu können.

Um schnelle Veröffentlichungszyklen realisieren zu können, wird für jede Version der Software vorab ein Akzeptanz-Test angelegt. Sobald dieser und die, im

Zuge des Test-driven-developements entwickelten, Unit-Tests fehlerfrei sind, ist der Veröffentlichungszyklus abgeschlossen und es kann der Akzeptanz-Test für die nächste Version erstellt werden.

2.7 Digitale Zertifikate

Digitale Zertifikate werden verwendet um bestimmte Eigenschaften eines Kommunikatiospartners (z.B. die Identität) zu bestätigen. Da die Kommunikationspartner sich zunächst untereinander nicht vertraut können, bedarf es hierzu einer vertrauenswürdige dritte Partei (*Trusted Third Party*). Diese händigt, von ihr kryptografisch signierte, Bescheinigung (Zertifikat) für die entsprechenden Eigenschaft an die Kommunikationspartner aus.

X.509 ist ein verbreiteter Standard für digitale Zertifikate [SHF02]. Diese Zertifikate werden u.A. für die Identifizierung von Webseiten bzw. die Verschlüsselung der Kommunikation mit Webseiten eingesetzt [AD99].

Für die Ausstellung eines X.509 Zertifikates exsistiert ein hierarchisches System von Zertifizierungsstellen (*Certificate Authorities*). Dies Zertifizierungsstellen sind ebenfalls in der Lage, über eine Sperliste, ausgestellte Zertifikate für ungültig zu erklären. Ein X.509 Zertifikat beinhaltet dabei u.A. Informationen über den Zertifikatinhaber, die Zertifizierungsstelle, die Gültigkeit und die verwendeten kryptologischen Algorithmen.

3 Konzept und Design

In diesem Kapitel wird zunächst das Netzwerkprotokoll konzeptionell beschrieben und darauf folgend der Aufbau der Protokollnachrichten erläutert. Im Anschluss wird die Achritektur der entwickelten Verteilungsbibliothek „VoluntLib“ vorgestellt.

3.1 Netzwerkprotokoll

In diesem Abschnitt wird das, in dieser Bachelorarbeit entwickelte, Netzwerkprotokoll zur Verteilung und Verwaltung von verteilten Berechnungen vorgestellt. Zunächst werden die Anforderungen dargelegt. Im Anschluss wird zunächst die Konzeption des Protokolls vorgestellt und darauf folgend die Strukturen und Nachrichten dieses Protokolls.

3.1.1 Anforderungen an das Netzwerkprotokoll

Das zu entwerfende Netzwerkprotokoll soll in der Lage sein, die folgenden Ziele umzusetzen.

A1 **Aufgabenverwaltung** - Das Protokoll soll es ermöglichen, Aufgaben in einem Netzwerk zu erstellen und diese unter den Teilnehmern des Netzwerkes auszutauschen. Zusätzlich soll es dem Ersteller der Aufgabe oder einem Administrator möglich sein, die Aufgabe aus dem Netzwerk zu entfernen.

A2 **Welten** - Das Protokoll soll es ermöglichen Aufgaben, unterschiedlichen Welten zuzuordnen. Ein Teilnehmer soll alle Aufgaben innerhalb einer von ihm gewählten Welt erhalten können. Auch eine Abfrage der existierenden Welten soll möglich sein.

A3 **Berechnen** - Das Protokoll soll es ermöglichen, die aktuell besten Teilergebnisse und den Zustand zu einer Aufgaben unter den Teilnehmer auszutauschen oder diese anzufordern.

A4 **Algorithmen** - Das Protokoll soll die Verwendung des Epochen-Verteilungs-Algorithmus und des Sliding-Window-Verteilungs-Algorithmus für die Verteilung ermöglichen, aber nicht an diese Algorithmen gebunden sein.

A5 **Nachweisbarkeit** - Innerhalb des Protokolls soll jede Nachricht eindeutig einer Person zuordbar sein. Zudem sollen nur autorisierte Personen an einem Netzwerk teilnehmen können.

A6 **Erweiterbarkeit** - Das Protokoll soll versionierbar sein. Es soll ermöglicht werden, Informationen hinzuzufügen und dennoch kompatibel zu alten Versionen zu bleiben. Dies wird auf als Rückwärtskompatibilität bezeichnet.

A7 **One-to-Many und Bidirektional** - Das Protokoll soll primär auf Basis des Multicasts agieren. Zusätzlich soll aber auch eine direkte bidirektionale Verbindung zwischen zwei Teilnehmern möglich sein.

3.1.2 Konzeption des Protokolls

Zunächst wird das Protokoll konzeptionell beschrieben. Dabei wird das Verhalten des Absenders und der Empfänger dargelegt und erläutert. Da in den meisten Anwendungsfällen mehr als nur ein Empfänger existiert, werden diese zur Vereinfachung als Netzwerk bezeichnet. Wobei aus einem Netzwerk immer nur ein Teilnehmer antworten soll. Die Umsetzung dazu wird in 3.1.3 erläutert. Der Aufbau der verwendeten Nachrichten wird in Abschnitt 3.1.4 beschrieben.

3.1.2.1 Nachweisbarkeit und Autorisierung

Zur Erfüllung der geforderten Nachweisbarkeit wird jede Nachricht mittels eines X.509 Zertifikates [SHF02] signiert. Der öffentliche Teil des verwendeten Zertifikates wird jeder Nachricht beigefügt. Beim Empfang einer Nachricht wird zunächst überprüft, ob das Zertifikat zur Teilnahme am Netzwerk autorisiert ist. Dies ist dann der Fall, wenn es von einer definierten Zertifizierungsstelle ausgestellt wurde. Anschließend wird die Singnatur der Nachricht überprüft. Diese muss mit dem, in der Nachricht angegeben, Zertifikat erstellt worden und valide sein. Sollte eine der beiden Überprüfungen fehlschlagen, wird die Nachricht verworfen.

3.1.2.2 Teilnahme an einer Aufgabe

Grundsätzlich gibt es zwei Möglichkeiten, an einer Aufgabe mitzuwirken.

- **Aktive Teilnahme** - Bei einer aktiven Teilnahme werden eigene Ressourcen dazu verwendet, Teilergebnisse der Aufgabe zu berechnen und die Ergebnisse bzw. den Berechnungszustand im Netzwerk zu verwalten.
- **Passive Teilnahme** - Bei einer passiven Teilnehme werden keine eigenen Berechnungen durchgeführt, aber bei der Verwaltung der Teilergebnisse und des Berechnungszustandes mitgeholfen.

Eine solche Unterteilung ist nötig, da nicht jeder Teilnehmer des Netzwerks die Berechungslogik zu jeder Aufgabe haben muss. Die Berechungslogik ist dazu in der Lage, anhand der ausgetauschten Informationen Teilergebnisse der Aufgabe zu berechnen. Teilnehmer ohne die aufgabenspezifische Berechungslogik können aber dennoch dafür sorgen, dass das Netzwerk auch den Zustand und die Ergebnisse von Aufgaben verwalten kann, an denen aktuell kein Teilnehmer aktiv arbeitet.

3.1.2.3 Aufgabe erstellen

Um eine Aufgabe im Netzwerk anzulegen, wird die CreateNetworkJob-Nachricht verwendet. Die in 3.1.4.2.1 beschriebene Nachricht enthält alle Informationen zu der Aufgabe und kann daher zustandslos an alle Teilnehmer versendet werden. Sobald eine Nachricht empfangen wird, fügt der Teilnehmer die erstellte Aufgabe seiner Aufgabenliste hinzu. Zusätzlich wird der Name des Erstellers gespeichert.

3.1.2.4 Aufgabe entfernen

Zur Löschung einer Aufgabe im Netzwerk wird die DeleteNetworkJob-Nachricht verwendet. Da die Löschung einer Aufgabe nur von ihrem Ersteller oder von einem Administrator vorgenommen werden darf, wird dies bei dem Empfang einer DeleteNetworkJob-Nachricht zunächst überprüft. Ist die Nachricht valide, wird die Aufgabe vorerst für eine definierte Zeitspanne T_{store} lokal als gelöscht markiert und zwischengespeichert. Erhält der Teilnehmer innerhalb von T_{store} eine die Aufgabe betreffende Nachricht, wird darauf mit der gespeicherten DeleteNetworkJob-Nachricht geantwortet. Die Nachricht wird dabei nicht selbst neu signiert, da sie sonst invalide würde. Nach Ablauf von T_{store} wird die Aufgabe gelöscht.

3.1.2.5 Aufgabenaustausch

Der Austausch der Aufgaben wurde in zwei Prozesse aufgeteilt. Zum einen können die Metadatan von allen Aufgaben innerhalb einer Welt ausgetauscht werden. Hierzu wird die RequestJobList bzw. die ResponseJobList-Nachricht verwendet. Wie in Abbildung 3.1 zu sehen ist, wird beim Empfang einer RequestJobList-Nachricht alle lokal verfügbaren Aufgaben, die der angefragten Welt zugeordnet sind, mittels der ResponseJobList-Nachricht versendet.

Zudem können mithilfe der ResponseJobList und der ResponseJobDetails, Zusatzinformationen zu einer Aufgabe ausgetauscht werden. Der Ablauf dieser Prozedur ist in Abbildung 3.2 zu sehen. Dabei antwortet ein Teilnehmer, wenn der über die Zusatzinformationen verfügt. Ansonsten wird die Request-Nachricht verworfen.

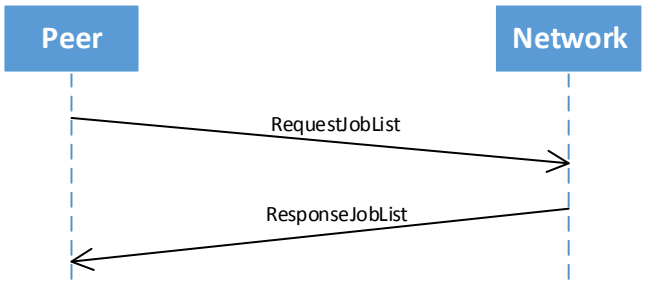

Abbildung 3.1: Austausch der Aufgaben

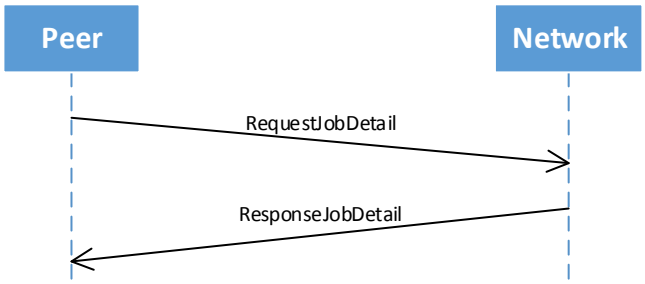

Abbildung 3.2: Austausch der Zusatzinformationen zu einer Aufgabe

3.1.2.6 Weltenaustausch

Die Teilnehmer können die existierenden Welten innerhalb eines Netzwerkes mittels der RequestWorldList-Nachricht erfragen. Sobald eine Anfrage empfangen wird, werden alle bekannten Welten, denen mindestens eine Aufgabe zugeordnet ist, mittels einer ResponseWorldList-Nachricht gesendet. Abbildung 3.3 verdeutlicht den Prozess.

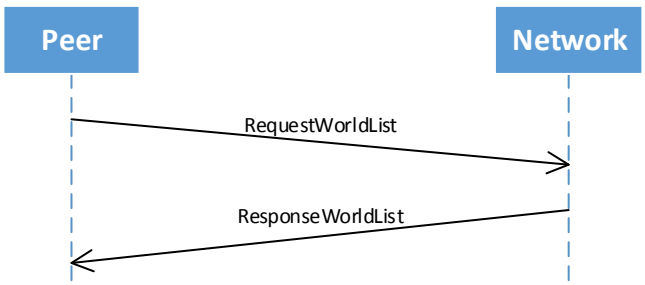

Abbildung 3.3: Sequenzdiagram des Welten-Austausches

3.1.2.7 Berechnung beginnen

Um an einer verteilten Berechnung aktiv mitzuwirken, wird die JoinNetworkJob-Nachricht verwendet. Der Prozess des beitrittes ist in Abbildung 3.4 verdeutlicht und wird im Folgenden beschrieben. Nach dem Versenden der Nachricht wartet der

sendende Teilnehmer ein definiertes Zeitintervall. Nach Ablauf des Zeitintervalls kann er davon ausgehen, dass er im Besitz des aktuellen Berechnungszustandes ist und kann mit seiner Berechnung beginnen. Sollte er keine Antwort erhalten, wurde die Aufgabe noch nicht gestartet und er beginnt die Berechnung mit dem initialen Zustand der Aufgabe.

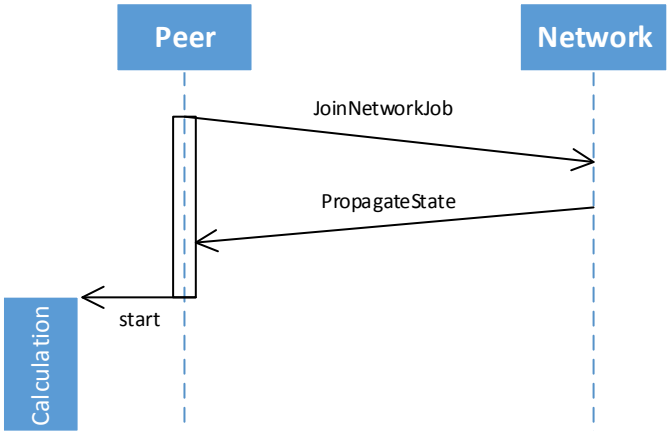

Abbildung 3.4: Beitritt zu einer Berechnung

Die JoinNetworkJob-Nachricht stellt eine Kurzform, der im nächsten Abschnitt beschriebenen, PropagateState-Nachricht dar. Der Empfänger reagiert dabei auf die gleiche Weise, als würde er eine PropagateState-Nachricht mit dem Initialzustand der jeweiligen Aufgabe erhalten. Er antwortet mit seinem aktuellen Zustand.

3.1.2.8 Ergebnisaustausch

Der Austausch der aktuell besten Teilergebnisse und des Berechnungszustandes geschieht über die PropagateState-Nachricht. Diese Nachricht soll nach jeder Berechnung eines Blockes an alle Teilnehmer gesendet werden.

Der Empfänger einer validen PropagateState-Nachricht hat – abhängig von seinem lokalen Berechnungszustand – drei mögliche Arten zu reagieren.

1. Der eigene Zustand ist eine echte Obermenge des empfangenen Zustandes.

   In diesem Fall hat der Empfänger bereits mehr Informationen über die Berechnung als der Sender und kann die empfangen Daten ignorieren. Weiterhin teilt er seinen Zustand mit dem Absender.

2. Der eigene Zustand ist eine Teilmenge des empfangenen Zustandes.

   In diesem Fall hat der Sender mindestens genau so viele Informationen wie der Empfänger. Daher können der lokale Zustand und die lokal besten Ergebnisse des Empfängers mit den empfangen Daten überschrieben werden.

