

18. März 2002

Optimierung client-/serverbasierter Statistiksysteme



Diplomarbeit
zur Erlangung des Grades
eines Diplom-Volkswirtes
an der Wirtschaftswissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

vorgelegt von
Jörg Feuerhake
(Matrikel-Nr. 136271)

Prüfer: Prof. Dr. Wolfgang Härdle

Berlin, 19. März 2002

Inhaltsverzeichnis

1	Einleitung, Motivation und Vorgehen	4
2	Technische Gegebenheiten	6
2.1	Konzepte, Umgebungen	6
2.1.1	Kommunikation	6
2.1.2	Das Protokoll	6
2.1.3	Unified Modelling Language	7
2.1.4	Java	8
2.1.5	Schnittstelle XQSListener	9
2.1.6	Kommunikationsstruktur	10
2.1.7	ReX-Client	10
2.2	Resultierende Vorgehensweisen	11
3	Fehlertoleranz	13
3.1	Methoden der Bestandsaufnahme, Bestandsaufnahme	13
3.1.1	Statusnachricht	13
3.1.2	Laufzeit	14
3.1.3	Fehlerinformation	14
3.2	Ziele der Implementation	15
3.3	Implementation	15
3.3.1	Laufzeit	15
3.3.2	Statusnachricht	16
3.3.3	Generelle Fehlerbehandlung	16
3.3.4	Information über die Programme	17
3.3.5	Kommunikationsfehler	18
3.4	Ergebnis der Implementation	20
4	Geschwindigkeit	22

4.1	Methoden der Bestandsaufnahme, Bestandsaufnahme	22
4.1.1	Methode 'echter' Zeitmessungen am Code	22
4.1.2	Methode der 'Referenzquantlets'	25
4.2	Ziele der Implementation	25
4.3	Implementation	25
4.3.1	Identifizieren der Engpässe	25
4.3.2	Mankos des java.io.Data(IO)Streams	26
4.3.3	BufferedDataInputStream / BufferedDataOutputStream . .	27
4.3.4	Bytefelder MD*Cryptseitig	30
4.3.5	MD*Serv	31
4.4	Ergebnis der Implementation	32
4.4.1	Technische Umgebung	32
4.4.2	Analyse	32
5	Funktionalitätserweiterung	37
5.1	Handlungsfelder	37
5.1.1	Erweiterung der Kommunikationsstruktur	37
5.1.2	Abbildung der XploRe Kommandos	37
5.1.3	Optimierte Informationstransfermethoden	38
5.1.4	Drittprogrammierer	39
5.2	Bestandsaufnahme, Möglichkeiten	39
5.2.1	Datentypen	39
5.3	Ziele der Implementation	40
5.4	Methoden der Implementation	40
5.5	Ergebnis der Implementation	43
6	Protokolloptimierung	45
6.1	Geschwindigkeit	45
6.2	Komprimierungsalgorithmus	45

7	Ergebnisanalyse	49
7.1	Neue Funktionalitäten	49
7.2	Geschwindigkeit	49
7.3	Fehlertoleranz	50
7.4	ReX	50
7.5	XQC und nahe Projekte	50
8	Ausblick	52
8.1	MD*Crypt	52
8.2	Interaktive Grafik	53
8.3	MD*Serv	54
8.4	Drittprogrammierer	54
8.5	Sicherheit	55
8.6	Java 1.4	56

1 Einleitung, Motivation und Vorgehen

Diese Arbeit wird sich mit der Optimierung client-/serverbasierter Statistiksyste-
me befassen. Die rechnergestützte Lösung statistischer Probleme ist durch zwei
Haupt Herausforderungen gekennzeichnet: Erstens sind die Algorithmen zu ih-
rer Lösung oft sehr rechenaufwändig, und zweitens existiert eine Fülle statis-
tischer Methoden, welche idealer Weise von möglichst vielen Anwendern er-
reicht werden sollen. Diese computergestützte Lösung statistischer Aufgaben war
lange eine Domäne leistungsfähiger Großrechenmaschinen, welche über Termi-
nals kontaktierbar waren. Mit dem Aufkommen von Computernetzwerken, hier
speziell des Internets, ergab sich die Möglichkeit, diese leistungsfähigen Ma-
schinen geografisch unabhängig zu kontaktieren. Da sie regelmäßig multiuser-
und multitaskingfähig sind, führt dies zur Entwicklung von Architekturen, die
als C-/S-Architekturen bezeichnet werden. Solche C-/S-Architekturen zeichnen
sich dadurch aus, dass alle anfallende Rechenarbeit auf derjenigen Serverma-
schine abgearbeitet wird, auf der sowohl Rechenkapazität als auch Algorithmen
zur Verfügung stehen. Der Client übernimmt nunmehr die Aufgabe der Nut-
zerinteraktion. Die Kommunikation zwischen Client und Server wird über ein
Protokoll ermöglicht, welches alle nötigen Datentypen sowie die Befehlsstruk-
tur des zugrunde liegenden statistischen Programmes abbildet. Sowohl Client- als
auch Serverprogramme implementieren diese Protokolle, was die Kommunikati-
on möglich macht. Viele Statistiksyste-
me gehen den Weg der C-/S-Architektur
(Rweb).

Ein Problem teilen Clientanwendungen mit anderen herkömmlichen Anwendun-
gen: Sie implementieren eine Oberfläche und damit eine Bedienungsumgebung, die
vorgegeben ist und so entweder die Einsatzgebiete der Clients beschneidet oder,
falls der Client eine gewisse generelle Einsetzbarkeit behalten soll, das Client-
programm unnötig groß werden läßt. Gerade für Internetanwendungen (Applets)
mit oft geringen Datendurchsatzraten ist das Letzgenannte ein schweres Problem.
Bleibt also nur, für jedes Problem, das sich stellt, einen eigenen, möglichst klei-
nen, möglichst gut angepassten Client zu entwickeln. Diesen Ansatz verfolgt
XploRe mit seiner XploReQuantletServer-/XploReQuantletClient (XQS/XQC) -
Architektur. Wenn aber jedes Problem einen angepassten Client erhält, wird die
clientseitige Kommunikationsschnittstelle jedesmal neu implementiert werden.
Das Java-Paket MD*Crypt löst - zumindest in XQS-/XQC-Architekturen - die-
se Ineffizienz elegant. Es schafft die Möglichkeit, sich beim Entwickeln einer
Clientanwendung im XQS-/XQC-Rahmen voll auf die Implementation der gra-
phischen Benutzerschnittstelle zu konzentrieren, ohne viel Aufmerksamkeit auf
die Entwicklung der C/S-Kommunikationsschnittstelle zu verwenden. Für Details
möchte ich auf das Kapitel 'Technische Gegebenheiten' verweisen.

Parallel zur C-/S-Entwicklung entstand das Segment der Personalcomputer mit leistungsfähigen, weitverbreiteten Standard-Anwendungen, die mit immer schnelleren Prozessoren und unter immer stabileren Betriebssystemen zu Alternativen für die Bewältigung statistischer Probleme reiften (Microsoft's Excel). Sowohl im geschäftlichen als auch im privaten Bereich stellen Tabellenkalkulationsprogramme (Excel) heute den Weg zur Lösung fast aller mathematischer Aufgabenstellungen dar. Für den Entwickler statistischer Algorithmen heißt das aber, dass seine Methoden in diesen Tabellenkalkulationsprogrammen verfügbar sein sollten, um möglichst viele Anwender in ihrer gewohnten Umgebung zu erreichen.

Es entstand nun die Frage, ob eine der weitverbreiteten Standardtabellenkalkulationen ein XploReQuantletClient sein kann und damit die in XploRe entwickelten leistungsfähigen statistischen Methoden einer großen Menge von Nutzern zugänglich zu machen. Der ReXClient realisiert unter Verwendung der MD*Crypt-Technologie diesen Client. ReX ist ein Excel-AddIn, welches Excel befähigt, XQS zu kontaktieren und so statistische Methoden aus XploRe (Quantlets) zur Verfügung zu stellen.

Das Verwenden des ReX-Clients brachte Erfahrungen, die es wünschenswert machen, MD*Crypt als Packet, das die Schnittstellen zum XQS bereitstellt, zu verbessern. Bei der Arbeit mit ReX hat sich gezeigt, dass die bisher implementierten Datentypen in der (Java/VBA) Umgebung von Excel nicht ausreichen, um den hohen Anforderungen eines Quantlets an die Datengenauigkeit gerecht zu werden. Es besteht Handlungsbedarf bei der Implementation genügend genauer Datentypen zur Kommunikation. Auch die Projekte GraFitI und XQS können hier erheblich profitieren (siehe: Ressourcen im World Wide Web).

Weiter zeigte sich, dass der Geschwindigkeitsverlust, den die Kommunikation über das Netz mit sich bringt, nicht vernachlässigbar klein ist. Im Rahmen dieser Arbeit wird also auch die Verbesserung der Bearbeitungsgeschwindigkeit von MD*Crypt eine große Rolle spielen. Als dritter Punkt konnte darin ausgemacht werden, dass das Bearbeiten von auftretenden Ausnahmen in MD*Crypt nicht ausreichend stabil gelöst ist. Es wird im Rahmen dieser Arbeit versucht werden, hier bessere Algorithmen zu finden, die Ausnahmen besser behandeln und das Informationsangebot am XQSListener erhöhen. Wenn am XQSListener mehr Information über die beteiligten Akteure und ihren Zustand vorliegen, hat der Listener mehr Möglichkeiten, entsprechend fehlertolerant zu reagieren. Bei der Implementation wird sicher auch eine stabilere Ausnahmenbehandlung innerhalb MD*Crypt zu erreichen sein. Die drei Punkte Funktionalitätserweiterung, Erhöhung der Bearbeitungsgeschwindigkeit und fehlertolerantere Datenfernübertragung werden im Rahmen dieser Arbeit betrachtet und bearbeitet werden.

2 Technische Gegebenheiten

Im folgenden Kapitel sollen zu Grunde liegende Konzepte, technische Umgebungen und daraus resultierende Vorgehensweisen genauer benannt werden. Zum weiteren Vorgehen scheint dies sinnvoll.

2.1 Konzepte, Umgebungen

Am Beginn einer jeden Analyse müssen Begriffe geklärt, die technische Umgebungen und Konzepte der zu Grunde liegenden Architektur identifiziert werden.

2.1.1 Kommunikation

Da es in dieser Arbeit um die Optimierung client-/serverbasierter Systeme gehen wird, ist es nötig, vorab das Wesen von Kommunikation zu klären.

Kommunikation findet zwischen zwei Akteuren statt, die wechselseitig Sender und Empfänger sind. Kommunikation basiert immer auch darauf, dass beide Akteure den gleichen Zeichensatz sowie die gleiche Syntax, verwenden. Dies ist im MD*Crypt-/XQS-Kontext das Protokoll. Es stellt Methoden zur Verfügung, die einen Zeichensatz oder eine Syntax definieren. Mit Hilfe dieser Syntax können dann Sinnzusammenhänge geschaffen werden, die von beiden Akteuren verstanden werden.