3. Es existiert eine Differenzmenge zwischen eigenem und empfangenem Zustand.

   In diesem Fall haben der Sender und der Empfänger E verschiedene Blöcke berechnet. Hier gibt es wiederrum zwei mögliche Arten zu reagieren.

   Sollte der Empfänger die Berechnungslogik für die Aufgabe besitzen (aktive Teilnahme), fügt er die lokalen und empfangenen Daten zusammen und überschreibt die lokalen Daten mit den zusammengefügten. Der neue Zustand wird nun ebenfalls dem Absender mitgeteilt.

   Sollte der Empfänger die Berechnungslogik für die Aufgabe nicht besitzen (passive Teilnahme), wird der Zustand mit den meisten Informationen lokal abgespeichert und der alte Zustand wird an den Absender bzw. an die Multicast-Gruppe geschickt.

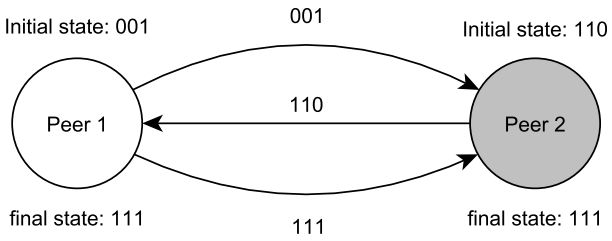

Abbildung 3.5: Austausch mit partieller Berechungslogik

Diese Vorgehensweise sorgt dafür, dass jedem Teilnehmer die Möglichkeit gegeben wird, die Zustände zusammenzufügen ohne Informationen zu verlieren. In der Abbildung 3.5 wird dies veranschaulicht, wobei Peer 2 nicht über die Berechungslogik verfügt. Zunächst schickt Peer 1 den Zustand 001. Peer 2 hat nun eine Differenz zwischen seinem und dem Empfangenem. Er behält den „besseren“, also den mit den meisten Berechnungen, Zustand 110 und schickt seinen alten 110 an Peer 1. Peer 1 ist nun in der Lage, anhand der Berechungslogik die beiden Zustände zusammenzufügen und das Resultat an Peer 2 zu schicken.

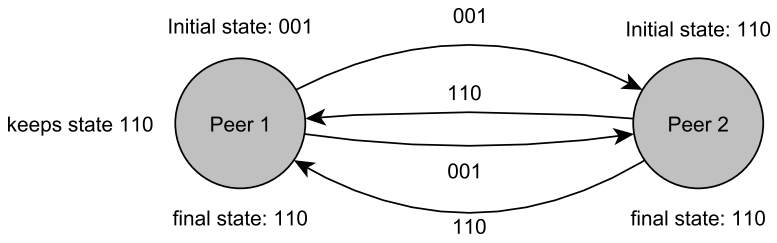

Abbildung 3.6: Austausch ohne Berechungslogik

Wenn kein Teilnehmer im Besitz der Berechungslogik ist, behält das Netzwerk mit dieser Methode den besseren Zustand. Dieses wird anhand von Abbildung 3.6 verdeutlicht. Zunächst schickt Peer 1 den Zustand 001. Peer 2 behält nun den seinen besseren Zustand 110 und schickt ihn an Peer 1. Dieser behält nun wiederum ebenfalls den besseren Zustand 110 und sendet, gemäß Protokoll, seinen alten Zustand 001. Peer 2 antwortet daraufhin wieder mit dem Zustand 110. Peer 1 empfängt nun seinen gespeicherten Zustand und antwortet nicht mehr.

Somit kann das Netzwerk auch Aufgaben mit minimalen Verlusten verwalten ohne die Berechungslogik für die Aufgabe zu kennen.

3.1.3 Netzwerkbedingte Anpassungen

Das Protokoll ist grundsätzlich unabhängig von dem verwendeten Netzwerk. Zur Verbesserung der Skalierbarkeit und der Verfügbarkeit von Daten werden allerdings einige Anpassungen vorgenommen.

3.1.3.1 Broadcast/Multicast

Um die Anzahl der Nachrichten in einer Multicastgruppe bzw. einer Broadcastumgebung zu reduzieren, soll möglichst nur ein Teilnehmer auf eine Anfrage antworten. Damit dies gewährleistet ist, wird das im Folgenden beschriebene Verfahren verwendet.

Möchte ein Teilnehmer P auf eine Anfrage antworten, wählt P zunächst einen zufälligen Zeitpunkt T_x innerhalb eines definierten Antwortintervalls I. Sollte bis zum Eintreten von T_x kein anderer Teilnehmer geantwortet haben oder P über eine bessere Antwort verfügen, so wird die Antwort geschickt.

Durch dieses Verfahren wird sichergestellt, das innerhalb des Antwortintervalls I die beste Antwort mit möglichst wenig Nachrichten ausgetauscht wurde. Zudem werden somit die Daten unter allen Teilnehmern synchronisiert.

3.1.3.2 Direkte Verbindung

Bei einem direktem Austausch zwischen zwei Teilnehmern kann aufgrund der Peer-to-Peer Struktur nicht vorausgesetzt werden, dass beide Teilnehmer zu einem beliebigen Zeitpunkt wieder miteinander Kontakt aufnehmen können.

Daher müssen die ausgetauschten Daten nach dem Beenden der Verbindung vollständig sein. Innerhalb des Protokolls werden allerdings, aufgrund der begrenzten Paketgröße beim Senden der Aufgabenlisten (ResponseJobList), Aufgaben ohne

ihre Zusatzinformationen ausgetauscht. Deswegen müssen bei jeder direkt ausgetauschten Aufgabe entweder die Zusatzinformationen beim Empfänger schon vorhanden sein oder diese im Zuge der Verbindung von dem Sender angefordert werden. Hierzu wird nach dem in 3.1.2.5 beschriebenen Verfahren argiert. Weiterhin darf der Sender nur Aufgaben schicken, bei denen er auch über die Zusatzinformationen verfügt.

Dies ist der Grund, warum eine direkte Verbindung zwischen zwei Teilnehmern zwangsläufig bidirektional sein muss. Der Aufbau der direkten Verbindung ist nicht Teil des Protokolls.

3.1.4 Spezifikationen des Protokolls

Im Folgenden werden die technischen Spezifikationen dargelegt. Dazu werden zunächst die, innerhalb der Nachrichten verwendeten, Datentypen vorgestellt und erläutert. Anschließend werden Struktur und Aufbau der Nachrichten beschrieben.

3.1.4.1 Datentypen

Um den definierten Aufbau der Nachrichten erläutern zu können, müssen zunächst deren Bestandteile, die Datentypen, vorgestellt werden. Dazu werden im Folgenden zuerst die verwendeten nicht trivialen nativen Datentypen definiert und anschließend die komplexeren Datentypen wie, z.B. der Header der Nachrichten.

3.1.4.1.1 Nativen Datentypen

Für die Zusammensetzung der Nachrichten werden zwei nicht triviale native Datentypen, string und ushort, verwendet. Innerhalb dieses Protokolls sind diese wie folgt definiert.

- **string** - Ein String repräsentiert eine UTF8 kodierte Zeichenkette mit einer beliebigen Länge. Das Ende dieser Zeichenkette wird durch ein Null Byte dargestellt.
- **ushort** - Der Datentyp ushort stellt eine 2-Byte große, positive Ganzzahl dar, die im Little-Endian-Format übertragen wird.

3.1.4.1.2 Header

Der Header ist Bestandteil jeder Nachricht und enthält Information über den Sender, die Zugehörigkeit und die Art der Nachricht. Die Struktur des Headers ist wie folgt aufgebaut.

```
struct {
    byte   ProtocolVersion
    byte   MessageType
    byte[] JobID              //128-bit
    string WorldName          //null-terminated string
    string SenderName         //null-terminated string
    string HostName           //null-terminated string
    ushort CertificateLength  //16-bit
    byte[] CertificateData
    ushort SignatureLength    //16-bit
    byte[] SignatureData
    ushort NumberOfExtensions//16-bit
    Extension[] Extensions
} Header;
```

Die *ProtocolVersion* gibt die Version des verwendeten Protokolls an.

Der *MessagesType* gibt an, um welche Art von Nachricht es sich dabei handelt. Jede Nachricht hat dementsprechend einen eigenen *MessageType*, der als Zahl zwischen 1 und 255 angegeben wird. Der MessageType 0 ist für Testzwecke reserviert.

Die *JobID* ist ein 16-Byte großer Wert zur Identifizierung einer Aufgabe im Netzwerk. Da Aufgaben im Netzwerk nur mit einer eindeutigen ID erstellt werden können und ein Peer auch ohne Kenntnis aller Aufgaben eine neue Aufgabe erstellen kann, wird die *JobID* zufällig gewählt werden. Zudem ist die *JobID* 0 für einige Management-Nachrichten reserviert und darf nicht für eine Aufgabe benutzt werden.

Der *WorldName* ist eine nullterminierte Zeichenkette, der die Welt einer Nachricht angibt. Ein Peer kann Nachrichten aus Welten, in denen er keine Aufgaben bearbeitet, ignorieren. Der *WorldName* „*“ ist für einige Management-Nachrichten reserviert und darf nicht ignoriert werden.

SenderName und *HostName* sind beides nullterminierte Zeichenketten und dienen zur Identifizierung des Absenders. Während der Sendername den Benutzer identifiziert und aus dem Zertifikat des Benutzers entnommen wird, repräsentiert der Hostname den Absenderrechner. Der Hostname kann frei gewählt werden.

Zur Erfüllung der geforderten Nachweisbarkeit wird jede Nachricht signiert. Für das entsprechende Zertifikat ist das Feld *CertificateData* vorgesehen und für die Signatur das Feld *SignatureData*. Für das Signieren einer Nachricht wird das Signatur-Feld ignoriert.

Damit unterschiedliche Versionen des Protokolls untereinander kompatibel bleiben, bietet das Protokoll die Möglichkeit, durch, die in Kapitel 3.1.4.1.3 beschriebenen, Extensions um bestimmte Aspekte erweitert zu werden. Über dem 16-byte

Integer Wert *NumberOfExtensions* wird die Anzahl der vorhandenen Extensions angegeben und das *Extensions*-Feld enthält diese.

3.1.4.1.3 Extension

Extensions sind Key-Vaule-Paare und stellen Protokollerweiterungen dar. Extension ermöglichen es, ebenfalls verschiedene Versionen des Protokolls untereinander kompatibel zu machen. Zukünftige Änderungen am Protokoll werden nur noch über Extensions stattfinden. Dadurch ist das Protokoll sowohl abwärtskompatibel als auch aufwärtskompatibel. Der Aufbau der Extension-Struktur ist im Folgenden dargelegt.

```
struct {
        string Key          //null-terminated string
        ushort ValueLength //16-bit
        byte[] Value
} Extension;
```

3.1.4.1.4 NetworkJobMetaData

Für die Übertragung von Aufgaben im Netzwerk wurden, aufgrund der geringen Paketgröße von UDP und dem Bedarf, möglichst große Zusatzinformationen zu einer Aufgabe austauschen zu können, zwei Strukturen angelegt. Die im Folgenden beschriebene NetworkJobMetaData-Struktur und der in 3.1.4.1.5 dargestellten NetworkJobPayload-Strucktur.

```
struct {
    byte[] JobID                  //128-bit
    string Creator                //null-terminated string
    ushort AlgorithmInformationLength //16-bit
    byte[] AlgorithmInformation
    string JobName                //null-terminated string
    string JobType                //null-terminated string
    ushort JobDescriptionLength //16-bit
    byte[] JobDescription
} NetworkJobMetaData;
```

Die *JobID* ist, wie in 3.1.4.1.2 beschrieben, ein zufällig gewählter 16-Byte Wert zur Identifizierung der Aufgabe.

Die nullterminierte Zeichenkette *Creator* enthält den im Zertifikat des Aufgabenerstellers angegeben Namen.

Das Feld *AlgorithmInformation* beinhaltet Informationen, die zur Initialisierung der Verteilungsalgorithmen benötigt werden.

Die Felder *JobName*, *JobType* und *JobDescription* können beliebig gewählt werden und ermöglichen es, Aufgaben für den Anwender zu benennen, zu Kategorisierungen und zu beschreiben.

3.1.4.1.5 NetworkJobPayload

Die im Folgenden gegebenen NetworkJobPayload-Struktur dient dazu, nur durch die Größe beschränkte, Zusatzinformationen zu einer Aufgabe auszutauschen.

```
1 struct {
2     ushort JobPayloadLength
3     byte[] JobPayload
4 } NetworkJobPayload;
```

3.1.4.2 Nachrichten

Die Identifizierung der Nachrichten erfolgt durch einen eindeutigen MessageType, der in den Strukturen der Nachrichten angegeben ist. Da das Protokoll broadcastfähig bzw. multicastfähig sein soll, ist jede Nachricht auf die maximale Größe eines UDP-Paketes von 2^{16} Bytes (64 Kb) begrenzt.