Die Sprache wird auf einem Kanal übertragen. Im MD*Crypt-/XQS-Fall besteht dieser Kanal aus einer TCP-/IP-Verbindung, die zwischen MD*Crypt und MD*Serv existiert, und einer Standard Input-/Output-Verbindung, die zwischen MD*Serv und XploRe Quantlet Server gehalten wird.

Die Sprache wird im MD*Crypt/XQS Kontext dazu verwendet, Anfragen an den XploRe Quantlet-Server zu richten und dessen Antworten zu erhalten. Bisher ist die Sprache der Akteure MD*Crypt und XQS sehr stark an genau dieser Anforderung ausgerichtet. Dies grenzt die Einsatzgebiete für MD*Crypt-Anwendungen ein.

2.1.2 Das Protokoll

Das MD*Crypt-Paket ermöglicht die Kommunikation zwischen XQC und XQS auf Basis eines Protokolls (Kleinow 2002). Vereinfachend kann man zur Funktionsweise sagen: Befehle, Befehlsstrukturen sowie komplexe Objekte aus Xplo-

Re werden serialisiert und in einen 'Stream' oder 'Queue' geschrieben. Das Programm am anderen Ende des Streams implementiert Methoden, um die Befehle und Befehlsstrukturen zu erkennen und bearbeitbar zu machen und um die serialisierten XploRe Objekte wieder zusammenzufügen sowie verfügbar zu machen. Betrachte hierzu Grafik 1.

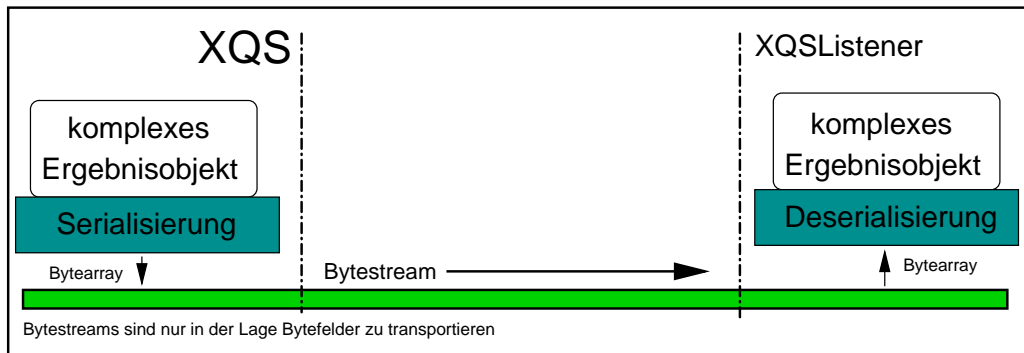


Abbildung 1: Konzept der Bytestreamserialisierung

Das MD*Crypt-Protokoll ist speziell für die Szenarien entwickelt worden, für die der XploReQuantleClient bei seiner Erstimplementierung vorgesehen war. Dies führte zu einer sehr speziellen Anpassung der Funktionalitäten. Diese Szenarien bestanden darin, Kommandos an den Server zu schicken und die als Antwort erhaltenen Ergebnisobjekte darzustellen. Es erweist sich, dass XploReQuantletClients schon jetzt in weiteren Szenarien eingesetzt werden. Mit diesen neuen Einsatzgebieten stößt jedoch das MD*Crypt-Protokoll an erste Grenzen. In einigen Einsatzgebieten werden als Kernfunktion schon das Senden größerer und komplexerer Datenstrukturen zum XploReQuantletServer gefordert. Dies geschieht momentan noch auf dem unbefriedigenden Weg der Konvertierung der Strukturen zu Zeichenketten. Eine genauere Diskussion erfolgt im Abschnitt 2.1.6.

2.1.3 Unified Modelling Language

Die Unified Modelling Language (UML) wurde für objektorientierte Programmieransätze zu einer Standardmodellierungssprache. Es ist mit UML möglich, Beziehungen zwischen Objekten, deren Vererbungshierarchien sowie deren Beziehungen untereinander leicht lesbar darzustellen. Das UML-Konzept soll hier dazu dienen, sowohl Beziehungen zwischen neu zu schaffenden Objekten zu modellieren als auch - für Dokumentationszwecke in dieser Arbeit und später - leicht lesbare Diagramme zu erstellen (www.omg.org 2001).

Speziell UML-Klassendiagramme sollen hier verwandt werden. Es sollen also einige grundlegende Eigenschaften dieser Diagramme beleuchtet werden. Ein Objekt nach Lesart objektorientierter Programmierung hält immer Eigenschaften und Methoden. Es lässt sich weitergehend auf seine Eigenschaften und Methoden reduzieren. Ein Objekt wird also in einem Klassendiagramm als horizontal dreigeteiltes Rechteck dargestellt. In der oberen Ebene steht der Name des Objekts. Im mittleren Feld sind seine Eigenschaften aufgeführt, und zuletzt werden im unteren Feld seine Methode aufgelistet (vergleiche Abbildung 4). Mit Klassendiagrammen kann man Objekte ausreichend beschreiben, um grundsätzliche Beziehungen und Eigenschaften zu klären.

UML-Klassendiagramme eignen sich hervorragend, um Objekte darzustellen. Sie werden im Rahmen dieser Arbeit benutzt werden, um Konzepte und zu implementierende Objekte darzustellen. Dies scheint der beste Weg, um Aspekte dieser Arbeit fassbar zu machen.

2.1.4 Java

Um Plattformunabhängigkeit zu garantieren, wurde MD*Crypt als Java-Paket entwickelt. Da sich Java von Anfang an Plattformunabhängigkeit auf die Fahnen geschrieben hat ('Write once, run anywhere'), schien dies die Sprache der Wahl, um die Anforderungen zu erfüllen. Dass die Ansätze und Paradigmen der Objektorientierung bei Modellierung und Entwicklung des Packetes zur Anwendung kamen, versteht sich von selbst.

Gerade im Input-/Output-Bereich bleibt leider festzustellen, dass 'Write once, run anywhere' nicht mehr als ein Versprechen ist. Input-/Output-Methoden - im Bereich Geschwindigkeitsoptimierung werden diese von besonderem Interesse sein - gehen immer mit dem Rufen von Systemmethoden einher. Diese können sehr unterschiedlich arbeiten. Dies wiederum heißt, es wird immer nötig sein, die Kommunikationsmethoden für alle Plattformen, welche man unterstützen will, getrennt zu untersuchen und zu implementieren (Donehower 2001).

In einem speziellen Projekt, dem Microsoft Excel Add In Rex, wird aus den Java Quellen eine COM DLL generiert. Diese wird dann von einem VBA-Programm angesprochen. Für diesen Einsatz müssen gesonderte Lösungen entwickelt werden, da an der Schnittstelle zwischen Java und VBA nur einfache Datentypen übergeben werden können. Weiter erweist sich das Konzept des XQSListener-Interfaces als nicht mehr verwendbar.

2.1.5 Schnittstelle XQSListener

Als besondere Herausforderung erwies sich die Anforderung, es einem späteren Programmierer zu überlassen, wie und in welcher Umgebung sein Client arbeiten soll. Als guter Lösungsansatz wurde eine Herangehensweise gewählt, die die Clientklasse als Schnittstelle definiert. Aus den Ansätzen der Objektorientierung ergibt sich die Möglichkeit, eine Klasse eines Packetes eben nicht als Klasse, sondern als Schnittstelle (Interface) zu implementieren; in diesem werden Methoden nur definiert und nicht ausimplementiert, was den zukünftigen Benutzer dieses Interfaces zwingt, bestimmte Methoden mit streng definierten Parametern an seinem Client zu implementieren. Das gibt dem Programmierer des Interfaces wiederum die Möglichkeit, an dem späteren Client, den er selbst noch gar nicht kennt, Methoden aufzurufen.

Im MD*Crypt Packet existiert das Interface `mdcrypt.XQSListener`, welcher durch den Benutzer des Packetes zu implementieren ist. Das heißt, der Benutzer von MD*Crypt hat mit der Implementierung des `mdcrypt.XQSListeners` zwei Methoden zu implementieren. Diese heißen `XQSListener.serverStatusChanged(int i)` und `XQSListener.handleServerReply(XQSObject xobj)`. Die Erstere wird von MD*Crypt verwandt, um dem Client Informationen über den Zustand des kontaktierten Servers zu übermitteln; die zweite wird benutzt, um dem Client die Möglichkeit zu geben, auf Ergebnisse des XQS zu reagieren.

Der Vorteil dieses Vorgehens besteht vorrangig darin, dass es einem zukünftigen Programmierer überlassen bleibt, die Ergebnisse des XploRe Quantlet Servers für seine Bedürfnisse und seine technischen Gegebenheiten optimiert zu behandeln. Nicht alle XQSListener werden zwingend Grafikkontexte aus XploRe behandeln müssen. Wenn es in bestimmten Einsatzgebieten darauf ankommt, bestimmte Methoden von XploRe aufzurufen, die lediglich Zeichenketten zurückliefern, kann der Client genau dafür optimierte Ergebnisbehandlungsmethoden bereitstellen. MD*Crypt-Objekte rufen Methoden, deren Parameter ihm schon bekannt sind, deren Implementation aber nicht bekannt ist.

Aus dieser Herangehensweise ergeben sich sofort zwei Folgen: Erstens ist MD*Crypt kein selbst lauffähiges Programm, sondern eine Sammlung von Methoden, um einen XploRe-Quantlet-Server zu kontaktieren; und zweitens wird immer ein erhöhter Dokumentationsbedarf, die MD*Crypt-Methoden betreffend, nötig sein, da Drittprogrammierer immer erst die Implementation eines XQSListeners leisten müssen. Vergleiche (Kleinow 2002).

2.1.6 Kommunikationsstruktur

Die Kommunikationsstruktur in XQS-/MD*Crypt ist durch ein strenges Frage-Antwortmuster gekennzeichnet. Der Client, respektive MD*Crypt, sendet eine Anfrage, und der Server, hier XQS, liefert eine Antwort. Der Unterschied zwischen Frage und Antwort besteht darin, dass die Antwort im Gegensatz zur Frage aus hochkomplexen Gebilden bestehen kann. Ein Grafikkontext - beispielsweise - besteht aus einem Displaykontext, welches aus einer $n \times m$ -Matrix von Grafikkontainern besteht. In diesen Grafikkontainern kann wiederum ein Grafikoobjekt angezeigt werden, welches wiederum aus einer Liste von zwei- oder dreidimensionalen Datenobjekten und ihrer jeweils korrespondierenden grafischen Repräsentation besteht. Die Frage ist nur eine Zeichenkette. Der Hauptteil der Logik, die MD*Crypt implementiert, ist damit befasst, die komplexen Antwortstrukturen in abfragbare Objekte umzuwandeln. Im Rahmen dieser Arbeit wird die Frage gestellt werden, ob diese Struktur ausreichend für die Szenarien ist, in denen MD*Crypt heute schon zum Einsatz kommt.