3.1.4.2.1 CreateNetworkJob-Nachricht

Die CreateNetworkJob-Nachricht besteht sowohl aus der NetworkJob-MetaData-Struktur als auch aus der NetworkJobPayload-Struktur für die zu erstellende Aufgabe. Die Welt im Header dieser Nachricht ist abhängig von der Welt der zu erstellende Aufgabe. Der *MessageType* ist drei. Der Aufbau der Nachricht ist im Folgenden gegeben.

```
1 struct {
2         Header{ MessageType = 3};
3         NetworkJobMetaData;
4         NetworkJobPayload;
5 } CreateNetworkJob
```

3.1.4.2.2 DeleteNetworkJob-Nachricht

Die DeleteNetworkJob-Nachricht besteht, wie viele Managementnachrichten, nur aus der Header-Struktur. Der *MessageType* wird bei dieser Nachricht auf zwölf gesetzt. Die im Header angegebene Welt und *JobID* werden entsprechend der Aufgabe gesetzt. Die gesamte Struktur der Nachricht wird nachfolgend gegeben.

```
struct {
        Header {MessageType = 12};
} DeleteNetworkJob;
```

3.1.4.2.3 RequestJobList-Nachricht

Die RequestJobList-Nachricht besteht ebenfalls nur aus der Header-Struktur. Die Welt des Headers gibt an, für welche Welt die Aufgabenliste angefragt wird. Die *JobID* wird auf null gesetzt. Die Struktur selbst ist im Folgenden gegeben.

```
struct {
        Header {MessageType = 4};
} RequestJobList;
```

3.1.4.2.4 ResponseJobList-Nachricht

Die ResponseJobList-Nachricht enthält die Metadaten zu Aufgaben innerhalb der im Header angegeben Welt. Dabei gibt *NumberOfJobs* an, wie viele Aufgaben übermittelt werden. Der *MessageType* dieser Nachricht ist fünf. Die Nachricht ist wie folgt ausgebaut.

```
struct {
        Header {MessageType = 5};
        ushort NumberOfJobs
        NetworkJobMetaData[] JobList
} ResponseJobList;
```

3.1.4.2.5 RequestJobDetails-Nachricht

Die Welt und die *JobID* im Header der RequestJobDetails-Nachricht werden auf die Daten der entsprechenden Aufgabe gesetzt. Der Aufbau der Nachricht ist im Folgenden gegeben.

```
struct {
        Header {MessageType = 6};
} RequestJobDetails;
```

3.1.4.2.6 ResponseJobDetails-Nachricht

Innerhalb der RequestJobDetails-Nachricht sind die Zusatzinformationen zu der im Header angegeben Aufgabe in Form einer NetworkJobPayload-Struktur angegeben. Die Nachrichtenstruktur ist nachfolgend gegeben.

```
struct {
        Header {MessageType = 7};
        NetworkJobPayload;
} ResponseJobDetails;
```

3.1.4.2.7 RequestWorldList-Nachricht

Die RequestWorldList-Nachricht besteht nur aus der Header Struktur. Innerhalb dieser wird die *JobID* auf null und die *World* auf „*“ gesetzt und muss somit von allen Peers beachtet werden. Der Aufbau der RequestWorldList-Nachrichten sieht wie folgt aus.

```
struct {
        Header {MessageType = 8, World = "*"};
} RequestWorldList;
```

3.1.4.2.8 ResponseWorldList-Nachricht

Die ResponseWorldList-Nachricht ist folgendermaßen aufgebaut.

```
struct {
        Header {MessageType = 9, World = "*"};
        ushort   NumberOfWorlds
        string[] WorldNames
} ResponseWorldList;
```

Die Weltnamen werden als Liste von Strings übermittelt. Zusätzlich gibt das Feld *NumberOfWorlds* die Anzahl der Welten an. Der *MessageType* für diese Nachricht wurde auf neun festgelegt. Wie bei der RequestWorldList-Nachricht wird auch hier der Weltname auf „*“ gesetzt.

3.1.4.2.9 JoinNetworkJob-Nachricht

Die Welt und die *JobID* für diese Nachricht müssen mit denen der Aufgabe übereinstimmen, der beigetreten werden soll. Dabei hat die Nachricht selbst den *MessageType* zehn und ist im Folgenden gegeben.

```
struct {
        Header {MessageType = 10};
} JoinNetworkJob;
```

3.1.4.2.10 PropagateState-Nachricht

Der Aufbau der Nachricht im folgenden Textabschnitt definiert.

```
struct {
        Header {MessageType = 11};
        byte[] StateDataLength
        byte[] StateData
        byte[] ResultDataLength
        byte[] ResultData
} PropagateState
```

Der *MessageType* im Header dieser Nachricht ist elf. Auch bei dieser Nachricht stimmen die *JobID* und die Welt im Header mit denen der Aufgabe überein. Innerhalb der *StateData* werden die für den Verteilungsalgorithmus notwendigen Zustandsinformationen (z.B. Epoche) übermittelt. Das Feld *ResultData* beinhaltet die aktuell besten Teilergebnisse der Aufgabe.

3.2 VoluntLib

In diesem Abschnitt wird die Architektur und Struktur der, im Rahmen der vorliegenden Bachelorarbeit entwickelten, Volunteer-Computing-Bibliothek „VoluntLib“ vorgestellt.

3.2.1 Anforderungen an die Bibliothek

Die Hauptanforderung an die zu entwickelnde Bibliothek ist die Implementierung des in 3.1.4.2 gezeigten Protokolls. Die Bibliothek soll es ermöglichen generische Aufgaben innerhalb eines Peer-to-Peer Netzwerke zu berechnen. Neben dieser Anforderung wurden im Rahmen der Bachelorarbeit noch weitere Anforderungen an die Bibliothek gestellt. Diese sind nachfolgend aufgezählt.

B1 **IPv4 und IPv6** – Die Bibliothek soll sowohl unter IPv4 als auch unter IPv6 funktionsfähig sein.

B2 **Persistenz** – Die Bibliothek soll aktuelle Berechnungszustände und Aufgaben auf der Festplatte persistieren.

B3 **Netzwerkübergreifende Kommunikation** – Die Bibliothek soll neben der Kommunikation im lokalen Netzwerk auch in der Lage sein mit einem entfernten Netzwerk zu kommunizierten.

3.2.2 Architektur der Bibliothek

Die Bibliothek lässt sich grob in drei Schichten einteilen, welche nach außen hin durch einen Fassade repräsentiert und gesteuert werden können. Eine Übersicht der Komponenten und der entsprechenden Verantwortlichkeiten ergibt sich aus dem in Abbildung 3.7 gezeigtem FMC (Fundamental Modeling Concepts) Diagram. In Folgenden werden die Schichten detailliert vorgestellt.

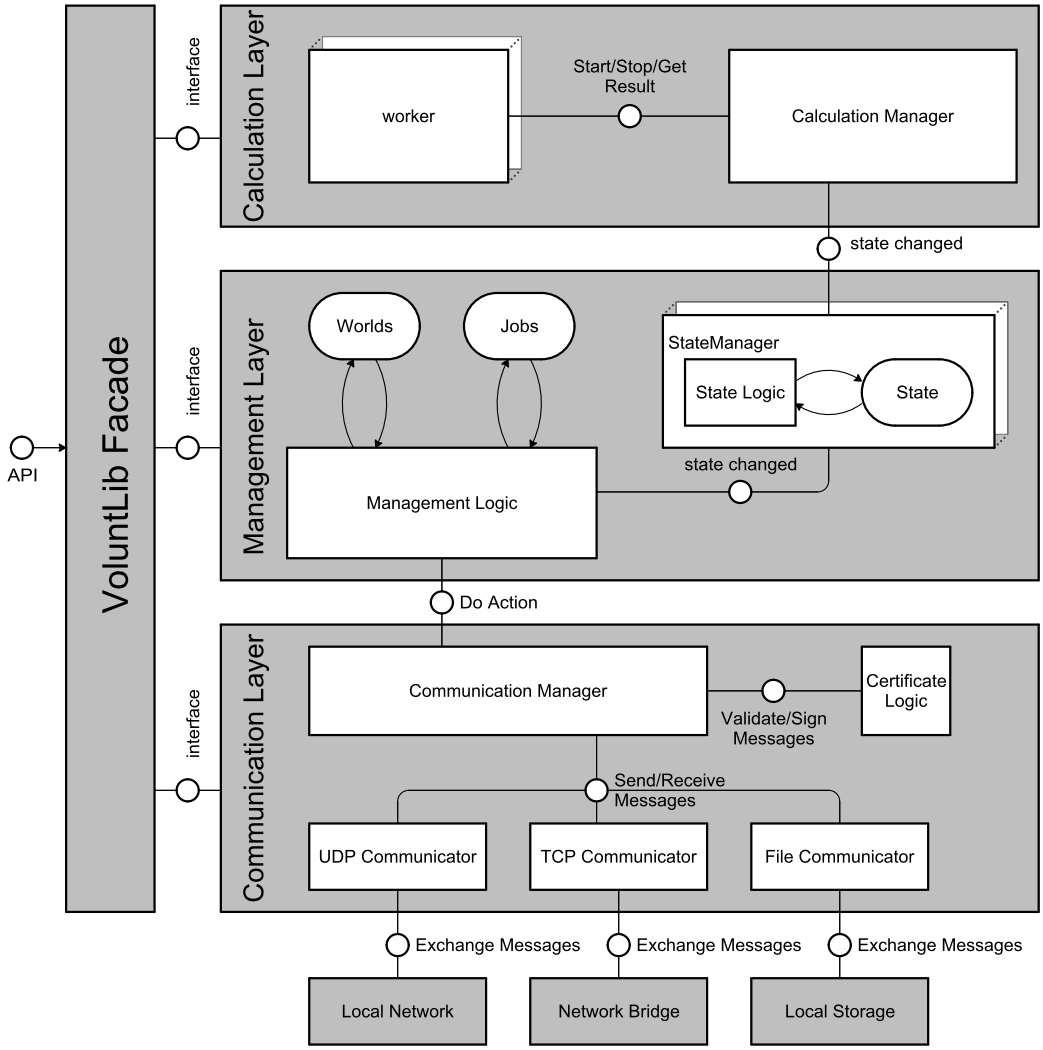

Abbildung 3.7: VoluntLib als FMC Diagram

3.2.2.1 Kommunikationsschicht

Die Kommunikationsschicht ist die unterste Schicht von VoluntLib. Sie ist mit der nächst höheren Schicht, der Verwaltungsschicht, verbunden und besitzt die nachfolgenden Verantwortlichkeiten.

- **Versenden von Nachrichten** – Hierbei wird die Kommunikationsschicht von der Managementschicht mit einer bestimmten Aktion beauftragt. Ein Beispiel für eine Aktion ist, das Erstellen einer übergebenen Aufgabe in einer bestimmten Kommunikationsdomäne. Die Kommunikationsschicht ist für das Erstellen der entsprechenden Nachricht, die die übergebenen Daten enthält, zuständig. Anschließend wird diese von der Kommunikationsschicht signiert, serialisiert und an die angegebene Domäne versendet.
- **Empfangen von Nachrichten** – Hierbei agiert die Kommunikationsschicht als ein Verteiler (*dispatcher*) für die von den Domänen empfangenen Nachrichten. Eine Nachricht wird zunächst deserialisiert und verifiziert. Anschließend wird eine der Nachricht entsprechende Aktion an die Managementschicht weitergeleitet.
- **Signieren und Verifizieren** – Aus den Verantwortlichkeiten des Sendens und des Empfangens von Nachrichten ergibt sich die dritte Aufgabe: Das Signieren jeder aus ausgehenden Nachricht und das Verifizieren jeder empfangenen Nachricht.

Alle, für das Signieren und das Verifizieren der Nachrichten benötigten, Operationen sind in einer eigenen ständigen Komponente (*Certificate Logic*) innerhalb der Kommunikationsschicht gebündelt. Das soll neben einer klareren Struktur auch eine Überprüfung der verwendeten kryptologischen Verfahren erleichtern und somit das Vertrauen in die Implementierung erhöhen.

Die Kommunikation mit den oben angesprochenen Domänen wird mit sogenannten Kommunikatoren umgesetzt. Diese besitzen die Logik, Nachrichten mit ihrer Domäne austauschen zu können. Aktuell gibt es drei unterschiedliche Domänen: Die Broadcast-Domäne, die NetworkBridge-Domäne und die Persistenz-Domäne. Diese Architektur hat den Vorteil, dass neue Kommunikationsdomänen möglichst einfach in die existierende Struktur eingebunden werden können. Es folgt eine Übersicht der einzelnen Kommunikatoren.

3.2.2.1.1 Multicast-Kommunikator

Als Hauptkommunikationskanal verwendet VoluntLib eine Multicastgruppe. Dies hat den Vorteil, dass mehrere Instanzen der Bibliothek im gleichen Netzwerk unabhängig voneinander aktiv sein können. Es macht die Bibliothek darüber hinaus auch unabhängig von der verwendeten IP-Version, da Multicast, im Gegensatz zu Broadcast, sowohl in IPv4 als auch in IPv6 unterstützt wird.

Der Multicast-Kommunikator dient zur Kommunikation mit einer konfigurierbaren Multicastgruppe. Die Kommunikation selbst geschieht über UDP-Pakete und ähnelt im Optimalfall der in Abbildung 2.3 gezeigten Netzwerkstruktur.

3.2.2.1.2 Networkbridge-Kommunikator

Der Networkbridge-Kommunikator dient zum Austausch von Nachrichten über eine Netzwerkbrücke. Das Konzept der Netzwerkbrücken wurde aus zwei Gründen eingeführt. Zum einen da nicht gewährleistet ist, dass alle Router zwischen zwei Peers Multicast-Nachrichten weiterleiten und zum anderen können bei einem weltweiten Fluten innerhalb einer Multicast-Gruppe, ab einem gewissen Punkt Skalierungsprobleme auftreten. Netzwerkbrücken dienen daher sowohl als Aggregierungspunkt für alle Nachrichten eines Netzwerkes als auch als Tunnel zwischen zwei Netzwerken (Siehe Abbildung 3.8). Die Nachrichten werden dabei bidirektional über einen TCP-Stream ausgetauscht. Eine Verbindung wird terminiert, sobald fünf Sekunden keine Nachricht mehr übertragen wurde.

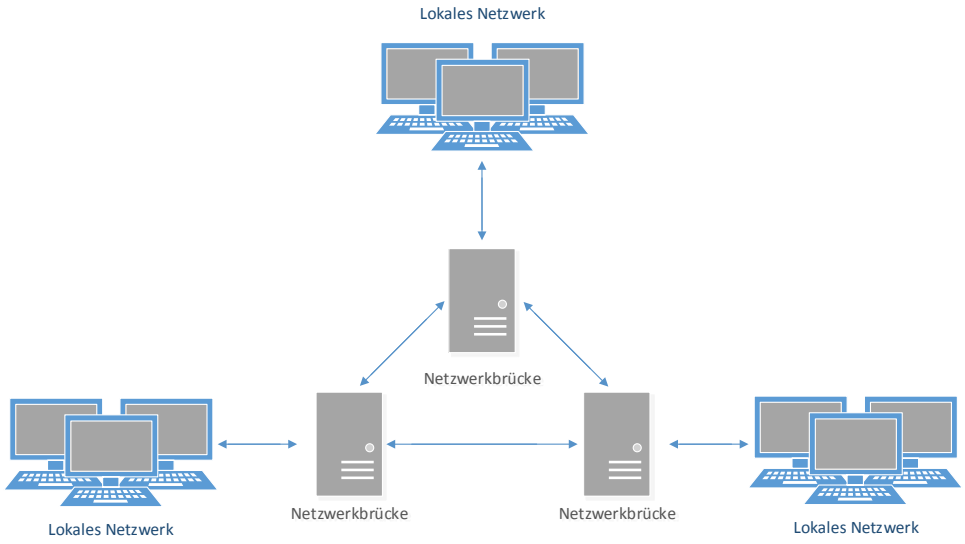

Abbildung 3.8: Peer-to-Peer Netzwerkbrücken

Es gibt zwei Arten von Networkbridge-Kommunikatoren: aktive und passive. Eine aktive Netzwerkbrücke kann initial Verbindungen zu einer passiven Netzwerkbrücke aufbauen. Die passive Brücke muss, im Gegensatz zu den Aktiven, offen über das Internet ansprechbar sein. Ein Teilnehmer kann gleichzeitig beliebig viele aktive Netzwerkbrücken betreiben, jedoch nur eine passive. Durch dies Vorgehensweise können, wie in Abbildung 3.9 zu sehen, offen über das Internet erreichbare Peers anderen Peers als Austauschplattform dienen.

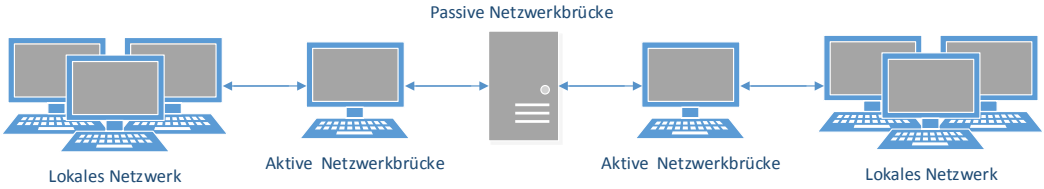

Abbildung 3.9: Netzwerkbrücken verbinden Netzwerke

3.2.2.1.3 Persistenz-Kommunikator

Eine andere Art des Kommunikators stellt der Persistenz-Kommunikator dar. Da hier nicht mit anderen Peers kommuniziert wird, sondern mit einem lokalen Speicher. Die Aufgabe des Persistenz-Kommunikator besteht darin, dauerhaft alle Zustände des Netzwerkes zu persistieren, sodass auch nach einem kompletten Ausfall aller Teilnehmer des Netzwerkes die Zustände gesichert sind. Ebenfalls soll der Persistenz-Kommunikator beim Start eines Teilnehmers die zuletzt gespeicherten Zustände wiederherstellen können.

Die Persistierung soll dadurch erreicht werden, dass alle Nachrichten, die das Erstellen, Löschen oder Berechnen einer Aufgabe betreffen, parallel zum Versenden auch in eine XML-Datei geschrieben werden. Dabei sollen die Nachrichten entsprechend des in 3.1.4.2 beschriebenen Protokolls als Bytestream serialisiert werden. Dieser Bytestream wird dann als Base64-kodierte Zeichenkette in einer XML-Datei gespeichert. Beim Starten der Bibliothek können diese Nachrichten ausgelesen und an die Managementschicht weitergereicht werden.

3.2.2.2 Verwaltungsschicht

Die Verwaltungsebene ist die mittlere der drei Schichten. Sie ist sowohl mit der Kommunikationsschicht verbunden als auch mit der Berechnungsschicht.

Eine zentrale Aufgabe der Verwaltungsschicht ist es, die von der Kommunikationsschicht übergebenen Aktionen zu bearbeiten. Diese Aufgabe übernimmt die Management-Logic-Komponente. Sie sorgt dafür, dass empfangene und selbst erstellte Aufgaben sowie deren Zusatzinformationen in entsprechenden Datenstrukturen gespeichert werden. Auch die existierenden Welten werden von ihr verwaltet. Auf eine übergebene Aktion wird dabei analog zum Protokoll reagiert (siehe 3.1).

Für jede Aufgabe des Netzwerks existiert ein StateManager. Dieser ist für die Verwaltung der Berechnungszustände zuständig. Jeder StateManager implementiert dabei einen der in 2.5 beschriebenen Verteilungsalgorithmen. Somit ist er dazu in der Lage, die Relation (Teilmenge, Obermenge, Differenzmenge) zwischen zwei Zuständen zu berechnen. Die Management-Logic benötigt diese Relation, um auf eine eingehende PropagateState-Nachrichten, wie in 3.1.4.2.10 beschrieben,

reagieren zu können. Zusätzlich ist der StateManager dazu imstande, eine lokale Änderung des Zustands eventbasiert, z.B. durch eine abgeschlossene Berechnung, über die Management-Logic an die Kommunikationsschicht weiterzugeben. Über ein weiteres Eventsystem ist es ebenfalls möglich, der Berechnungsschicht eine externe Änderung des Zustandes mitzuteilen. Da die Berechnungsschicht unabhängig vom Verteilungsalgorithmus ist, werden die Verteilungszustände (z.B. Epoche und Position eines freien Bits in der Bitmaske) vorher so umgewandelt, dass sie eine eindeutige Teilaufgabe des gesamten Berechnungsraum angeben.

Da die Management-Logic dabei die Domäne, aus der eine Nachricht gekommen ist, kennt, kann sie anhand dieser unterschiedlich reagieren. Hierbei wird speziell die Kommunikation innerhalb einer Multicast-Domäne und zwischen zwei Netzwerkbrücken anhand des im Protokoll festgelegten Verhaltens unterschieden (siehe 3.1.3). Für die Persistenz-Domäne, also der Kommunikation mit einer lokalen XML-Datei, ist auf der Verwaltungsschicht kein besonderes Verhalten mehr notwendig.

3.2.2.3 Berechnungsschicht

Die Berechnungsschicht verwaltet eine konfigurierbare Anzahl an Threads (Worker), welche die eigentliche Berechnung der Aufgabe durchführen. Sie ist mit einem StateManger der Verwaltungsschicht verbunden und wird von diesem über externe Änderungen des Zustandes informiert. Zudem kann sie Teilaufgaben zur Berechnung anfordern und die Ergebnisse für eine Teilaufgabe an den StateManger übergeben.

Beim Start einer Berechnung werden zunächst die konfigurierte Anzahl an Workern gestartet. Sind diese mit der Berechnung fertig, werden die Ergebnisse zurückgegeben. Weiterhin wird bei jeder vom StateManager gemeldeten Änderung am Zustand überprüft, ob eine Teilaufgabe als berechnet markiert wurde, an der grade lokal ebenfalls gerechnet wird. Ist dies der Fall, wird der lokale Worker gestoppt und mit einer neuen Teilaufgabe beauftragt. Die Logik für das Berechnen einer Aufgabe wird vom Anwender der Bibliothek übergeben.

Für eine aktiven Teilnahme muss der Peer, laut Protokoll, imstande sein, zwei Zustände, sowie die dazugehörigen Ergebnisse, zu vereinen. Die Zustände werden dafür direkt von den jeweiligen StateManagern zusammengefügt. Für das Zusammenfügen der Ergebnislisten ist die Berechnungsschicht verantwortlich. Die dafür notwendige Logik muss vom Anwender implementiert werden.

3.2.2.4 Fassade

Die Fassade von VoluntLib bündelt die Funktionalitäten der einzelnen Schichten mit der Absicht, eine klare Schnittstelle (API) für den Anwender bereitzustellen. Sie ist ebenfalls dafür verantwortlich, beim Anlegen einer VoluntLib Instanz, die oben beschriebenen Schichten zu initialisieren und untereinander zu verknüpfen. Der Anwender kann dabei eine Vielzahl an Konfigurationen vornehmen, oder die voreingestellten Werte verwenden. Eine Übersicht, der Methoden findet sich in 4.1.2.

4 Implementierung

Dieses Kapitel beschreibt die Implementierung der in 3.2 konzeptionell beschriebenen Bibliothek. Anschließend wird der, für das in 3.1 beschriebene, Netzwerkprotokoll entwickelte Wireshark-Dissector vorgestellt. Zu Letzt werden drei Referenzanwendungen vorgestellt, die während der Entwickelung von VoluntLib entstanden sind. Zunächst werden allerdings die angewandten Entwicklungsmethodiken vorgestellt.

4.1 VoluntLib

Die Programmbibliothek VoluntLib wurde in C# mit dem .NET Framework 4.0 entwickelt. Das .NET-Framework ermöglicht es, dem Anwender für die Implementierung der Berechungslogik hardwarenahe und Performanceoptimierte Bibliotheken zu verwenden. Diese können in unterschiedlichen Programmiersprachen entwickelt worden sein (u.A. C/C++, ASM). Im Folgenden wird zunächst auf die Implementierung der Schichten eingegangen und die verwendeten Schnittstellen der Schichten beschrieben. Im Anschluss wird die Schnittstelle der Fassade bzw. die API vorgestellt.

4.1.1 Implementierung der Schichten

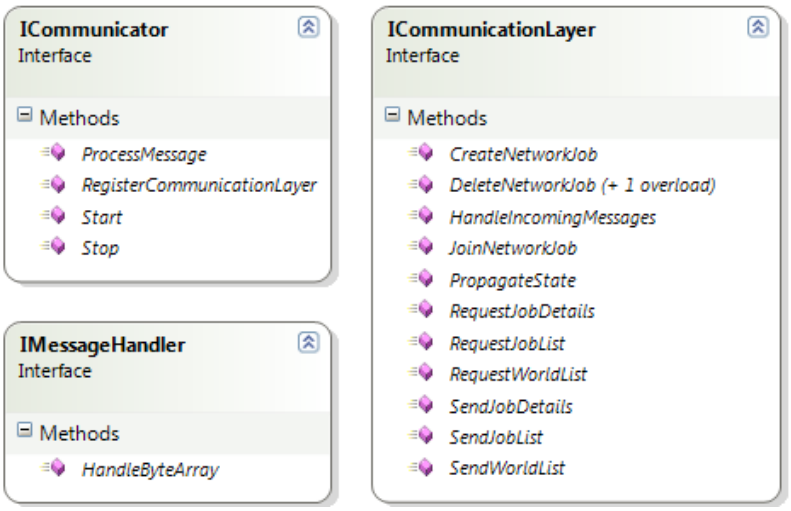

Abbildung 4.1: CommunicationLayer

Abbildung 4.1 zeigt die innerhalb der Kommunikationsschicht verwendeten Schnittstellen. Das Hauptobjekt dieser Schicht implementiert die **ICommunicationLayer**-Schnittstelle. Über diese Schnittstelle kommuniziert sowohl die Verwaltungsschicht als auch die Kommunikatoren mit dem Hauptobjekt. Jeder Kommunikator wird durch die **ICommunicator**-Schnittstelle repräsentiert. Bei einem am Hauptobjekt eingehenden Auftrag (z.B. über die **CreateNetworkJob**-Methode) wird von diesem aus den übergebenen Daten eine Nachricht nach dem im Kapitel 3.1 beschriebenen Protokoll erstellt. Die Nachricht wird anschließend mittels der **ProcessMessage**-Methode an den jeweiligen Kommunikator weitergeleitet. Bei eingehen Nachrichten werden diese von den Kommunikatoren über die **HandleIncomingMessage**-Methode an das Hauptobjekt weitergegeben. Für das Abarbeiten der eingehenden Nachrichten stehen dem Hauptobjekt eine entsprechende Anzahl an **IMessageHandler**-Objekten zur Verfügung. Jedes dieser Objekte beinhaltet die Logik zum Dekodieren, Validieren und Weiterleiten für einen Nachrichten-Typen.

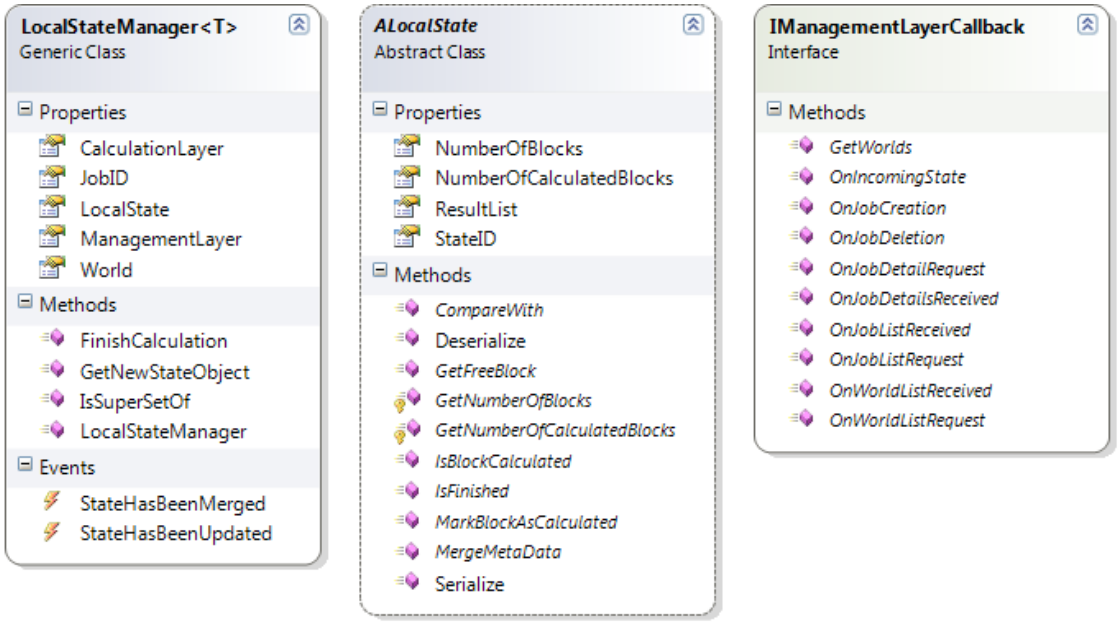

Abbildung 4.2: CommunicationLayer und ManagementLayer

Die relevanten Schnittstellen und abstrakten Klassen der Verwaltungsschicht sind in Abbildung 4.2 dargestellt. Das Hauptobjekt dieser Schicht implementiert die **IManagementLayerCallback**-Schnittstelle. Diese Schnittstelle wird von allen IMessageHandler-Objekten der Kommunikationsschicht verwendet, um die eingehenden Nachrichten an die Verwaltungsschicht zu übergeben. Innerhalb der Verwaltungsschicht wird für jede Aufgabe, an der aktiv gearbeitet wird, ein **LocalStateManager**-Objekt angelegt. Dabei ist dieses Objekt generisch an eine Implementierung des **ALocalState** gebunden. Dieser repräsentiert einen Verteilungsalgorithmus und stellt entsprechende Methoden für die Verteilung zur Verfügung. Das LocalStateManager-Objekt verfügt dabei über zwei Events. Mithilfe des **State-**

HasBeen-Updated-Event wird die Berechnungsschicht über eine Änderung am lokalen Zustand informiert. Zusätzlich wird über das **StateHasBeenMerged**-Event das Hauptobjekt über eine Änderung informiert, die an das Netzwerk weitergegeben werden muss.