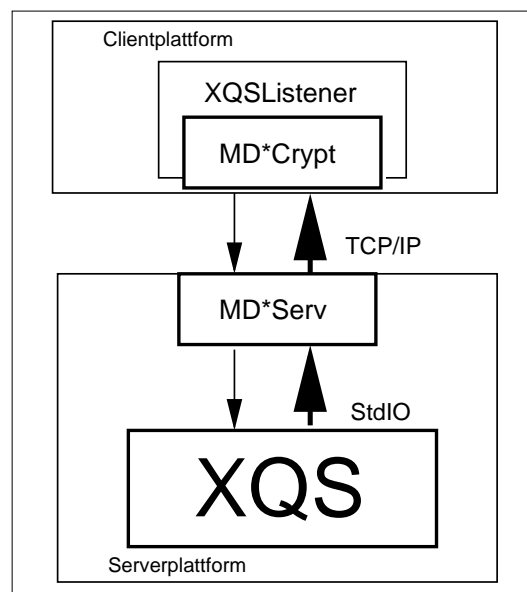


Abbildung 2: Die dreistufige XQS-/MD*Crypt-Architektur

2.1.7 ReX-Client

Für einen der Referenzclients dieser Arbeit, den Rex-Client, ergeben sich einige Besonderheiten. ReX ist ein MSEXcel-AddIn, dessen natürliche Sprache VBA,

ein Visual Basic-Dialekt, ist. Zur technischen Spezifikation siehe (Aydinli 2002b). In VBA können Pakete nur in VBA selbst oder als Dynamically Linked Library (DLL) importiert werden. Man kann zwar aus Javaquelldateien eine DLL erzeugen, zwingt sich aber, Microsoft's Implementation der Java Virtual Maschine zu verwenden. Diese ist aber technisch auf einem Niveau, das uns einige Zugeständnisse abnötigt. Dazu gehören die Nichtverwendbarkeit von Threads und die Nichtverwendbarkeit von Java's Swingkomponenten. Diese Zugeständnisse ermöglichen aber das Schreiben einer DLL in Java, die wieder mittels VBA ansprechbar ist. Wie bereits erwähnt, ist das Verwenden komplexerer Datentypen zwischen VBA und der aus Javaquelltext erzeugten COM DLL sehr eingeschränkt. Es mag möglich sein, dass sich der Zwang, unter zwei Java Versionen zu entwickeln, zu einem ernsteren Problem auswächst, was auch durch das später nötige Kapseln in VBA nicht geringer wird.

Weiter ist Microsoft's Excel eine Standardanwendung zur Bearbeitung von Daten. Ein Excel-Add-In, das einen Server kontaktiert, der statistische Methoden vorhält, muss zwingend in der Lage sein, auch größere und komplexere Datenstrukturen zu übertragen. Mit der bisherigen Kommunikationsstruktur ist dies nur eingeschränkt zu realisieren. Auf dem Weg der Umwandlung der Datenstrukturen zu zeichenkettenbasierten XploRekommandos ist dies zwar bisher möglich. Aber sowohl die Genauigkeit der übertragenen Daten als auch die Geschwindigkeit der Übertragung bleiben nicht zufriedenstellend.

Um den RexClient nutzen zu können, war es erforderlich, den XQSListener innerhalb der Javaklassen zu implementieren. Das Konzept der Interfaceimplementation, welches die Kommunikationsimplementation getrennt von der Clientimplementation ermöglichte, wird so ausgehebelt. MD*Crypt-Kommunikationsklassen haben keine echte Möglichkeit, die Methoden am Client aufzurufen. Das Laufzeitverhalten des ReX-Clients leidet darunter.

Da MD*Crypt grundsätzlich offen für andere technische Systeme sein muss, kann man, ausgehend von der bisher verwandten Threadtechnologie, die Frage stellen, ob diese threadorientierte Herangehensweise - ausser bei ReX - auch noch bei zukünftigen Projekten Probleme aufwerfen kann. Um eine möglichst generelle Anwendbarkeit zu gewährleisten sollte, dem Cliententwickler auf jeden Fall die Möglichkeit eingeräumt werden, seine Clients threadfrei zu schreiben.

2.2 Resultierende Vorgehensweisen

Weil MD*Crypt bisher nicht für spezielle Clients optimiert wurde, obwohl auf Microsoft-Java-Kompatibilität des Codes geachtet wurde, wurde mit Sun-Java-

Version 1.3.1 unter Solaris gearbeitet. Für die Dauer dieses Projektes wird jedoch eine Migration nach Windows zur Microsoft Java Virtual Machine unumgänglich sein, um schnelle Entwicklungen und Tests zu gewährleisten. Das Portieren in eine Java-1.3.1-Umgebung sollte unproblematisch sein, wenn man sich an die absolute Schnittmenge der Befehlsstruktur zwischen den Java-Versionen 1.0 und 1.3.1 hält. Dies ist freilich auch schon wieder ein Zugeständnis, womöglich sogar Quelle von späteren technischen Problemen.

Für das Handlungsfeld 'Geschwindigkeitserhöhung' müssen zwei Methoden der Ergebnismessung berücksichtigt werden, und zwar zum einen die direkte Methode im Wege der millisekundenweisen Zeitnahme direkt während des Programmlaufs und zum anderen die näher am 'usecase' orientierte Methode der Referenzquantlets, bei der die Zeit für das Bearbeiten verschiedener Quantlets unter verschiedenen Bedingungen untersucht wird. Beide Methoden haben Vor- und Nachteile. Die zweite ermöglicht Messungen, ohne den ausführenden Code selber zu belasten oder schwer beherrschbar zu machen. Die erste liefert präzisere Ergebnisse und kann bis zu Verweilzeiten in einzelnen Elementarmethoden alles erfassen.

Im Handlungsfeld "Funktionalitätserweiterung" sollen im Rahmen dieser Arbeit eng an den Referenzprojekten orientierte Fragestellungen angegangen werden, nämlich das Schicken numerischer Datenstrukturen zum XploRe-Quantet Serverdoubles. Um hier, ausgehend von den Gegebenheiten, eine konsistente Implementation zu gewährleisten, werden neuere Methoden des Anwendungsdesigns (classdiagrams etc.) angewandt werden.

Die Fehlertoleranz der MD*Crypt-Kommunikationsklassen wird durch die Identifizierung von Fehlerklassen angegangen werden. In einem weiteren Schritt werden Möglichkeiten zur Vermeidung dieser Fehler betrachtet. Weiterhin werden die derzeitigen Fehlerbehandlungsstrategien auf ihre Tauglichkeit überprüft.

Zur Optimierung der XQS/MD*Crypt-Sprache werden Veränderungen des eigentlichen Protokolls vorgeschlagen werden, die eine effizientere Kommunikation und damit ein günstigeres Laufzeitverhalten MD*Crypts erreichen können. Diese werden im Kapitel "Protokolloptimierung" diskutiert werden. Diese erfordern sowohl MD*Crypt- als auch XQS-seitige Neuimplementationen, die ausführlichen Tests unterzogen werden sollten.

Da Drittprogrammierer die zur Verfügung gestellten Methoden auch immer benutzen können müssen, ist es zwingend notwendig, die Änderungen und Neuimplementationen genau und gewissenhaft zu dokumentieren.

3 Fehlertoleranz

3.1 Methoden der Bestandsaufnahme, Bestandsaufnahme

Die Analyse des Ist-Zustands im Handlungsfeld gestaltet sich einfach. Im reinen Betrieb eines Clients sind Fehler zu beobachten, deren Ursache man jeweils mit den üblichen Mitteln der Fehlersuche finden kann. Grundsätzlich handelt es sich bei MD*Crypt um ein Programmpaket zur Implementation einer Kommunikation zwischen zwei Programmen. Der Großteil der Fehler ist also im Bereich purer Kommunikation zu suchen.

3.1.1 Statusnachricht

Damit zwischen XQSListener und XploRe-Quantlet Server überhaupt eine Kommunikation möglich wird, muss dem XQSListener Information über den Zustand des Servers zur Verfügung stehen. Vereinfachend gesagt: Der Fragende muss wissen, ob der Gefragte anwesend und zuhörend, anwesend und beschäftigt oder nicht anwesend ist. Diese Information erhält XQSListener über seine Methode `serverStatusChanged`.

Beim Einsatz von MD*Crypt zeigt sich, dass Informationen über den Zustand der Kommunikation mit dem XploRe-Quantlet-Server in nur sehr geringem Maße an den entsprechenden XQSListener weitergegeben werden. Der Clientprogrammierer wird zwar über die Definition der XQSListener-Schnittstelle gezwungen, die Methode `XQSListener.serverStatusChanged(int Status)` zu implementieren, aber gerade in Ausnahmesituationen sind die hier übertragenen Nachrichten oft nicht ausreichend.

Ob die Bearbeitung eines Kommandozeuges viel Zeit in Anspruch nimmt oder ein Fehler oder eine fehlerverdächtige Situation aufgetreten ist, kann dem XQSListener auf diesem Wege nicht vermittelt werden. Vielmehr wird nur der generelle Information weitergegeben. Die derzeit zu übertragenden Zustände sind:

```
SOCKET_INITIALIZED,  
HANDSHAKE_DONE,  
CONNECTION_ACCEPTED,  
SERVER_READY ,  
SERVER_BUSY,
```

SERVER_WAITING,
NOT_CONNECTED.

Diese Zustände bilden zwar die Grundlage der XQSListener-Information und schaffen überhaupt erst die Möglichkeit für eine Kommunikation zwischen XQSListener und XploRe-Quantlet-Server. Trotzdem zeigt sich, dass in einigen Szenarien mehr Information hilfreich wäre. Es wird im Weiteren zu klären sein, ob diese Information ausreichend ist oder ob Wege gefunden werden müssen, mehr und andere Information zu übertragen.

Im Rahmen dieser Arbeit wird die Implementierung weiterer Methoden zur Informationsübertragung an den XQSListener dargestellt werden. Diese werden den Informationsstand über den Status der Kommunikation und des Servers auf XQSListenerseite nachhaltig verbessern.

3.1.2 Laufzeit

Ein großes Problem aus Nutzersicht ist die Laufzeit eines Programms. Ein Nutzer, der an einer Oberfläche keine Veränderung bemerkt, wird nach einer gewissen Toleranzzeit annehmen, dass im Programm ein Fehler aufgetreten ist. Dies hat zunächst nichts damit zu tun, ob wirklich ein Fehler aufgetreten ist. Solche Versuche, Programme, die noch planmäßig laufen, zu beenden, kann man dadurch unterbinden, dass in kürzeren Abständen Information über die Aktivität (hier: des Servers an den Nutzer) weitergegeben werden.