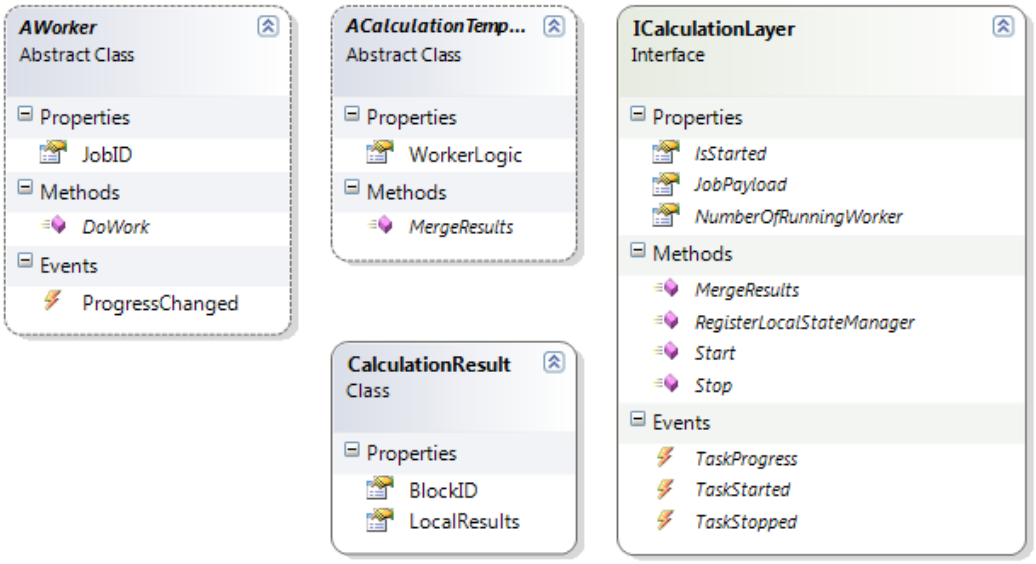

Abbildung 4.3: CalculationLayer, Template und Worker

Die dritte Schicht ist die Berechnungsschicht. Die wichtigsten Schnittstellen sind in Abbildung 4.3 gegeben. Das Hauptobjekt für diese Schicht implementiert die **ICalculationLayer**-Schnittstelle. Dieses Objekt kommuniziert mit der Verwaltungsschicht über einen LocalState-Manager. Der Berechnungsschicht wird, ein vom Anwender implementiertes, **ACalculationTemplate**-Objekt übergeben. Die WorkerLogic des Templates stellt eine Instanz der **AWorker**-Klasse da. Zum Berechnen einer Aufgabe wird die **DoWork**-Methode dieses Objektes ausgeführt. Dabei gibt die DoWork-Methode nach der Berechnung ein **CalculationResult** zurück, das an den LocalState-Manager der Verwaltungsschicht übergeben wird.

4.1.2 Implementierung der Fassade

Die Fassade der Bibliothek wird durch eine Instanz der in Abbildung 4.4 gezeigten Klasse umgesetzt. Nach der Instanzierung der Klasse stehen dem Anwender Funktionen und Eigenschaften zur Konfiguration der verschiedenen Schichten bereit. So ist es beispielswiese möglich, über die **EnablePersistance**-Eigenschaft, die Persistierung der Bibliothek zu aktivieren, oder mit der**AddNetworkBridge**-Methode eine entfernte Netzwerkbrücke hinzuzufügen. Beim Aufruf der **Init**-Methode wird, anhand der zuvor vorgenommenen Konfiguration, die Instanz initialisiert. Dabei exsistiert für jede Eigenschaft der Bibliothek ein Standardwert, damit die Bibliothek auch ohne Konfiguration des Anwenders initialisiert werden kann. Über die

Start-Methode kann, nach der Initialisierung, dem eingestellten Netzwerken beigetreten werden. Ab diesem Zeitpunkt kann der Anwender den vollen Funktionsumfang der Fassade in Anspruch nehmen. Dabei werden die Funktionen meist direkt an die entsprechende Schicht weitergeleitet. Auch die Events werden an den Anwender weitergeleitet.

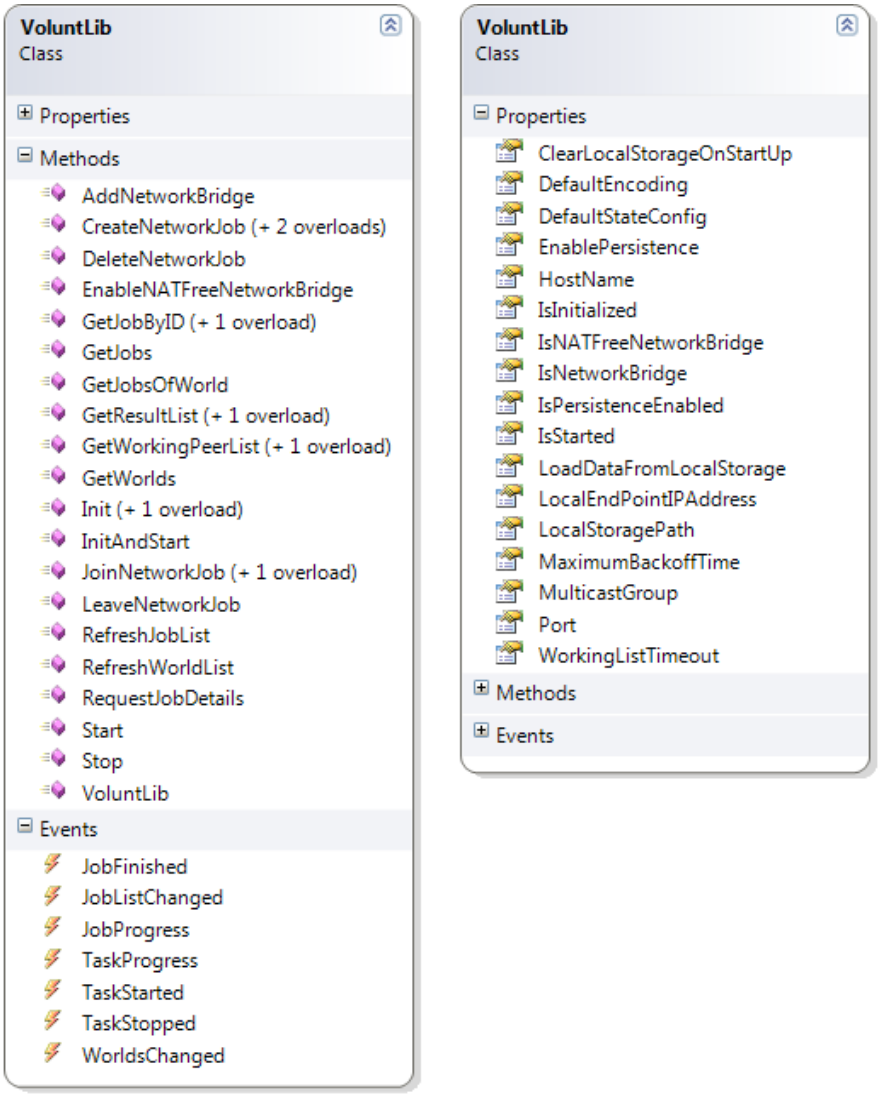

Abbildung 4.4: VoluntLib Fassade

4.2 Wireshark-Dissector

Bei der Entwicklung einer, über das Netzwerk kommunizierenden, Software ist es oft hilfreich, die Kommunikation direkt beobachten zu können. Ein weit verbreitetes Tool zur Netzwerkanalyse ist die OpenSource-Software Wireshark [Wir14a]. Wireshark zeichnet den Datenverkehr der Netzwerkschnittstellen auf und zeigt die

einzelnen Pakete an. Bei bekannten Protokollen oder Pakettypen wird das Paket entsprechend dekodiert und die Daten einzeln dargestellt. Für die Dekodierung sind sogenannte Diessectors (eng: Sezierer) verantwortlich. In der Version 1.10.5 ist Wireshark dazu in der Lage, 1377 verschiedene Protokolle zu dekodieren. Für die Erstellung von weiteren Dissectorn wird eine Schnittstelle, für die Skriptsprache LUA, bereitgestellt [Wir14b]. LUA ist eine imperative Skriptsprache, die aufgrund ihrer Interoperabilität zu anderen Programmiersprachen und ihrer Schnelligkeit häufig für die Erweiterung von Programmen verwendet wird.

Um die Handhabung von VoluntLib weiter zu erleichtern, wurde ein solcher Dissector für Wireshark entwickelt. Dieser wurde als LUA-Script, wie in [Wir14b] beschrieben, umgesetzt und ist imstande, den definierten Paket-Header zu dekodieren. Nach der Installation des Dissector werden automatisch alle UDP-Pakete, die an den Default-Port von VoluntLib gesendet werden, dekodiert.

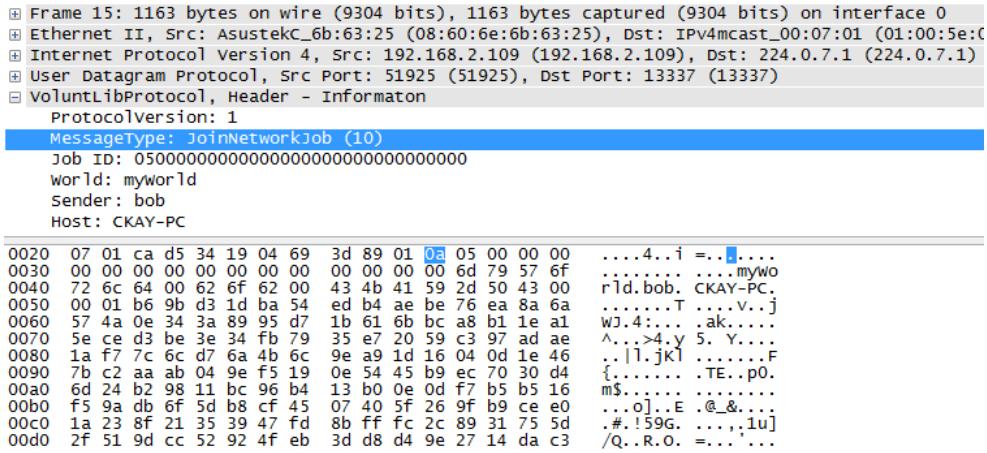

Abbildung 4.5: Wireshark-Dissector für VoluntLib

In der Abbildung 4.5 ist die Detailansicht eines VoluntLib Paketes in Wireshark zu sehen. Im unteren Teil werden die Rohdaten des Paketes als HEX-Werte und als ASCII-Werte angezeigt. Im oberen Bereich sind die von Wireshark dekodierten Daten aufgelistet.

Der Dissectors validiert zudem, dass das, von der Bibliothek „gesprochene“ Protokoll dem in 3.1 beschriebenen entspricht und unabhängig von der .NET Umgebung umgesetzt wurde.

4.3 Beispielanwendungen

In diesem Kapitel werden drei Anwendungen vorgestellt, die im Zuge dieser Arbeit entstanden sind. Zunächst wird eine Demo-Anwendung gezeigt, die die Funktionsweise der Bibliothek aufzeigt. Im Anschluss werden zwei Anwendungen vorgestellt, die VoluntLib benutzen, um reale Aufgaben verteilt zu berechnen. Diese

wären zum einen der HashSearcher, eine Anwendung, die mit Hilfe von VoluntLib einen SHA1-Hash (Secure Hash Algorithm 1) [EJ01] bricht. Zum anderen der KeySearcher, welcher den Raum des Data Encryption Standard (DES) [PUB77] durchsucht.

4.3.1 Demo-Anwendung

Die Demo-Anwendung wurde für die erste Version der Bibliothek entwickelt. Sie soll eine grundlegende Übersicht, der Funktionalitäten geben. Zur Simulierung einer verteilten Berechnung werden drei Worker gestartet. Diese berechnen zunächst für jeden Block eine triviale Aufgabe ($BlockID$ mod 255) und warten anschließend eine Sekunde.

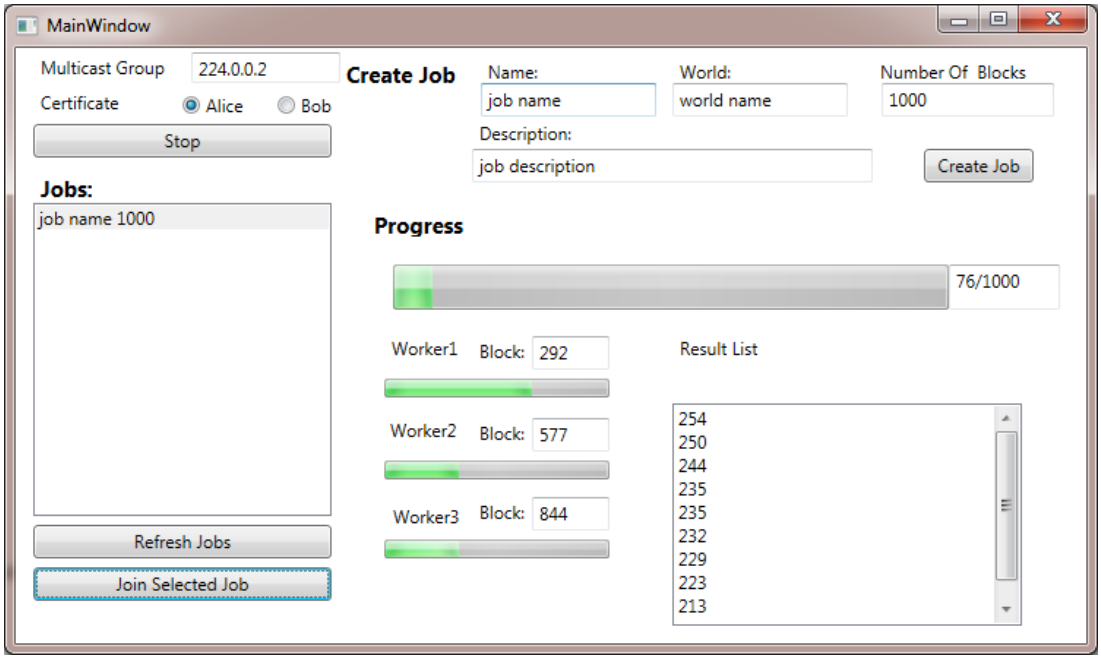

Abbildung 4.6: Demo-Anwendung

In Abbildung 4.6 ist die Oberfläche der Demo-Anwendung zu sehen. Der Benutzer kann dabei im oberen linken Bereich zunächst die Konfigurationen der Bibliothek einstellen (Multicast Gruppe und Zertifikat) und die Bibliothek starten bzw. stoppen. Direkt darunter sind die Aufgaben zu finden, die aktuell im Netzwerk verfügbar sind. Mittels der „Join Selected Job“-Schaltfläche kann die Berechnung an einer Aufgabe gestartet werden. Im oberen rechten Bereich kann über eine entsprechende Eingabemaske eine neue Aufgabe erstellt werden. Der untere rechte Bereich zeigt den Fortschritt der aktuellen Berechnung an. Hierbei gibt es sowohl einen Fortschrittsbalken für die gesamte Berechnung als auch jeweils einen für die drei Worker. Zudem ist hier die aktuelle Ergebnisliste zu sehen.

4.3.2 KeySearcher

Die zweite Beispielanwendung, der KeySearcher, berechnet – im Gegensatz zur Demo-Anwendung – eine reale Aufgabe. Der Keyseacher sucht den Schlüssel zu einem, mit DES verschlüsselten, Ciphertext, wobei 4 Bytes des Schlüssels bekannt sind. Somit muss der Keyseacher einen Raum von 2^{32} durchsuchen. Der zu durchsuchende Raum wurde dabei in 2^{16} Blöcke von je 2^{16} möglichen Schlüsseln aufgeteilt und mit dem Epochen-Verteilungsalgorithmus mit einer Bit-Masken-Breite von 128 Bit durchsucht. Die Bestimmung der zu durchsuchenden Schlüssel erfolgte nach dem folgenden Muster.

$$Key : YY - YY - YY - YY - bb - bb - ** - **$$

Wobei $YY-YY-YY-YY$ gegeben ist, $bb-bb$ die BlockID darstellt und $**-**$ die zu durchsuchenden Schlüssel repräsentiert. Die Auswahl der besten Ergebnisse geschieht über die Entropie des entschlüsselten Textes. Mithilfe der Entropie kann die Zufälligkeit eines Textes bestimmt werden [Sha01]. Unter der Annahme, dass bei der Entschlüsselung mit dem richtigen Schlüssel, ein Text in einer natürlichen Sprache entsteht, würde, aufgrund der erhöhten Redundanz einer solchen Sprache, eine deutlich niedrigere Entropie erzeugt werden als es bei einem falschen Schlüssel der Fall wäre.