Gerade bei rechenintensiven Programmen wie XploRe ist es wichtig, solche Informationen auszugeben, um den Nutzer davon abzuhalten, planmäßig laufende Prozeduren zu beenden. XploRes Standalone-Version macht diese Ausgaben bereits.

Es wäre ein erster Schritt, diese Statusnachrichten auch dem XQSListener zur Verfügung zu stellen.

3.1.3 Fehlerinformation

Drittprogrammierer müssen, wenn sie den XQSListener implementieren, bisher die Methoden *XQSListener.handleServerReply* und *XQSListener.serverStatusChanged* implementieren. Sie werden nicht aufgefordert, Informationen über aufgetretene Fehler explizit zu behandeln.

Schon um Drittprogrammierern die Möglichkeit zu geben oder um schon bei der Implementation des XQSListeners eine Fehlerbehandlung zu erzwingen, ist die Schaffung einer Methode *XQSListener.handleErrorMessage(XQSStatusMessage err)* nötig. Die Implementation bleibt wie bei allen XQSListener-Methoden dem zukünftigen Programmierer überlassen. Wenn nun eine XQSStatusMessage die Priorität *error* oder *fatal* hält, wird MD*Crypt nicht mehr die Methode *XQSListener.handleServerReply*, sondern die Methode *XQSListener.handleMdCryptException* rufen.

3.2 Ziele der Implementation

Ziel der Arbeiten wird sein, Wege zu finden die die Kommunikation möglichst fehlertolerant gestalten. Einerseits soll die Behandlung von Ausnahmen untersucht und, wenn nötig, an die Gegebenheiten angepasst werden. Hierzu wird in erster Linie die Information, die dem Client zur Laufzeit über den Zustand der Kommunikationsklassen zur Verfügung steht, erweitert werden. Dies soll durch die Implementation eines neuen Objekts erreicht werden, welches als Hauptaufgabe die Übermittlung von Statusmitteilungen haben wird. Weiter sollen die Kommunikationsklassen schon bei der Kontaktaufnahme ein Maximum von relevanter Information austauschen. Als weiterer Punkt soll der Drittprogrammierer bei Implementation des XQSListeners gezwungen werden, Fehlernachrichten grundsätzlich anders als XQS-Ergebnis-Objekte zu behandeln.

3.3 Implementation

3.3.1 Laufzeit

Die Statusangaben, die der XploRe Quantlet-Server ausschreibt, sollen dem XQS-Listener zur Verfügung gestellt werden. Dazu wird vom bereits bestehenden XQSOutputObject ein Objekt des Typs XQSStatusMessage abgeleitet. Der XQS hat somit die Möglichkeit, zur Laufzeit Informationen über den Status des Servers an den XQSListener weiterzugeben. Der Nutzer erhält also Information über die Aktivitäten des Servers und wird so auch bei längeren Programmlaufzeiten erkennen können, ob der Server planmäßig arbeitet.

Bei der Implementation der XQSStatusMessage fällt auf, dass sie in der Lage ist, auch Informationen über den Zustand der anderen beteiligten Programme an den XQSListener weiterzugeben.

3.3.2 Statusnachricht

Beim Einsatz von MD*Crypt zeigt sich, dass an Informationen über den Zustand der Kommunikation nur die nötigsten Angaben an den XQSListener weitergibt. Es ist wünschenswert, auch zusätzliche Nachrichten, die MD*Serv und MD*Crypt zur Verfügung stehen, an den XQSListener weiterzugeben. Ein möglicher Kandidat, um diese Aufgabe zu übernehmen, ist die gerade eingeführte XQSStatusMessage.

Die XQSStatusMessage ist eine Kindklasse des XQSOutputObjekts. Während das XQSOutputObjekt als Attribute seinen Typ und die zu machende Ausschrift als Zeichenkette enthält, wird die XQSStatusMessage als weitere Attribute den Absender der Nachricht sowie ihre Priorität halten. Als Absender werden xqs, md-serv und mdcrypt zulässig sein. Als Prioritäten wird es info, warning, error, fatal geben. So wird die Möglichkeit geschaffen, Informationen von XploRe Quantlet Server, MD*Serv und MD*Crypt über den Zustand der Kommunikation sowie ihre Priorität an den XQSListener weiterzugeben.

Die Information über Absender und Priorität wird innerhalb der Statusnachricht jeweils als Integerwert gehalten werden. Die Werte werden vordefiniert im XQSStatusMessage-Objekt abfragbar sein. So kann der Programmierer den jeweils eigentlichen Wert über Absender und Priorität mit den vordefinierten Werten vergleichen und dadurch den Absender sowie die Priorität der jeweiligen Instanz des XQSStatusMessage-Objekts ermitteln. Dieser Absender- und Prioritätenansatz folgt Ansätzen, wie sie im Packet log4j (Gülcü 2001) für das Schreiben von Logdateien vorgeschlagen wurden.

3.3.3 Generelle Fehlerbehandlung

Es zeigt sich als Erstes, dass das md*crypt-Protokoll zwar verschiedenste Ergebnisobjekte kennt, aber kein Fehlerobjekt. Dies mag einem tiefen Vertrauen in die modernen Netzwerktechnologien geschuldet sein. Punkt eins in den Bemühungen, Fehlertoleranz zu erreichen, muss sein, den XQSListener von auftretenden Problemen informieren zu können. Aufbauend auf dem XQSOutputObject, wurde ein XQSStatusMessage-Objekt abgeleitet, welches zur Laufzeit Informationen über den Server weitergibt. Die XQSStatusMessage eignet sich auch, um Informationen von MD*Serv und MD*Crypt an den Listener weiterzugeben. MD*Crypt kann für unbehandelte Ausnahmen eine Statusnachricht generieren und diese dem XQSListener über die Methode *XQSListener.handleErrorMessage* weiterreichen.

Weiter stellt sich die Frage, ob Ausnahmen und Fehler innerhalb MD*Crypt-Paketes in ausreichendem Maße XQSListener freundlich behandelt werden. Vie-

le Fehler werden bisher mit einem `System.exit(-1)` quittiert, obwohl sie nicht absturzwürdig sind. Selbst wenn sie es wären, ist es Drittprogrammierern nicht zuzumuten, Abstürze in Kauf zu nehmen, obwohl ihnen eigentlich die Möglichkeit gegeben werden sollte, eigene Strategien der Fehlerbehandlung zu finden. Eine einheitlichere Ausnahmenbehandlung ist, wenn nicht nötig, so doch hoch wünschenswert. Zumindest für die Eingabe- und Ausgabemethoden der `SocketIO` Klasse wurde eine einheitliche Ausnahmebehandlung implementiert. Da der `XQ-Server` dem `XQSListener` das Auftreten einer Ausnahme mitteilen muss, erhält er nun sämtliche aufgetretene Ausnahmen. Probleme in den `SocketIO` Methoden führen auf jeden Fall immer zu Problemen in der Ergebnisbehandlung. Die bis zum Auftreten der Ausnahme erhaltenen Informationen werden damit wertlos. Eine Beendigung der Verbindung zum Server ist beim heutigen Stand der `MD*Serv`-Kommunikationsklassen unvermeidlich. Diese zentrale Ausnahmebehandlung führt also dazu, dass erstens der Client grundsätzlich über jede nicht behandelbare Ausnahme informiert wird und zweitens dazu, dass keine unnötigen Beendigungen des Programms irgendwo im Quelltext ausgelöst werden.

Ein weiterer, vorerst kosmetischer, aber bei näherer Betrachtung doch relevanter Punkt ist der Paketname `"mdcrypt"`. Paketnamen mit flachen Namenshierarchien können leicht zu Fehlern in den importierenden Programmen führen, weil die Namen der Paketklassen Gleichheiten aufweisen. Eine Änderung der Packethierarchie soll hier vorgenommen werden. Vergleiche hierzu (Joy 2000).

3.3.4 Information über die Programme

Zu Beginn jeder Kontaktaufnahme zwischen Client und `MD*Serv` findet ein sogenannter Handshake statt. Dieser Handshake dient zuerst dazu, den beteiligten Programmen die Möglichkeit zu geben, die Gültigkeit ihrer Partner zu klären und eine Verbindung möglich zu machen. Bei dieser Gelegenheit können Informationen über die beteiligten Programme ausgetauscht werden. Diese Information ist hilfreich, um fehlerhafte Programmabläufe zu analysieren.

Der Handshake kann Informationen über Versionen und unterliegende Plattformen sowohl der Middleware als auch des `XploRe Quantlet-Servers` an `MD*Crypt` übermitteln. Bevor dieser erweiterte Handshake nicht fehlerfrei durchgeführt wurde, wird keine nutzbare Verbindung aufgebaut. Zuerst muss hierbei `MD*Serv` in die Lage versetzt werden, Informationen über seine Version weiterzugeben. Hierzu wurde ein Objekt entwickelt, das als Eigenschaften eine Versionsnummer und die Versionsnummer der verwendeten Java Virtual Machine hält. Diese Eigenschaften sind `static`, gelten also nicht nur für die Instanz des Objektes, sondern sind für das komplette Programm gleich, dessen Teil die Instanz ist. Außerdem

sind diese Eigenschaften final, das heißt sie dürfen zur Laufzeit des Programms nicht geändert werden. Diese Eigenschaften werden nun während des Handshakes an MD*Crypt übertragen. Dieses wiederum hält auch die Instanz eines Objekts, das die Information über Version und kompilierende Javaumgebung mit den Attributen static und final hält. Während des Handshakes werden nun die Informationen dieser beiden Objekte einem durch den XQSListener abfragbaren Objekt übergeben. Als Nächstes werden während des Handshakes Informationen des Servers erfragt. Dies geschieht durch das Senden von Quantlets. Es werden die Quantlets `getenv("system")`, `getenv("os")` und `getenv("build")` geschickt, deren Ergebnisse als weitere Attribute des Informationsobjektes gesetzt. Als Ergebnis dieses Handshakes stehen dem XQSListener jetzt Informationen über das System des Servers, seine Bezeichnung, seine genaue Version sowie über die Version von MD*Serv, ihre kompilierende Javaversion, die MD*cryptversion und ihre kompilierende Javaversion zur Verfügung. Für die Fehlersuche sind diese Informationen von unschätzbarem Wert.

Als weitere zusätzliche Information übergibt MD*Serv die eindeutige ID, die jedem Client zugewiesen wird, an MD*Crypt. Auch wenn mehrere Clients aktive Sitzungen an einer MD*Serv haben, wird es damit möglich, an MD*Serv Fragen über den Zustand einer bestimmten Sitzung zu stellen.

3.3.5 Kommunikationsfehler

Kommunikation zweier Partner beruht immer auch darauf, dass die Kommunikationspartner die gleichen Regeln der Syntax und Semantik benutzen. Im Fall von XQS-/XQSListener-Architekturen sind diese sowohl serverseitig als auch MD*Cryptseitig implementiert. Die Syntax ist clientseitig in XJprotocol.java implementiert, und die Sinnzusammenhänge sind die gesendeten XQSObjekte. Treten nun Fehler beim Transport der Sinnzusammenhänge auf, können sie zerstört werden. Dieses Phänomen des fehlerhaften Informationstransports wird Kanalauschen genannt. Beim heutigen Stand der Entwicklung von MD*Serv sind solche zerstörten Sinnzusammenhänge nicht reparabel. Es bleibt als Mittel der Fehlerbehebung nur noch, die Verbindung und die Information des aufsitzenden XQS-Listeners abzubrechen.

Die XQS/MD*Crypt-Kommunikation basiert auf einem Bytestream. MD*Crypt hat zu jeder Zeit Informationen über die Anzahl der Bytes, die zu dem zu lesenden Objekt oder Teilobjekt gehören. Die Lesemethoden MD*Crypts lesen immer genau die Anzahl Bytes, die sie erwarten. Grundsätzlich können zwei Arten von Fehlern auftreten. Bei Fehler eins können durch fehlerhafte Kommunikation zu viele Bytes in den Stream geschrieben worden sein. Dann wird das nächste Kom-

mando nicht richtig gelesen werden (vergleiche Grafik 3). Wenn aber kein gültiges Kommando gelesen wurde, ist klar, dass ein Fehler aufgetreten ist. Dieser kann dann entsprechend behandelt werden. Es kann etwa die Verbindung unterbrochen werden. Auf jeden Fall ist es mit Hilfe der `XQSStatusMessage` möglich, den `XQSListener` über den aufgetretenen Fehler zu informieren.

Problematischer ist Fehler zwei; hier stehen weniger Bytes als erwartet im Stream. Dieser Fehler tritt weitaus häufiger auf, etwa wenn der Server unerwartet beendet wird, ohne dass `MD*Serv` Information darüber hat; er ist schwerer zu behandeln. Die entsprechende Methode versucht, das fehlende Byte vom Stream zu lesen und blockiert so den Programmablauf. Hierbei besteht das Problem, dass nicht klar ist, ob ein Fehler aufgetreten oder ob die Verbindung zu langsam ist.

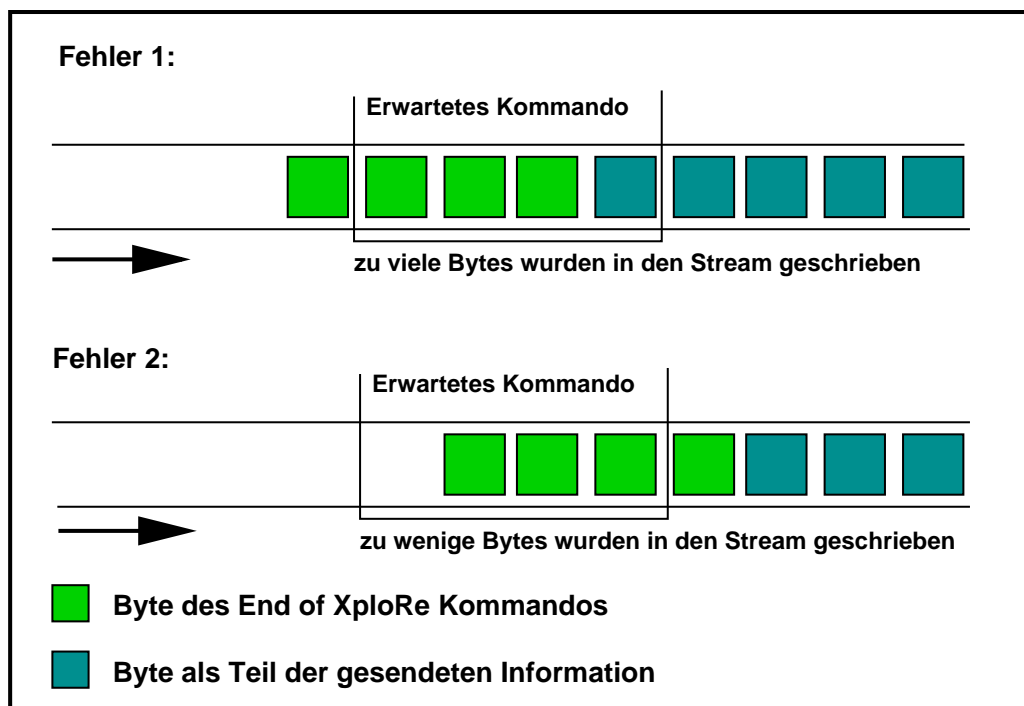


Abbildung 3: Zwei grundlegende Fehlerquellen der Bytestreamkommunikation

Welche Strategien können das Blockieren verhindern? Es besteht grundsätzlich die Möglichkeit, einen Thread, das heißt ein unabhängig laufendes Teilprogramm, zu starten, welches in definierten Zeitabständen überprüft, ob die korrespondierende Methode noch liest. Das Vorgehen ist wie folgt: Direkt vor dem Aufruf der Lesemethode wird der Kontrollthread gestartet. Bei planmäßigem Beenden des Lesens wird der Thread gestoppt. Tritt ein Problem, das zu Fehler zwei führt, auf,

wird das unabhängig laufende Teilprogramm nur dadurch, dass es nicht gestoppt wird, Information darüber weitergeben können, dass eine Lesemethode länger als der definierte zeitliche Schwellwert mit dem Lesen von Bytes zubringt. Der Clientprogrammierer kann eine solche Ausnahme behandeln. Er kann sie ignorieren oder, angepasst an die Plattformgegebenheiten, behandeln.

Da aber Threads gerade in Szenarien, wie sie bei dem Microsoft Excel Add In Rex zu finden sind, nicht zum Einsatz kommen sollten, muss immer die Möglichkeit bestehen, solche threadbasierten Kontrollstrukturen bei der Instanziierung des MD*Crypt-Objekts XQSServer-Threadstrukturen grundsätzlich abzuschalten. Dies sollte idealer Weise über einen entsprechenden Konstruktor erfolgen.

Für die neu zu rufenden kontrollierenden Threads treten freilich wieder neue Fehlerquellen auf; man muss also fragen, ob es weitere threadfreie Möglichkeiten gibt, blockierte Lesemethoden zu unterbrechen. Eine weitere, leichter zu implementierende Variante sei jetzt beschrieben. Das Problem des Fehlers zwei ist eigentlich der, dass die Lesemethode blockiert. Es existiert aber in der Klasse Socket eine Eigenschaft SoTimeout. Dieser Integerwert gibt die Zeit an, die eine Lesemethode brauchen darf, um zu lesen. Wenn nun im Laufe der Implementationen dieser Arbeit ausreichend schnelle Lesemethoden implementiert sind, kann man diesen Wert setzen. Wird dieser Zeitwert dann überschritten, bricht die Methode ab, und eine Ausnahme wird ausgelöst. Diese kann dann nur dazu führen, dass die Verbindung zum Server abgebrochen und der XQSListener entsprechend informiert wird (siehe auch (Reilly 1999)).

3.4 Ergebnis der Implementation

Mit der XQSStatusMessage ist eine Klasse entstanden, die nicht nur Laufzeitinformationen des Servers halten kann, sondern auch weitere Informationen über den Zustand der Kommunikation zwischen MD*Crypt und XQS tragen sowie über die Methode *XQSListener.handleServerReply(XQSObject xobj)* an den XQSListener weitergegeben werden kann an den Listener. Weiter ist mit dem TimeCheck-Objekt ein Mittel entstanden, um Warnungen an den Listener zu senden, wenn bestimmte Zeiten bei der Behandlung von Quantlets überschritten werden. Da die Szenarien, in denen die Clients laufen werden, nicht bekannt sind, sollte die Information über eine Laufzeitausnahme von den zukünftigen Clientprogrammierern behandelt werden.

Sämtliche Schreib- und Lesemethoden reichen aufgetretene Ausnahmen jetzt an das XQServer Objekt durch, wo sie in der Methode *XQServer.handleEx(Exception e)* zentral behandelt werden. Zusammen mit dem hinzugekommenen Sockettimeout gelingt es nun, das Blockieren des Streams für fast alle Szenarien zu unterbinden.

Clientprogrammierer werden schon bei der Implementation des XQSListeners gezwungen, Fehlernachrichten grundsätzlich anders als andere XQS-Objekte zu behandeln. Allein diese Maßnahme erhöht die Fehlertoleranz zur Laufzeit. Das allgemeine Verhalten von MD*Crypt beim Auftreten von Ausnahmen konnte verbessert werden; dadurch konnte auch die Stabilität der aufsitzenden Clients verbessert werden. Um Namenskonflikte beim Import des MD*Crypt Packetes zu vermeiden, wurde der Packetname in `com.mdcrypt.mdcrypt` geändert. Für die MD*Serv Klassen wurde, abgesehen von der Main-Klasse, ebenfalls ein Packet angelegt, welches unter dem Namen `com.mdcrypt.mdserv` zu importieren ist. So kann die Distribution der Pakete fehlertoleranter erfolgen. Gerade für MD*Crypt ist dies ein hoch relevanter Punkt.

4 Geschwindigkeit

4.1 Methoden der Bestandsaufnahme, Bestandsaufnahme

Um zeitliche Ineffizienzen in Quelltexten zu finden, ist es hochrelevant, flexible, genaue und effiziente Zeitmessmethoden zur Verfügung zu stellen. Im Rahmen dieser Arbeit werden zwei Ansätze verfolgt werden. Erstens sollen Methoden bereitgestellt werden, die den Zeitverlauf direkt aus dem Quelltext ermitteln und so Messdaten über jeden Teilalgorithmus des Programms bereitstellen. Als Zweites soll ein eher an den Anwendungsfällen orientiertes Verfahren angewandt werden. Es soll die Zeit, welche gebraucht wird, um eine typische Aufgabe zu erfüllen, als Maß verwandt werden. Diese Methode wird im folgenden 'Methode der Referenzquantlets' genannt werden.

4.1.1 Methode 'echter' Zeitmessungen am Code

Da in diesem Projekt die Möglichkeit besteht, den Quelltext zu verändern, kann man Zeiten, die für einzelne Funktionalitäten gebraucht werden, direkt am Quelltext messen. Java bietet die Möglichkeit, über die Methode `System.currentTimeMillis` die Systemzeit des Systems, auf dem das Programm läuft, zu ermitteln. Mit dieser Option ist es dann natürlich auch möglich, Zeitabstände zu messen.

Das Halten der gemessenen Zeitabstände sollte idealer Weise in einer Datei erfolgen. Diese Datei muss Minimalinformationen zu den gemessenen Zeitabständen und der korrespondierenden Methode halten. Da die hier gemessenen Methoden immer Bytes lesen, erweist es sich als notwendig, auch die Anzahl der gelesenen Bytes festzuhalten.