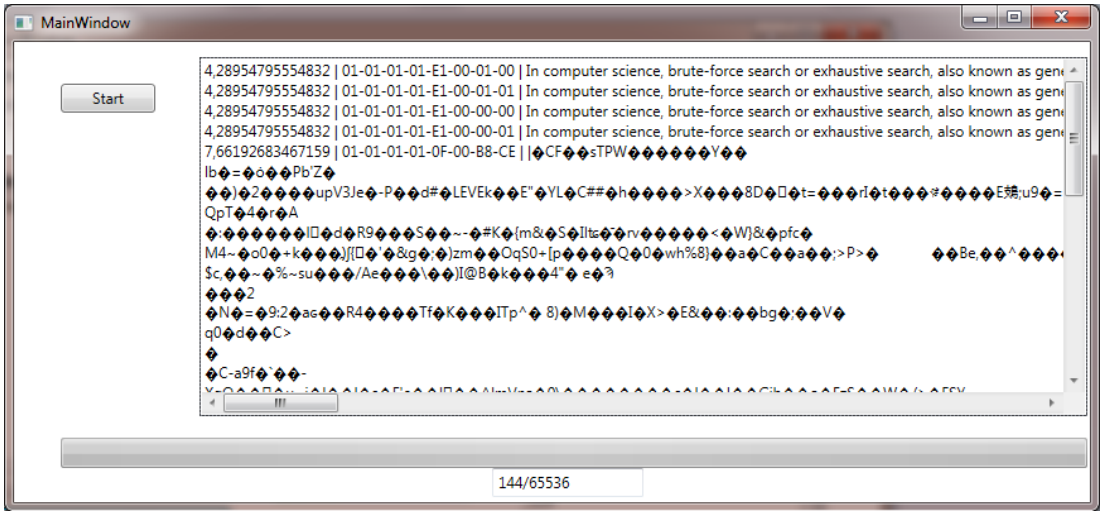

Abbildung 4.7: KeySearcher

Die minimalistische Oberfläche (Siehe Abbildung 4.7) ermöglicht lediglich das Starten der Bibliothek. Im unteren Bereich ist der Gesamt-Fortschrittsbalken zu sehen. Der Großteil der Oberfläche ist für die Ausgabe der Ergebnisse vorgesehen. Diese werden dabei mit ansteigender Entropie angezeigt, sodass die besten Ergebnisse oben stehen. Dabei wird zunächst die Entropie, anschließend der verwendete Schlüssel und der entschlüsselte Text angegeben.

Die Implementierung des KeySearchers wird in den folgenden Textabschnitten gegeben und kommentiert. Der Quelltext wurde für die Darstellung an einigen Stellen modifiziert und ist daher nicht lauffähig.

```
1  //Constructor
2  public MainWindow() {
3    //init GUI
4    InitializeComponent();
5
6    //create instance of VoluntLib
7    lib = new VoluntLib();
8
9    // bind on event for progressbar changes
10   lib.JobProgress += OnLibOnJobProgress;
11
12   //change bitMask width
13   lib.DefaultStateConfig.BitMaskWidth = 128;
14
15   //init
16   var rootCert = new X509Certificate2(Resources.rootCA);
17   var myCert = new X509Certificate2(Resources.alice);
18   lib.Init(rootCert, myCert);
19
20   //start
21   lib.Start();
22 }
23
24 private void Start_Click(object sender, EventArgs e) {
25  //find exsiting jobs in the network
26  lib.RefreshJobList("world_name");
27  Thread.Sleep(1000);
28
29  //get job or create it
30  var job = lib.GetJobByID(5);
31  if (job == null) {
32   lib.CreateNetworkJob("world_name",     //world
33                        "keySearch",      //job name
34                        "ExampleJob",     //job type
35                        "bruteforce DES",//description
36                        new byte[0],      //payload
37                        65536,            //blocks
38                        5);               //jobID
39   Thread.Sleep(1000);
40  }
41
42  //start calculation with 4 worker
```

```
43  lib.JoinNetworkJob(5, new CalcLogic(), 4);
44 }
45
46 //Calculation Logic
47 public class CalcLogic : ACalculationTemplate {
48
49  public CalcLogic() {
50   WorkerLogic = new WorkerLogic();
51  }
52
53  //Merges two resultlists
54  public ... MergeResults(... listA, ... listB) {
55   //concat lists and deserilize items
56   var listC = listB.Concat(listA)
57                    .Select(b => new BestListEntry(b))
58                    .ToList();
59   //sort by entropy
60   listC.Sort();
61
62   //return the top 10 (and serialize items)
63   return listC.Take(10)
64               .Select(entry => entry.Serialize())
65               .ToList();
66  }
67 }
68
69 // class for the worker logic
70 public class WorkerLogic : AWorker {
71  public override CalculationResult DoWork(byte[]
      jobPayload, BigInteger blockID, CancellationToken
      cancelToken) {
72
73  //encrypt plaintext (the encrypted text could also be
      within the job Payload)
74  var plainTextBytes = UTF8.GetBytes("In computer science
      ... attacks any other");
75  var text = Crypto.encryptDES(...);
76
77  // adapt key to blockID
78  var IV = new byte[] {0, 0, 0, 0, 0, 0, 0, 0};
79  var key = new byte[] {1, 1, 1, 1, 0, 0, 0, 0};
80  var keyBlock = BitConverter.GetBytes((ushort) blockID);
81  Array.Copy(keyBlock, 0, key, 4, 2);
82
83  var bestList = new List<BestListEntry>();
```

```
84  //do calculation for keyBlock
85  for (ushort i = 0; i < ushort.MaxValue; i++) {
86   //stop the worker if the Calculation layer demands it
87   cancelToken.ThrowIfCancellationRequested();
88
89   //replacing key-bytes
90   Array.Copy((BitConverter.GetBytes(i)), 0, key, 6, 2);
91
92   //decrypt and calulate entropy
93   var decrypt =
94    Crypto.decryptDES(text, key, IV, text.Length, 1);
95   var entropy =
96    Crypto.calculateEntropy(decrypt, decrypt.Length);
97
98   //add to bestList, if necessary
99   if (bestList.Count <= 10
100   || bestList[10].Entropy < entropy) {
101
102   //copy current key
103   var copyOfKey = new byte[key.Length];
104   Array.Copy(key, copyOfKey, key.Length);
105
106   //add
107   bestList.Add(new BestListEntry {
108    Entropy = entropy,
109    KeyBytes = copyOfKey,
110    Text = decrypt
111   });
112
113   //keep only the top10
114   bestList.Sort();
115   if (bestList.Count > 10) {
116    bestList = bestList.GetRange(0, 10);
117   }
118  }
119 }
120
121 // return results for keyBlock
122 var result = new CalculationResult {
123  BlockID = blockID,
124  LocalResults = bestList.Select(e => e.Serialize())
125                         .ToList()
126 };
127 return result;
128 }
```

4.3.3 Hashsearcher

Mit der ersten Version der Bibliothek, die aufgrund des verwendeten Continuous Delivery Ansatzes früh am Fachgebiet „Angewandte Informationssicherheit“ verwendet werden konnte, wurde von Nils Kopal der HashSearcher entwickelt. Die Anwendung kann im Kontext der Bachelorarbeit sowohl als Praxistest als auch als Usability-Test gesehen werden. Daher wird der HashSeacher im Folgenden kurz vorgestellt.

Der HashSearcher ist eine Konsolenanwendung, die zur Lösung einer MysteryTwister C3 Challenge entwickelt wurde. MysteryTwister C3 ist eine Webseite, die eine Sammlung an kryptographischen Challenges anbietet [MC14b].

Bei der Challenge soll zu einem gegebenen SHA-1 Hashwert der passende Schlüssel gefunden werden[MC14a]. Dabei wird der Suchraum stark eingegrenzt, so dass er relativ schnell vollständig durchsucht werden kann. Der HashSeacher teilt den Suchraum in 2048 Blöcke mit je etwa 2^{21} (2236962) möglichen Schlüsseln. Wenn ein Teilnehmer den richtigen Schlüssel gefunden hat, wird die Anwendung beendet.

5 Evaluation der Bibliothek

In diesem Kapitel wird die entwickelte Bibliothek evaluiert. Dabei wird zunächst die korrekte Funktionsweise der Bibliothek dargelegt und danach ihre Wartbarkeit und Erweiterbarkeit anhand einiger statischer Metriken ausgewertet. Anschließend wird die Effektivität der Parallelisierung evaluiert.

5.1 Validierung der Funktionsweise

Eine Validierung der korrekten Funktionsweise von VoluntLib ist aufgrund der testgetriebenen Entwicklung, die in 2.6.1 beschriebenen wurde, einfach möglich. Die insgesamt 56 Unit-Tests und 9 Acceptance-Tests validieren, wie in Abbildung 5.1 zu sehen ist, insgesamt 93% der Bibliothek (siehe Abbildung 5.1). Die 181 ungetesteten Zeilen sind Debug-Ausgaben für Fehlerfälle. Ein automatisches Testen dieser Zeilen würde lediglich den Umfang der Tests erhöhen, aber nicht zur Validierung der Funktion beitragen.

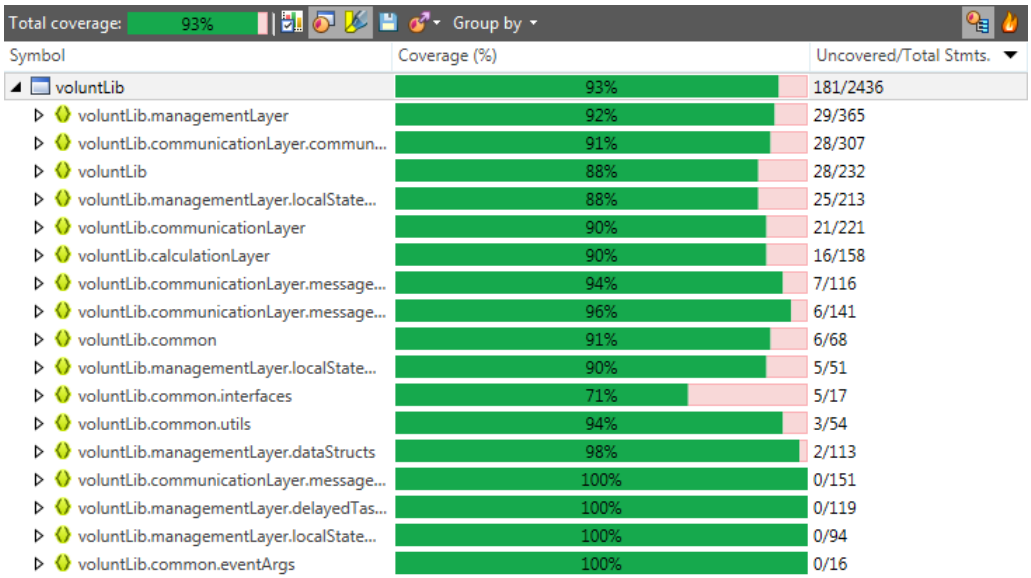

Abbildung 5.1: Testabdeckung von VoluntLib

Die Validierung der Funktionsweise ist – aufgrund der hohen Testabdeckung – durch Ausführen der automatischen Tests möglich. In Abbildung 5.2 ist das Ergebnis der Unit-Tests und Acceptance-Tests zu sehen. Alle Tests sind einwandfrei durchgelaufen und die Funktionsweise damit nachgewiesen.

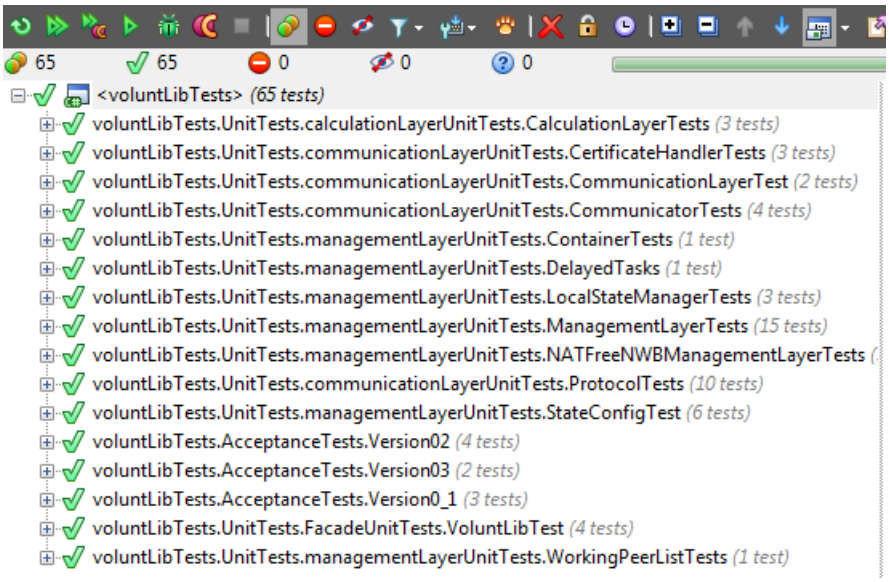

Abbildung 5.2: Testausführung

5.2 Wartbarkeit und Erweiterbarkeit

Der größte Aufwand, innerhalb des Lebenszyklus einer Software, wird für die Wartung und Erweiterung verwendet. Ein guter Softwareentwicklungsprozess wird diesen Aufwand daher so gering wie möglich halten. Neben der in 5.1 beschriebenen Testabdeckung und der durchgehenden Kommentierung der Quellcodes, muss dazu auch der Quellcode selbst einfach und intuitiv gestaltet sein. Um dies objektiv bewerten zu können, werden in der Softwaretechnik traditionell Metriken verwendet. Im Folgenden wird die Wartbarkeit und Erweiterbarkeit anhand von zwei primitiven und einer kombinierten Metrik evaluiert.

Die einfachste Metrik zu einer Software ist die Anzahl der verwendeten Quellcode-Zeilen. VoluntLib umfasst zurzeit **2436 Zeilen** produktiven Quellcode, welcher von 1553 Zeilen Unit Tests und 398 Zeilen Acceptence Tests validiert wird. Die Anzahl der Zeilen beschreibt aber lediglich den Umfang und macht keine Aussage über die Strukturierung bzw. die Komplexität.