Die nötigen Funktionalitäten wurden in einer Java-Klasse mit Namen `TimeLogFile` implementiert. Diese Klasse hält abgeleitete Methoden der Klasse `LogFile`, um eine Datei schreiben zu können. Als erweiterte Methoden stehen Methoden zum Messen von Zeitabständen zur Verfügung. Die Zeitnahme erfolgt mit Methoden zum Ermitteln von Startpunkten sowie aus Methoden, die für jedweden gehaltenen Startpunkt die abgelaufenen Zeit zurückliefern. Der Ablauf einer Zeitabstandsmessung würde dann folgendem beispielhaften Algorithmus folgen:

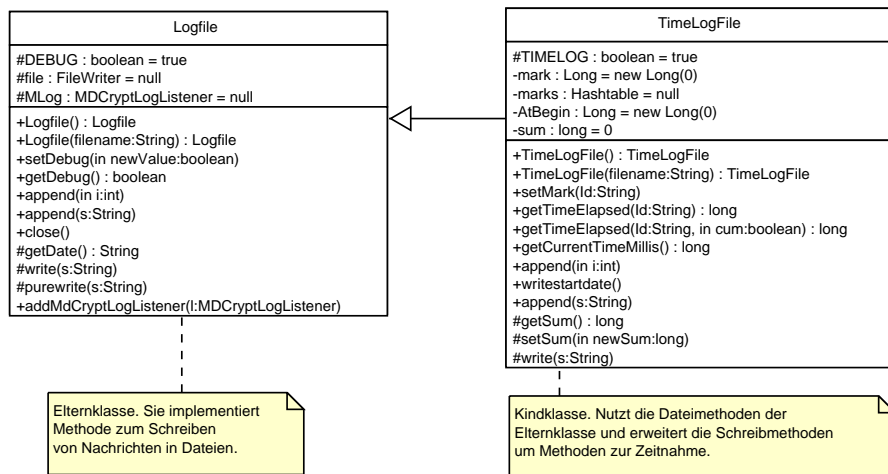


Abbildung 4: Klasse LogFile und ihre abgeleitete TimeLogFile

```

package mdcrypt;
...
import java.net.Socket;
import java.io.;
class SocketIO {
...
private BufferedDataOutputStream dataOut;
...
...
void writeToken(int token)
{
    XQServer.timelog.setMark("writetok");
    try{
        dataOut.writeInt(token);
        dataOut.flush();
    }
    catch (IOException e)
    {
        XQServer.logfile.append("writeToken: " + e);
    }
    XQServer.timelog.append("\ "writetok \ "\t "+
        XQServer.timelog.getTimeElapsed("writetok")+"\t "+4;
    }
}
}
  
```

Zuerst wird mit `XQServer.timelog.setMark(String Key)` die aktuelle Systemzeit verknüpft und mit dem Schlüssel "Key" in einer Hashtable abgelegt. Eine Hashtable erlaubt das Speichern, Halten und Abfragen von Schlüssel:Wert Paaren. (verleiche auch Abbildung 4). Nach Ablauf des zu messenden Algorithmus kann man die vergangene Zeit über die Angabe des Schlüssels abfragen. Die Methode `getTimeElapsed(String Key)` liefert die vergangenen Millisekunden als Wert zurück und übernimmt die Verwaltung der Hashtable, was hier so viel heißt wie: Sie entfernt den Eintrag "writetok" und ihren Wert aus der Hashtable. Die Methode `XQServer.timelog.append()` als geerbte Methode schreibt eine Zeichenkette in eine Datei. In diesem Fall wird das Wertetripel "Methodenschlüssel, benötigte Millisekunden, Anzahl der gelesenen Bytes" geschrieben. Diese Form wird für alle Lese- und Schreibmethoden von MD*Serv und MD*Crypt verwendet. Als Ergebnis entsteht eine Datei, die wie eine .dat-Datei aufgebaut und so mit statistischer Software wie XploRe leicht auszuwerten ist. Die erste Spalte identifiziert den gemessenen Lese- oder Schreibalgorithmus, die zweite Spalte gibt die Zeit in Millisekunden an, welche der Algorithmus benötigte, und die dritte Spalte gibt an, wieviel Bytes der gemessene Algorithmus gelesen hat. Diese Form wird die Grundlage späterer Analysen sein. Sie lässt sich soweit verfeinern, dass selbst in einzelnen Lesemethoden zwischen Wartezeiten und reinen Lesezeiten unterschieden werden kann. Dies wird später bei der Situationsanalyse relevant werden.

```

...
"readtok"      111  4
"readint"      15   20
"readbyte"     20   2
"readtok"      0    4
"readtok"      0    4
"readtok"      0    4
"readftasdbl"  47   2800
"readftasdbl"  13   2800
"readftasdbl"  20   2800
"readint"      20   2800
"readtok"      0    4
"readtok"      0    4
"writetok"     0    4
"writebyte"    1    14
...

```

4.1.2 Methode der 'Referenzquantlets'

Die technische Realisierung der Zeitmessung folgt weitestgehend der vorher beschriebenen Methode. Der Unterschied besteht generell darin, die Bearbeitungszeit für das Abarbeiten eines kompletten Quantlets zu messen. Die Schwierigkeit, die sofort auftaucht, ist, dass die Ergebnisbehandlung Sache des Clients ist. Der Client und seine Art der Ergebnisbehandlung sind aber noch gar nicht bekannt. Da hier jedoch die Methoden der Kommunikation untersucht werden, umgeht man dieses Problem am besten dadurch, dass man in Messreihen die Ergebnisbehandlung auf das Nullsetzen des Ergebnisobjekts beschränkt. Auf diese Weise können Verzerrungen der Ergebnisse durch Ergebnisbehandlungsroutinen des Clients verhindert werden, und das Messergebnis wird eher den Zustand der Kommunikation widerspiegeln.

4.2 Ziele der Implementation

Ziel der Quelltextänderungen ist es, zeitliche Ineffizienzen in den verwendeten Algorithmen zu identifizieren und, soweit möglich, zu beheben. Dazu wird es nötig werden, die Lese- und Schreibmethoden zu verändern. Diese Änderungen dürfen sich nicht auf die Richtigkeit des Ergebnisses auswirken.

Im Speziellen wird versucht werden, Engpässe in der Kommunikation zwischen Client und Server zu identifizieren.

4.3 Implementation

4.3.1 Identifizieren der Engpässe

Um eine zielgerichtete Änderung der Schwachstellen zu ermöglichen, ist im ersten Schritt genau zu analysieren, an welchen Stellen der Kommunikation die Engpässe liegen. In der vorliegenden, dreistufigen Architektur kommen vier Kandidaten zur Überprüfung in Frage.

Das Schreiben und Lesen des XQS (im gegebenen Rahmen nur indirekt zu ermitteln, da der Quelltext nicht zur Verfügung steht)

Das Schreiben und Lesen der MD*serv vom Client zum Server

Das Schreiben und Lesen der MD*serv vom Server zum Client

Das Lesen und Schreiben von MD*Crypt

Man sollte die Kommunikationsmethoden vom Client zum Server von denen vom Server zum Client trennen, weil die Richtungen durch grundsätzlich unterschiedliche Kommunikationsszenarien gekennzeichnet sind (betrachte Abbildung 5). Während vom Client zum Server Kommandos in Form von Zeichenketten geschickt werden, liefert der XQS hochkomplexe Ergebnisobjekte zurück. Die effiziente Weitergabe dieser Objekte muss durch besonders angepasste Algorithmen erfolgen.

Methoden, die Information zum Server übertragen, sind vorerst keine vielversprechenden Kandidaten für eine spürbare Verbesserung von Quantletbearbeitungszeiten, denn sie schreiben in momentanen Szenarien mengenmäßig einfach zu wenig Bytes. Ergebnisse der späteren Analyse müssen aber trotzdem auch für diese zum Server schreibenden Methoden gelten, weil es zukünftig für MD*Crypt auch Einsatzgebiete geben wird, in denen höhere Datenmengen zum Server transportiert werden müssen. Für den Moment allerdings wird eine Engpasssuche auf dem Weg vom Server zum Client mehr versprechen.

Messungen direkt an den Lese- und Schreibmethoden MD*Crypts können wertvolle Anhaltspunkte für die Identifizierung der Engpässe liefern. Das getrennte Messen von Wartezeiten und reinen Lese- oder Schreibzeiten ergab, dass MD*Crypts Lesemethoden, abhängig von der zu lesenden Anzahl von Bytes, erhebliche Zeit mit dem Warten auf die zu lesenden Bytes zubrachten. Folgerichtig muss der Engpass in der Architektur weiter vorn zu suchen sein. Der nächste Kandidat wären dann die Kommunikationsmethoden der MD*Serv-Klassen.

Um das Verhalten der MD*Serv Kommunikationsklassen zu analysieren, wurden, ähnlich wie schon bei den MD*Crypt-Klassen, Zeitnahmemethoden in einer Ableitung der LogFile-Klasse implementiert. Was sofort auffiel, war das häufige Lesen vom XplDataInputStream. Auf Solarisplattformen ist dieses Verhalten nicht zu beobachten. Sowohl für Linux- als auch für Windowsplattformen war dieses Verhalten zu beobachten. Es wurde also unumgänglich, den Lesealgorithmus, den MD*Serv vom XploRe Quantlet Server zu MD*Crypt verwendet, genauer zu betrachten, um auf allen Plattformen genügend hohe Schreibgeschwindigkeiten zu erzielen.

4.3.2 Mankos des `java.io.Data(IO)Streams`

Ein vielversprechender Kandidat für Geschwindigkeitsoptimierungen sind die Methoden der Klasse `java.io.DataInputStream` respektive `java.io.DataOutputStream`. In den XQS/MD*Crypt Kommunikationsstrukturen ist es sehr oft nötig, große Mengen von Datentypen höherer Komplexität als Byte zu übertragen. Methoden zur Übertragung liefern die Methoden der Klassen

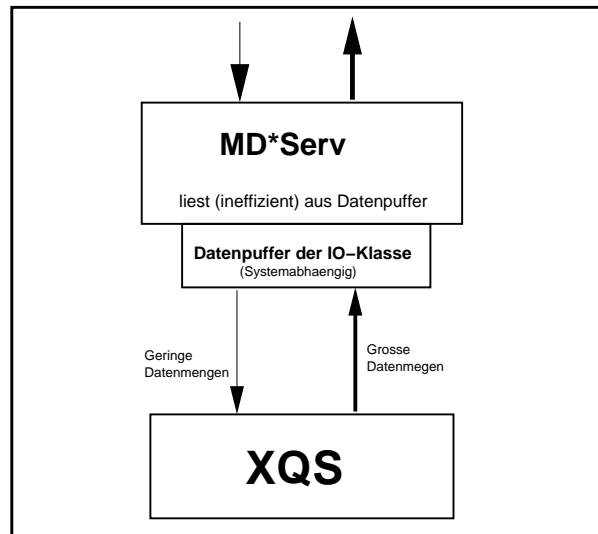


Abbildung 5: Engpass in der XQS/MD*Serv Kommunikation

java.io.DataInputStream und java.io.DataOutputStream. Allerdings bieten diese Klassen keine Methoden, um Felder von höheren Datentypen zu übertragen. Das Übertragen von einer höheren Anzahl dieser Datentypen erfordert das wiederholte Aufrufen von Lese- oder Schreibmethoden. Das Aufrufen von Lese- oder Schreibmethoden geht aber immer mit dem Aufruf von Systemmethoden einher. Diese Methoden sind aber, abhängig von den unterliegenden Betriebssystemen, sehr zeitintensiv.

Um also deren Bearbeiten ohne zu häufiges Aufrufen der Lese- und Schreibmethoden zu ermöglichen, ist es erforderlich, das Umwandeln der höheren Datentypen zu Bytefeldern vom Schreiben derselben zu trennen. Geeignete Methoden liefern die Klassen BufferedDataInputStream und BufferedDataOutputStream. Die hier zur Verfügung gestellten Methoden ermöglichen es, auch Felder höherer Datentypen bei minimalem Aufruf von plattformabhängigen Lese- und Schreibmethoden zu übertragen. Die BufferedDataOutputStream Methoden erreichen dies, indem sie das Umwandeln der Datentypen in Bytefelder vom Schreiben trennen. So erreichen sie beim Senden höherer Datentypen und, abhängig von deren Menge, rund eine zehnfache Schreibgeschwindigkeit.

4.3.3 BufferedDataInputStream / BufferedDataOutputStream

Der Kern der vorgenommenen Änderungen ist die Einbindung der Klassen BufferedDataInputStream und BufferedDataOutputStream. Diese Klassen wurden ge-

schaffen, um im Rahmen eines Linuxclusters genügend schnelle Methoden der Datenübertragung bereitzustellen sowie Methoden zu schaffen, die auch weitere, im `DataInput(Output)Stream` nicht implementierte Datenstrukturen effizient schreiben oder lesen.

Das VisAD Projekt (siehe: Ressourcen im World Wide Web), befasst sich mit der Visualisierung von Daten in verteilten Systemen und stand so vor einem ähnlichen Ein- und Ausgabegeschwindigkeitsproblem wie MD*Crypt. Im Rahmen dieses Projektes wurden die Klassen, welche für die Entwicklung der `com.mdcrypt.mdcrypt.BufferedDataInputStream` und `- BufferedDataOutputStream` Klassen Pate standen, entwickelt. Die `BufferedData(I/O)Stream` Klassen zeigen auf beispielhafte Weise, wie man bekannte Probleme der Java Ein- und Ausgabeklassen umgeht oder vermeidet.

Die `BufferedDataInputStream` und `BufferedDataOutputStream` zeichnen sich dadurch aus, die Vorteile des `java.io.DataInputStreams` und des `java.io.BufferedInputStreams` zusammenzuführen, ohne die Verluste aus Kapselung des gepufferten Streams durch den `DataInptuStream` in Kauf zu nehmen.

Beispielhaft soll hier das Schreiben eines Feldes von Fließkommawerten betrachtet werden. Benutzt man den von Java bereitgestellten `java.io.DataOutputStream`, ist man gezwungen, die Methode `writeInt(int i)` für jedes Element des Feldes einzeln aufzurufen. Es wird also jedes Element einzeln in ein Bytefeld der Länge vier transformiert und dann durch den Aufruf einer `write(byte[] b)`-Methode geschrieben. Für jedes Element wird also eine eigene Schreibmethode gerufen.

`BufferedDataOutputStream` bietet eine Methode `writeFloatArray(float[] f)`, deren Vorteil es ist, das Fließkommafeld als Ganzes in einem Arbeitsgang in ein Bytefeld zu transformieren und dann die Schreibmethode `write(byte[] b)` nur einmal aufzurufen. Je nach Szenario, also der Länge des zu schreibenden Fließkommafeldes, erreicht man so erhebliche Geschwindigkeitszuwächse. Siehe sowohl (Donehower 2001) als auch (McCluskey 1999). Abgesehen von Bytefeldern, die in den `java.io-Streamklassen` immer Feldweise les- und schreibbar sind, wirkt sich diese Verbesserung bei allen Feldern von Javadatentypen auf gleiche Weise aus.

Folgende Quelltextfragmente zeigen die Unterschiede in der Implementierung:

Fließkommawertfeld, geschrieben mit Methoden der `java.io.DataOutputStream`-Klasse:

```
...
DataOutputStream dataOut;
...
void writeFloat(float d[])
{
    XQServer.timelog.setMark("writeflt");
    try{
        dataOut.writeInt(d.length);

        for (int i=0; i<d.length; i++)
            dataOut.writeFloat(d[i]);

        dataOut.flush();
    }
    catch (IOException e)
    {
        XQServer.logfile.append("writeFloat: " + e);
    }
    XQServer.timelog.append("\ "writeflt \ "\t "+
        XQServer.timelog.getTimeElapsed("writeflt")+ "\ t "+(d.length*4));
    }
}
...
```

Fließkommawertfeld, geschrieben mit Methoden der `BufferedDataOutputStream`-Klasse:

```
...
BufferedDataOutputStream dataOut;
...
void writeFloat(float d[])
{
    XQServer.timelog.setMark("writeflt");
    try{
        dataOut.writeInt(d.length);

        dataOut.writeFloatArray(d);
    }
}
```

```

        dataOut.flush();
    }
    catch (IOException e)
    {
        XQServer.logfile.append("writeFloat: " + e);
    }
    XQServer.timelog.append("\ "writeflt \ "\t "+
        XQServer.timelog.getTimeElapsed("writeflt")+ "\ t "+(d.length*4));
    }
}
...

```

4.3.4 Bytefelder MD*Cryptseitig

Auf MD*Cryptseite entsteht ein weiterer Engpass beim Lesen von Zeichenketten als Bytearrays. Dies geschah bisher durch das Verwenden einer *for*-Schleife. Es entsteht dasselbe Problem wie beim Schreiben von höheren Java Datentypen. Für jedes Byte wird wieder eine Lesemethode gerufen. Dieses Rufen von Systemmethoden ist aus Java heraus per se ein Geschwindigkeitsproblem. Es musste also ein Algorithmus gefunden werden, der die erwarteten Bytes mit dem nur einmaligen Aufruf einer Lesemethode liest. Vergleiche hier sowohl (Kluge 1997) als auch (McCluskey 1999). Die folgenden Quelltextfragmente zeigen erstens den vorgefundenen Algorithmus sowie zweitens die vorgenommene Änderung. Das blockweise Lesen von Bytearrays erreicht rund dreifach schnellere Lesezeiten in Szenarien, wie sie beim XQC auftreten.

Lesen von Bytefeldern mit einer *for*-Schleife:

```

...
for (i = 0 ; i < d.length; i++)
d[i] = dataIn.readByte();
...

```

Das folgende Quelltextfragment zeigt den neuen Algorithmus. Auf Windowsplattformen tritt hierbei ein Fehler auf, weswegen das Lesen mit der Methode *ReadAgain* nochmals überprüft werden muss. Gegeben, es liegen mehr Bytes im Stream, als gelesen werden sollen, wird einmal die Lesemethode *Read* aufgerufen:

```
...
while(!done){
    datawait = dataIn.available();
    if( datawait >= size){
        b_read = dataIn.read(d);
        //some win platforms error some how ; check for missing bytes again
        while(b_read < size)
            b_read = b_read + readAgain( b_read, size,d );

        done = true;
    }
    else {
        Thread.sleep(10);
    }
};
...
```

Mit den Methoden der Klassen *BufferedDataInputStream* und *BufferedDataInputStream* lassen sich die bisherigen Lese- und Schreibalgorithmen auf gleiche Weise verbessern. Die Geschwindigkeitserhöhung beruht wieder darauf, die plattformabhängigen read-Methoden möglichst selten zu rufen.

4.3.5 MD*Serv

Um den MD*Serv-Engpass unter einigen Systemen zu beheben, waren weitere Maßnahmen nötig. Zuerst musste der Algorithmus dahingehend geändert werden, dass die im Puffer liegenden Bytes auch wirklich in einem Zug lesbar sind. Der Algorithmus, wie er bisher implementiert war, schien genau das ausführen zu wollen. Er ermittelte mit der Methode `java.io.InputStream.available()` die zur Verfügung stehenden Bytes, legte dann ein Bytefeld von entsprechender Größe an und benutzte sodann die Methode `java.io.InputStream.read(byte[] b)`, um das Bytearray mit den zur Verfügung stehenden Bytes zu füllen. Unter Linux und Windows liest diese Methode aber nur ein Feld der Größe 1024. Unter Linux und Windows führt dies zu einem horrenden Geschwindigkeitsverlust (Donehower 2001),(Kluge 1997). Für zum Beispiel 102400 bytes, die schon zur Verfügung stehen, muss die `java.io.InputStream.read(byte[] b)` hundertmal gerufen werden. Die Methode, die Java IOStreams hier vorsehen, ist die Methode `java.io.InputStream.readFully(byte[] b)`. Unter Verwendung dieser Methode konnte dieser Engpass für Linux-Plattformen schon beseitigt werden. Unter Microsoft's Windows ist aber selbst mit dieser Änderung keine Geschwindigkeitsverbesserung

zung zu beobachten. Unter Windows las MD*Serv weiterhin nur 1024 Bytes pro Methodenaufruf. Es zeigt sich, dass dies daran liegt, dass Java Data(IO)Streams unter Windows nur einen Puffer von 1024 Bytes zur Verfügung haben. Um dieses Problem möglichst effizient zu beheben, wurden wieder die Klassen `BufferedDataInputStream` und `BufferedDataOutputStream` verwendet. Damit ist es möglich, effizient, also ohne Kapselung von IO-Streams, die sowohl Pufferung vollständig zu unterstützen und dennoch höhere Datentypen zu bearbeiten. Mit diesen Klassen war es möglich, unter Windows explizit einen Puffer für einen Datenstrom zuzuweisen. Durch diese Maßnahme war es nun auch unter Windows möglich, auf Kosten des verwendeten Arbeitsspeichers die Rufe der Lesemethoden zu minimieren. Weiterhin war die Einbindung der Klassen `BufferedDataInputStream` und `BufferedDataOutputStream` erforderlich, um auf MD*Serv und MD*Crypt mit den gleichen Methoden zu lesen und zu schreiben.