Eine weitere Metrik ist die McCabe-Metrik (zyklomatische Komplexität). Sie ermittelt die Anzahl der voneinander unabhängigen Ausführungspfade innerhalb eines Programmstückes. Hierfür werden alle Anweisungen, bei denen exakt zwei Ausführungspfade entstehen (z.B. if, for, while), gezählt [McC76]. Je mehr solche Binärverzweigungen eine Funktion hat, desto mehr mögliche Testfälle müssen beachtet werden. Laut einer Studie des NISTs (National Institute of Standards and Technology) [WMW96] gilt bei Programmstücken, deren zyklomatische Komplexität die Zahl zehn überschreiten, die Wartbarkeit als gefährdet.

Abbildung 5.3 zeigt die Verteilung der zyklomatischen Komplexitäten über alle Funktionen innerhalb von VoluntLib. Wie zu sehen ist, liegt die zyklomatischen

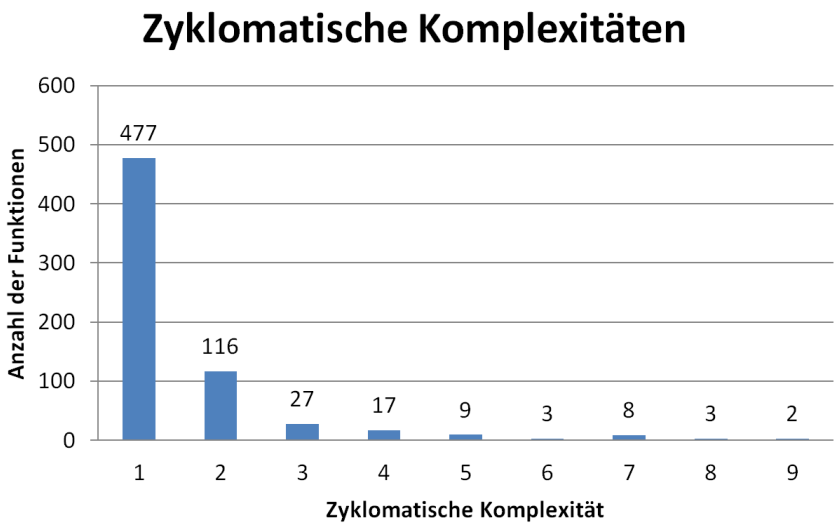

Abbildung 5.3: Verteilung der zyklomatischen Komplexitäten

Komplexität aller Funktionen unterhalb des empfohlenen Maximalwerts von zehn, der Großteil sogar deutlich.

Ein weiterer Indikator für die Wartbarkeit und Erweiterbarkeit ist der Maintainability Index (MI). Der MI kombiniert und gewichtet verschiedene Metriken. Die Formel für den verwendeten MI wurde in [CALO94] definiert und lautet:

$$MI = 171 - 5,2 \cdot ln(HV) - 0,23 \cdot CC - 16,2 \cdot ln(LoC)$$

Dabei gibt LoC die Anzahl der Programmzeilen (Lines of Code), CC die zyklomatische Komplexität und HV das Halstead-Volumen. Das Halstead-Volumen ist eine weitere Metrik, die anhand der verwendeten Operatoren und Operanden und der Programmzeilen die Komplexität des Programmes ermittelt. Laut [CALO94] verfügen Programme ab einem MI von 85 über eine gute Wartbarkeit. Bei einem MI zwischen 85 und 65 verfügt das Programm über eine mäßige Wartbarkeit. Programme mit einem MI unter 65 haben eine niedrige Wartbarkeit und sollten anstatt gewartet zu werden neu geschrieben werden. Die innerhalb dieser Arbeit beschriebene Bibliothek VoluntLib hat einen **MI von 143**. Demnach verfügt sie laut [CALO94] über eine gute Warbarkeit.

5.2.1 Parallelisierung

Die Effektivität einer Parallelisierung wird in der Softwaretechnik traditionell durch den Beschleunigungsfaktor (*Speedup*) gemessen. Der Speedup gibt das Verhältnis von sequenzieller zu paralleler Ausführungszeit eines Programmes an. Er wird wie folgt berechnet:

$$S_p = \frac{T_1}{T_p}$$

Dabei stellt T_1 die sequenzielle Ausführungszeit dar. T_p gibt die Ausführungszeit, unter der Verwendung von p parallelen Einheiten (Peers oder Prozessoren) an. Wenn ein Programm komplett parallelisiert ist, gilt: $S_p = p$. Dies ist aber aufgrund des Amdahlsches Gesetzes nicht möglich [Amd67].

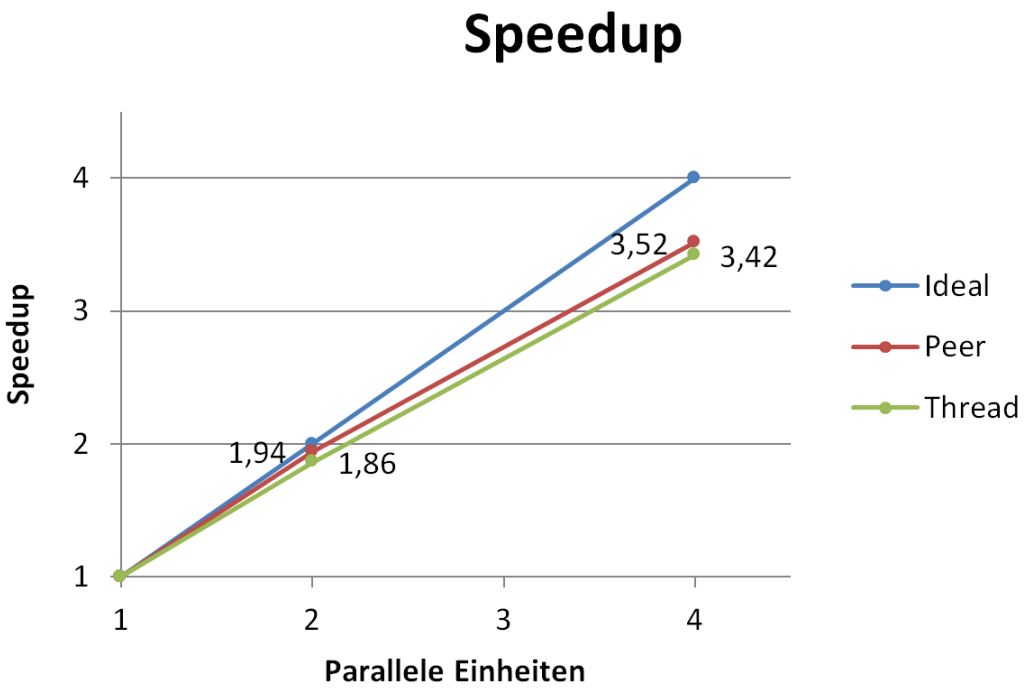

Abbildung 5.4: Speedup von VoluntLib

Die Abbildung 5.4 zeigt den Speedup von VoluntLib. Dabei wird zwischen Peer-Speedup und Thread-Speedup unterschieden. Der Peer-Speedup stellt den Beschleunigungsfaktor, der durch die Anzahl der VoluntLib-Instanzen erzeugt wird, dar. Jede VoluntLib-Instanz ist zusätzlich imstande mehrere Threads parallel arbeiten zu lassen. Daher wird ebenfalls der Thread-Speedup evaluiert.

Beide Beschleunigungsfaktoren sind nah an der idealen Beschleunigung. Die Aufgaben werden von VoluntLib also gut parallelisiert. Der Thread-Speedup ist dabei, erwartungsgemäß, aufgrund von Mutexen in der Algorithmenberechnung etwas geringer.

Die Evaluierung wurde auf einem Intel® Core™ i3 Prozessor durchgeführt. Dieser ist in der Lage vier Threads nativ parallel auszuführen. Aus diesem Grund ist die Evaluierung auf vier parallel Einheiten begrenzt.

6 Verwandte Arbeiten

In diesem Kapitel werden zwei verwandte Arbeiten vorgestellt. Zum einen die verbreitetste Volunteer Computing Plattform (BOINC). Zum anderen ein Volunteer Comptuing Netzwerk, welches ebenfalls auf einem unstrukturierten Peer-to-Peer Netzwerk aufsetzt.

6.1 Boinc

Die BOINC (Berkeley Open Infrastructure for Network Computing) Plattform [And04] wurde an der University of California, Berkeley entwickelt. Dies geschah mit der Absicht, das Aggregieren von Rechenkapazitäten von der wissenschaftlichen Arbeit zu trennen. Die quelloffene Software BOINC kann zum einen von Universitäten eingesetzt werden, um ungenutzte die Rechenzeit, von internen Computer, für eine gemeinsame Berechnung zu nutzen. Zum anderen kann sie auch als Volunteer Computing Plattform verwendet werden. Die Autoren von [And04] bezeichnen dies als „Public-Resource Computing“. Für die Teilnahme an einer Berechung werden zudem „Credits“ an die Voluneers vergeben. Dies soll eine zusätzliche Motivation für eine Teilnahme darstellen.

Bei der Aufgabenverteilung setzt BOINC auf ein zentralisiertes Client-Server-Verfahren. Der Verteilungsserver muss dabei von den Aufgabenbetreibern bereitgestellt und gewartet werden. Laut Angaben der Entwickler wird dafür etwa eine Woche für die Einarbeitung und etwa eine Stunde pro Woche für Pflege und Wartung des Servers benötigt.

Da BOINC Anwendungen nach dem Client-Server-Prinzip verfahren, ist die Architektur in zwei Bereiche zu unterteilen. Die Serverseite besteht aus den folgenden Komponenten.

- Der **Scheduler** verteilt die Arbeitspakte an die Clients und empfängt von diesen die Rückmeldung über den Erfolg oder Misserfolg der Berechnung.
- Der **Dataserver** dient als Vermittler. Die Clients können hier ihre zugeteilten Arbeitspakete herunterladen und die Ergebnisse hochladen.
- Der **Validator** überprüft die eingegangenen Ergebnisse auf Korrektheit. Die Umsetzung ist dabei abhängig von der konkreten Aufgabe.
- Der **Assimilator** aggregiert aus den validen Ergebnissen ein Gesamtergebnis oder reichert diese für weitere Analysen an.

- Der **File-Deleter** löscht die nicht mehr benötigten Teilergebnisse, die bereits vom Assimilator verarbeitet wurden.
- Der **Transistor** überwacht den Fortschritt der Arbeitspakete.

Die Clientseite besteht ganz allgemein aus folgenden Komponenten:

- Der **Core-Client** steuert und überwacht die Anwendung. Zudem kommuniziert er mit den projektspezifischen Scheduler und Datenserver.
- Der **BOINC-Manager** ist eine grafische Oberfläche zur Konfiguration und Überwachung des Core-Clients. Zudem gibt es ein **Boinc-Commandline-Interface**, der selbiges über die Kommandozeile ermöglicht.
- Die **Projektanwendung** stellt die projektspezifische Berechnungsanwendung dar.

Im Vergleich zu VoluntLib, entstehen durch den zentralisierten Ansatz von BOINC höhere Kosten für die Bereitstellung des Verteilungsservers. Durch die Selbstorganisierung der Teilnehmer innerhalb eines VoluntLib-Netzwerks können diese Kosten umgangen werden.

6.2 PPVC

Das **P**eer-to-**P**eer **V**olunteer **C**omputing System (kurz PPVC) stellt ein dezentrales Aufgabenverteilungssystem dar [ZYX09]. Die Teilnehmer sind dabei über ein dezentrales P2P-Netzwerk vernetzt. Ein neuer Teilnehmer kann sich zu einem beliebigen Teilnehmer des Netzwerkes verbinden, um beizutreten. Die Aufgabenverteilung muss dementsprechend ebenfalls dezentral organisiert sein.

Die Aufgaben, die ein PPVC selbstorganisierend verteilen und berechnen kann, müssen parallelisierbar und laut [ZYX09] auch rekursiv sein. Die Verteilung ist in Abbildung 6.1 dargestellt. Teilnehmer A teilt seine Aufgabe T entsprechend der bekannten Nachbarn in gleich große Unteraufgaben $T/3$ und verteilt diese. Jeder der Nachbarn teilt die, ihm zugeordnete, Aufgabe wiederum auf sich und seine Nachbarn auf. So kann Teilnehmer B die Unteraufgabe $T/3$ in zwei Unteraufgaben $T/6$ aufteilen und eine davon an Teilnehmer D weiterreichen. Teilnehmer C kann seine Unteraufgabe nicht aufteilen und weiterreichen, da er keine nicht arbeitenden Teilnehmer kennt. Daher berechnet er $T/3$ selbst. Sobald ein Teilnehmer die Berechnung einer Unteraufgabe abgeschlossen hat, übergibt er das Ergebnis an den Teilnehmer, der ihm die Unteraufgabe zugeteilt hat.

Da Teilnehmer das Netzwerk beliebig Betreten und Verlassen können, muss sich die Aufgabenverteilung entsprechend anpassen. Wenn sich ein neuer Teilnehmer P_{new} mit einem Teilnehmer des Netzwerkes P_o verbindet, teil P_o seine aktuelle Aufgabe in zwei Unteraufgaben und übergibt eine davon an P_{new}. Die zweite Unteraufgabe wird von P_o berechnet.

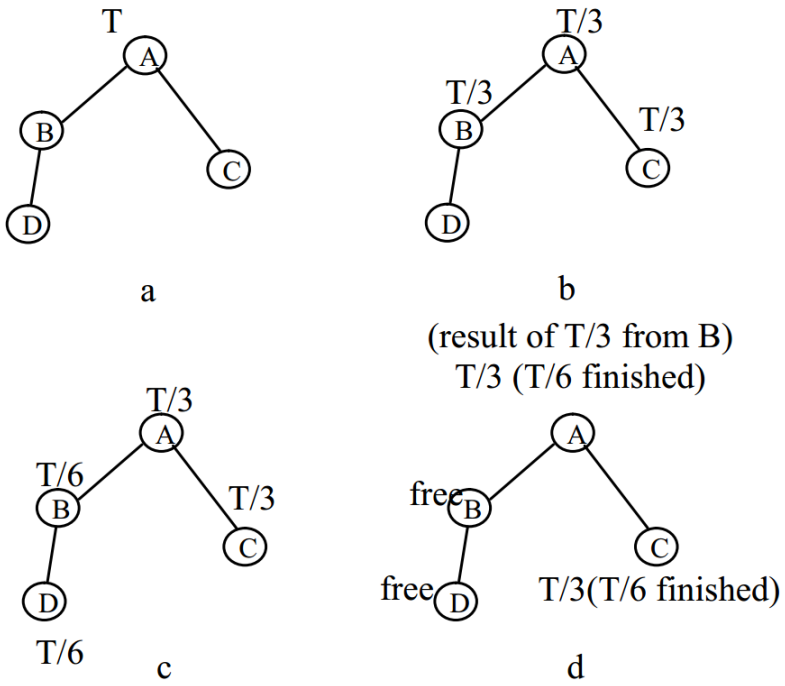

Abbildung 6.1: Aufgabenverteilung PPVC (entnommen aus [ZYX09])

Verlässt ein Teilnehmer P_{leave} das Netzwerk, müssen dessen ehemalige Nachbarn entsprechend reagieren. Ist P_{leave} ein Endknoten des Netzwerkes, wie Teilnehmer C in Abbildung 6.1, übernimmt das Berechnen der Unteraufgabe der Teilnehmer, der diese P_{leave} zugeteilt hat (hier Teilnehmer A). Für den Fall, dass P_{leave} selbst Unteraufgaben an andere Teilnehmer verteilt (hier Teilnehmer B), bauen diese Teilnehmer für die Rückgabe der Ergebnisse Verbindungen zu dem Teilnehmer auf, der P_{leave} die Aufgabe zugeteilt hat (hier Teilnehmer A).

Innerhalb des PPCV-Netzwerks werden die Aufgaben nicht gleichmäßig verteilt. So ist es möglich, dass – wie in Abbildung 6.1 zu sehen ist – zwei Teilnehmer je ein Drittel der Aufgabe berechnen und alle anderen Teilnehmer des Netzwerks das letzte Drittel auf sich verteilen. Zudem wird die Rechenleistung des Teilnehmers bei der Verteilung der Aufgaben nicht beachtet. Diese zwei Faktoren erhöhen die Berechnungszeit der Gesamtaufgabe. Die Verteilungsalgorithmen, die von VolunteerLib verwendet werden, verhindert dies dadurch, dass sich die Teilnehmer, aus einer geteilten Liste von Aufgaben (BitMaske), selbst ihre Aufgaben auswählen. Teilnehmer, die über weniger Rechenleistung verfügen, würden sich somit entsprechend weniger Teilaufgaben auswählen.

7 Zusammenfassung und Ausblick

In diesem Kapitel wird die Arbeit abschließend zusammengefasst. Insbesondere wird dabei auf die Umsetzung der, in der Einleitung gesetzten, Ziele eingegangen. Darauf folgend wird ein kurzer Ausblick über die weitere Entwicklung und Verwendung von VoluntLib gegeben.

7.1 Zusammenfassung

In dieser Arbeit wurde eine Programmbibliothek entwickelt, die es ermöglicht, Aufgaben selbstorganisiert in einem Peer-to-Peer Netzwerk zu verteilen. Die Aufgabenverteilung ist dabei unabhängig von der Bibliothek und kann somit um beliebige Algorithmen erweitert werden. Aktuell werden für Berechnungen die in 2.5.1 beschriebenen Algorithmen unterstützt.

Zu Beginn der Arbeit wurden die notwendigen Grundlagen für die Entwicklung der Bibliothek erläutert. Für die Kommunikation der Anwendung wurde anschließend ein eigenes Netzwerkprotokoll entworfen. Sodann wurde ihre Architektur besprochen und auf die Implementierung selbst eingegangen. Danach wurden Beispielprogramme vorgestellt, die mit Hilfe der Bibliothek eine Aufgabe verteilt berechnen. Darauf folgend wurde die Bibliothek in Bezug auf Funktionalität, Wartbarkeit und Parallelisierbarkeit evaluiert. Abschließend werden zwei verwandte Volunteer Computing Modelle vorgestellt. Zum einen die BOINC-Plattform, die nach einem zentralisierten Ansatz agiert und zum anderen das Volunteer Computing System, welches ebenfalls in einem unstrukturiertem Peer-to-Peer Netzwerk operiert.

7.2 Umsetzung der Ziele

Die Ziele der Arbeit und deren Umsetzung werden abschließend aufgezeigt.

- (Z1) Die Entwicklung eines Netzwerksprotokolls zur verteilten Berechnung, basierent auf dem Epochen-Verteilungsalgorithmus und dem Sliding-Window-Verteilungsalgorithmus.
- (Z2) Die Entwicklung einer .NET Bibliothek, welche das entwickelte Protokoll implementiert.
- (Z3) Die Erstellung von Beispiel-Applikationen für die entwickelte Bibliothek.

- (Z4) Die Evaluierung der Bibliothek.

Das Ziel **Z1** wurde durch das in Kapitel 3.1 vorgestellte Netzwerkprotokoll umgesetzt. Das Protokoll selbst ist dabei nicht an die beiden genannten Algorithmen gebunden, sondern nur an bestimmte Eigenschaften der Algorithmen.

Z2 wurde durch die in Kapitel 3.2 beschriebene .NET-Bibliothek umgesetzt. Die Bibliothek implementiert dabei den Epochen-Verteilungsalgorithmus. Aufgrund der begrenzten Zeit wurde, in Rücksprache mit dem Fachgebiet Angewandte Informationssicherheit, die Implementierung des Sliding-Window-Algorithmus auf einen späteren Zeitpunkt verlagert. Die Bibliothek selbst ermöglicht es, neben der Kommunikation in einem lokalen Netzwerk, auch durch den Einsatz von Netzwerkbrücken netzwerkübergreifend verteilte Aufgaben zu berechnen.

Für die Umsetzung von **Z3** wurden im Rahmen der Arbeit die zwei Beispielanwendungen entwickelt. Zum einen eine Demoanwendung, die die Funktionalität der Bibliothek zeigen soll, dabei aber keine Aufgabe berechnet. Zum anderen der Keysearcher. Dieser findet den Verschlüsselungsschlüssel zu einem DES-Ciphertext, indem er den vollständigen Schlüsselraum durchsucht. Eine weitere Referenzanwendung zur entwickelten Bibliothek stellt der Wireshark-Dissector da. Dieser Dissecor ermöglicht es, die Nachrichten, die zwischen den Teilnehmern ausgetauscht werden, direkt zu beobachten.

Bei der Evaluierung der Bibliothek (**Z4**) wurde zunächst die Funktionsweise der Bibliothek selbst anhand der Unit- und Acceptance-Tests dargelegt. Anschließend wurde die Erweiterbarkeit und die Wartbarkeit der entwickelten Anwendung anhand von statischen Metriken objektiv evaluiert.

7.3 Ausblick

Im Rahmen dieser Bachelorarbeit wurde eine .NET-Bibliothek für die Verteilte Berechnung in einem unstrukturierten Peer-to-Peer-netzwerk entwickelt. Abschließend werden noch einige Punkte vorgestellt, die im Nachgang der Bachelorarbeit angedacht oder geplant sind.

7.3.1 Veröffentlichung des Quellcodes

Die in dieser Arbeit entwickelte Programmbibliothek soll im Anschluss an diese Bachelorarbeit und nach einer Publizierung der verwendeten Verteilungsaltorithmen ebenso veröffentlicht werden. Der Quellcode soll dabei unter einer Open-Source-Lizenz veröffentlicht werden. Diese erlaubt eine kostenlose Benutzung, Modifikation und Verbreitung, sowohl im privaten als auch im kommerziellen Gebrauch. Die Veröffentlichung soll entweder über eine der gängigen Plattformen [Sur14, Git14] oder über eine eigene VoluntLib-Webseite erfolgen.

Quelloffene Software bietet aber auch Vorteile für die ursprünglichen Anwender und Entwickler. So kann die Anwendung von mehr Menschen angepasst, weiterentwickelt oder in der Praxis getestet werden, als dies ursprünglich der Fall war.

7.3.2 Einbau in CrypTool 2.0

CrypTool 2.0 ist eine OpenSource e-Learning Anwendung, die es interessierten Benutzern ermöglicht verschiedene kryptographische und kryptoanalytische Verfahren anzuwenden und auszuprobieren [Cry14].

Hierbei soll VoluntLib, die aktuell vorhandene Verteilungslogik in einigen Komponenten ersetzen. Die Anwender können so mit verschiedenen Computern in ihrem lokalen Netzwerk an einer Aufgabe rechnen. Weiterhin wird eine oder mehrere passive Netzwerkbücken den Anwendern von CrypTool 2.0 dabei helfen, gemeinsam Berechnungen durchzuführen.

7.3.3 Vertrauensmanagement und betrügerische Teilnehmer

Aktuell basiert das Vertrauen in die Korrektheit von Berechnungen anderer Teilnehmer darauf, dass eine Zertifizierungsstelle dem Teilnehmer ein Zertifikat ausgestellt hat. Dies ist allerdings ausschließlich eine Zugangsberechtigung und kein ein Vertrauensbeweis.

Im Nachgang zu dieser Bachelorarbeit ist es daher angedacht, VoluntLib um ein Vertrauensmanagement und eine Betrugserkennung zu erweitern. Damit soll es jedem Teilnehmer ermöglicht werden, die Korrektheit eines empfangenen Ergebnisses anhand der Vertrauenswürdigkeit des Senders abzuschätzen. Wobei das Vertrauen in einen Teilnehmer angibt, wie oft dessen Ergebnisse überprüft werden. Je höher das Vertrauen ist, desto weniger wird kontrolliert. Ein Teilnehmer soll anfänglich wenig Vertrauen zu anderen Teilnehmern haben und mittels einer heuristischen Betrugserkennung häufig eingehende Ergebnisse überprüfen. Wenn dabei kein Betrug erkannt wird, wird das Vertrauen zu dem Absender erhöht und seine Ergebnisse weniger häufig kontrolliert. Sollte jedoch ein Betrug erkannt werden, wird das Vertrauen des Absenders herunter gesetzt. Sobald dieses unter einen gewisse Schwellwert fällt, wird der Teilnehmer ignoriert.

Literaturverzeichnis

[AD99] Allen, Christopher und Tim Dierks: *The TLS protocol version 1.0*. 1999. RFC 2246.

[Amd67] Amdahl, Gene M: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, Seiten 483–485. ACM, 1967.

[And04] Anderson, David P: *Boinc: A system for public-resource computing and storage*. In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, Seiten 4–10. IEEE, 2004.

[Bak95] Baker, Fred: *Requirements for IP version 4 routers*, 1995. RFC 1812.

[CALO94] Coleman, Don, Dan Ash, Bruce Lowther und Paul Oman: *Using metrics to evaluate software system maintainability*. Computer, 27(8):44–49, 1994.

[Cli14] Climateprediction.net: *A distributed computing, climate modelling project*. `http://www.climateprediction.net/`, 2014. [Online; zugegriffen am 27. März 2014].

[Cry14] CrypTool2: *Cryptography for everybody*. `http://www.cryptool.org/de/cryptool2`, 2014. [Online; zugegriffen am 28. März 2014].

[Dee88] Deering, Stephen E: *Host extensions for IP multicasting*, 1988. RFC 1112.

[Dee98] Deering, Stephen E: *Internet protocol, version 6 (IPv6) specification*, 1998. RFC 2460.

[Dro97] Droms, Ralph: *Dynamic host configuration protocol*, 1997. RFC 2131.

[Ein14] Einstein@Home: *Catch a Wave From Space*. `http://einstein.phys.uwm.edu/`, 2014. [Online; zugegriffen am 27. März 2014].

[EJ01] Eastlake, Donald und Paul Jones: *US secure hash algorithm 1 (SHA1)*, 2001. RFC 3174.

[Fen97] Fenner, William C: *Internet group management protocol, version 2*. 1997. RFC 2236.

[FK14] Frans Kaashoek, David Karger, Robert Morris Emil Sit Jeremy Stribling: *Chord*. `https://github.com/sit/dht/wiki`, 2014. [Online; zugegriffen am 9. April 2014].

[Fol14] Folding@Home: *Client statistics*. `http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats2`, 2014. [Online; zugegriffen am 26. März 2014].

[Git14] GitHub: *Build software better, together*. `https://github.com/`, 2014. [Online; zugegriffen am 28. März 2014].

[Kop12] Kopal, Nils: *Verteilte Berechnungen in unstrukturierten Peer-to-Peer-Netzwerken*. Masterarbeit, Universität Duisburg-Essen, 2012.

[MC14a] MysteryTwister-C3: *Berechen von mit SHA1 gehashten Passworten*. `https://www.mysterytwisterc3.org/images/challenges/mtc3-kitrub-07-sha1crack-de.pdf`, 2014. [Online; zugegriffen am 5. April 2014].

[MC14b] MysteryTwister-C3: *The Crypto Challenge Contest*. `https://www.mysterytwisterc3.org/de/`, 2014. [Online; zugegriffen am 5. April 2014].

[McC76] McCabe, Thomas J: *A complexity measure*. Software Engineering, IEEE Transactions on, (4):308–320, 1976.

[Mog84] Mogul, Jeffrey C: *Broadcasting Internet datagrams in the presence of subnets*, 1984. RFC 922.

[Nak08] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system*. Consulted, 1:2012, 2008.

[Pos81] Postel, Jon: *Internet protocol*, 1981. RFC 791.

[PUB77] PUB, NIST FIPS: *46-3. Data Encryption Standard*. Federal Information Processing Standards, National Bureau of Standards, US Department of Commerce, 1977.

[Sen99] Senie, Daniel: *Changing the default for directed broadcasts in routers*. 1999. RFC 2644.

[Sha01] Shannon, Claude Elwood: *A mathematical theory of communication*. ACM SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

[SHF02] Solo, David, Russell Housley und Warwick Ford: *Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile*. 2002. RFC 5280.

[Sur14] SurceFroge: *Download, Develop and Publish Free Open Source Projects*. `https://sourceforge.net`, 2014. [Online; zugegriffen am 28. März 2014].

[top14a] TOP500: *Tianhe-2 Supercomputer*. `http://www.top500.org/system/177999`, 2014. [Online; zugegriffen am 26. März 2014].

[top14b] TOP500: *Titan Supercomputer*. `http://www.top500.org/system/177975`, 2014. [Online; zugegriffen am 26. März 2014].

[Wir14a] WIRESHARK: *Go Deep*. `https://www.wireshark.org/`, 2014. [Online; zugegriffen am 7. März 2014].

[Wir14b] WIRESHARK: *Lua Documentation*. `https://www.wireshark.org/docs/wsug_html_chunked/wsluarm.html`, 2014. [Online; zugegriffen am 7. März 2014].

[WMW96] WATSON, ARTHUR H, THOMAS J MCCABE und DOLORES R WALLACE: *Structured testing: A testing methodology using the cyclomatic complexity metric*. NIST special Publication, 500(235):1–114, 1996.

[ZYX09] ZHAO, ZHIKUN, FENG YANG und YINGLEI XU: *Ppvc: A p2p volunteer computing system*. In: *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, Seiten 51–55. IEEE, 2009.

Versicherung an Eides statt

Ich, Christopher Konze, Matrikelnummer 29237498, wohnhaft in 34369 Hofgeismar, versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen übernommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Ich versichere an Eides statt, dass ich die vorgenannten Angaben nach bestem Wissen und Gewissen gemacht habe und dass die Angaben der Wahrheit entsprechen und ich nichts verschwiegen habe.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 163 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Kassel, 13. April 2014 — Christopher Konze