4.4 Ergebnis der Implementation

4.4.1 Technische Umgebung

Die im Folgenden gemessenen Werte wurden unter einem Linux 2.4 Kernel mit Suns Java 1.3.1 Virtual Machine auf einem AMD Athlon 1.2GHz mit 256 Megabyte Arbeitsspeicher erreicht. Andere Betriebssysteme erreichen ähnliche Werte bei ähnlichen Hardwarekonfigurationen.

Für andere Hardwarekonfigurationen können die Ergebnisse stark variieren, obschon ein Geschwindigkeitszuwachs wohl immer messbar bleibt. Auf Sun Solaris Maschinen wird die geringste Veränderung zu messen sein, weil die Änderungen des MD*Servlesealgorithmus hier nicht tragen. Diese systemabhängigen Parameter müssen bei der folgenden Diskussion immer berücksichtigt werden.

4.4.2 Analyse

Zwei Hauptszenarien werden bei der Ergebnisanalyse relevant sein. Erstens wird die Übertragung von Grafikkontexten untersucht, und zweitens wird die Übertragung reiner TextOutput-Objekte betrachtet werden müssen. Technisch unterscheiden sich diese durch leicht verschiedene Behandlungsroutinen, in denen verschiedene Methoden verwandt werden. Zuerst sollen die Auswirkungen der Änderung auf die Übertragung von Grafikkontexten untersucht werden.

Durch die Beseitigung der ineffizienten Lesealgorithmen MD*Servs und durch die Verwendung der Klassen `BufferedDataInputStream` und `BufferedDataOut-`

putStream in MD*Crypt und MD*Serv wurden erhebliche Geschwindigkeitszuwächse realisiert. Diese konnten an den Kommunikationsmethoden selber und auch an Quantletbearbeitungszeiten gemessen werden.

Grafikkontexte. In Grafikkontexten werden große Mengen verschiedener Datentypen übertragen. Änderungen sowohl der Input-/Output-Algorithmen als auch der verwendeten Input-/Output-Klassen werden hier besonders wirksam. Tabelle 1 stellt die Zeiten für die Behandlung eines Quantlets der Form: show(d,1,1,A) dar, wobei die Dimension von A gleich $3 \times n$ mit wachsenden n ist.

show(d,1,1,A) mit dim(A)	Quantlet Bearbeitung	Quantlet Bearbeitung
	MD*Crypt alt in millisec.	MD*Crypt neu in millisec.
3×10	130	120
3×50	101	90
3×100	242	150
...
3×50000	91536	672
3×75000	139568	970
3×100000	187333	1361

Tabelle 1: Quantletbearbeitung bei Grafikkontexten

Textausgabe. Ein weiteres Einsatzszenario fordert das Empfangen größerer Mengen von Textoutput des XploRe-Quantlet-Servers. Die Menge an Daten ist hier in der Regel nicht so hoch wie bei Grafikkontexten. Da der XploRe-Quantlet-Server eine maximale Textausgabemenge hat, lassen sich nicht beliebig lange Texte übertragen. Mit den Servereinstellungen die Textausgabelänge betreffend bei Verbindungsaufnahme lassen sich folgende Werte messen. Tabelle 2 zeigt die Quantletbehandlungszeiten für das Quantlet "normal(1000,3)".

Textausgabe bei "normal(1000,3)"	
Quantlet Bearbeitung MD*Crypt alt in millisecc.	Quantlet Bearbeitung MD*Crypt neu in millisecc.
3193	730
2864	703
3046	596
2708	464
2873	575
2827	499
2721	793
2876	521
2845	528
2708	516
2911	518

Tabelle 2: Quantletbearbeitung bei Textausgaben

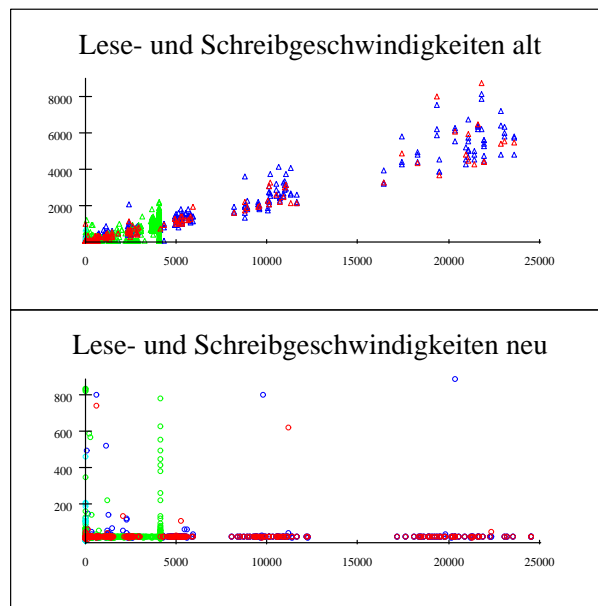


Abbildung 6: Für dasselbe Quantlet gemessene Verweilzeiten nach gelesenen Bytes

Leistung der Input-/Output-Methoden. Die gemessenen Quantletbehandlungszeiten setzen sich natürlich aus schneller laufenden Ein- und Ausgabemethoden zusammen. Wie diese ihre Verhalten geändert haben, soll im Folgenden kurz beleuchtet werden. Die Zeitmessungen im Quelltext lieferten einiges an Daten, um Analysen vornehmen zu können. Die Abbildungen 6 und 7 zeigen das Laufzeitverhalten dieser Methoden im Vergleich. Die in Abbildung 6 abgetragenen Daten bezeichnen die Dauer von Lese- und Schreibmethoden relativ zu der von ihnen bearbeiteten Bytemenge. Da Methoden wie `readByte()` in anderen Szenarien, wie zum Beispiel `readDoubleAsFloat()`, gerufen werden, sind die Schreib- und Lesemethoden gruppiert und farblich verschieden dargestellt. Rot bezeichnet die Methode `readInt()`, blau die Methode `readDoubleAsFloat()`, grün die Methode `readByte(byte[] b)`. Die `readFloatAsDouble()` Methode wird nur in Grafikkontexten gerufen, weswegen sie bei einzelnen Aufrufen auch hohe Bytemengen bearbeiten muss. Die `readByte(byte[] b)`-Methode wird zum Lesen von Zeichenketten verwendet. Da Zeichenketten serverseitig in ihrer Länge begrenzt sind, sind die zu bearbeitenden Bytemengen auch begrenzt; `readInt()` wird sowohl in Grafikkontexten als auch zur reinen Ablaufkontrollen innerhalb des Protokolls verwendet. Die Methode `readInt()` muss also sowohl sehr kleine als auch sehr grosse Bytemengen bearbeiten.

Abbildung 7 zeigt die Methoden der alten wie auch der neuen MD*Crypt-Version im direkten Vergleich. Die farbige Darstellung der alten Methoden ändert sich derart, dass readFloatAsDouble() der Farbe magenta, readByte(byte[] b) der Farbe gelb und readInt() der Farbe schwarz entspricht.

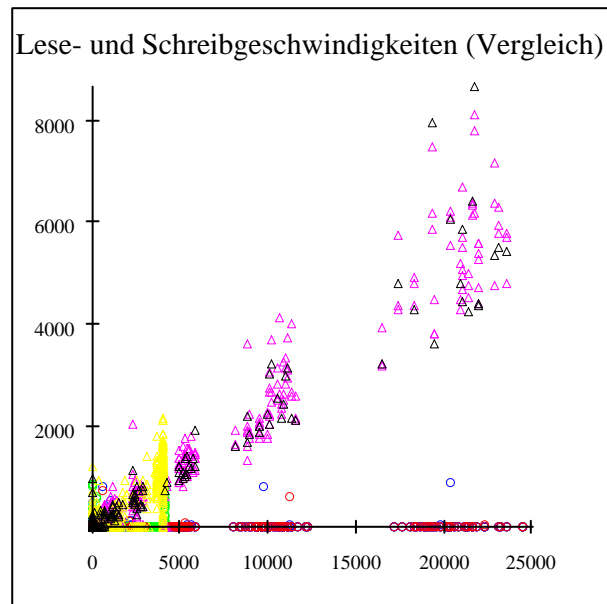


Abbildung 7: Vergleich der neuen und alten Lesemethoden

5 Funktionalitätserweiterung

5.1 Handlungsfelder

Ein Hauptpunkt jeder Optimierung eines Protokolls muss es sein, die implementierte Kommunikationsstruktur immer wieder an den Anforderungen der Nutzungsszenarien zu messen. In dieser fortlaufenden Neudefinierung muss immer wieder überprüft werden, ob erstens die Strukturen den sich ändernden Anforderungen noch genügen und zweitens, wie durch Neuimplementation oder Abänderung der bestehenden Strukturen die Nutzbarkeit der Protokollklassen erhöht werden kann.

Im Folgenden sollen nun Handlungsfelder zur Optimierung der vorliegenden Protokollklassen identifiziert werden, um Wege aufzuzeigen, wie diese an sich ändernde Bedingungen angepasst werden können.

5.1.1 Erweiterung der Kommunikationsstruktur

Eine weitere, entscheidende Frage bleibt die der Ausweitung der Kommunikationsstruktur. Das bisherige Senden von Kommandos als Zeichenketten erweist sich in Szenarien wie dem Rex-Excel-Add-in als unbefriedigend. Da Excel selbst in der Lage ist, große Datenmengen zu halten, wird es nötig, diese auch zum Server zu schicken. Dieses Schicken von Fließkommawerten als Zeichenkettenrepräsentation geht sowohl mit Präzisionsverlusten als auch mit Geschwindigkeitseinbußen einher. Dies gilt für alle Szenarien, in denen Fließkommawerte zum Server geschickt werden müssen.

5.1.2 Abbildung der XploRe Kommandos

Wenn man die Sprache von XploRe benutzt, ist es wünschenswert, auch die vollständige Syntax abzubilden. In diesem Zusammenhang entsteht Handlungsbedarf im Bereich des AddData-Kommandos, welches Bestandteil der XploRe-Sprache ist. Weiter entstanden neue Elemente wie das SlideValueObject, welches Verbesserungen im Bereich der Interaktivität erreichen soll.

Zur Erhöhung der Interaktivität sind neue Steuerelemente Bestandteil der XploRe-Sprache geworden. Clientprogrammierern wird die Möglichkeit gegeben werden müssen, diese Bestandteile zu benutzen. Zu diesen gehört das SlideValueObject. Es stellt ein Dialogobjekt dar, welches dem Nutzer erweiterte Interaktionsmöglichkeiten bieten soll. Das SlideValueObject gehört im MD*Crypt-

Kontext zu den Dialogobjekten. Diese Objekte werden zur Laufzeit eines Quantlets zwischen XQS und Client ausgetauscht, um dem Nutzer die Möglichkeit zu geben, auf Parameter der ablaufenden Berechnung Einfluss zu nehmen. Insbesondere in der Lehre entstehen die Einsatzgebiete dieser Objekte.

Mit der Implementation des SlideValueObjects sollen des Weiteren generalisierte Methoden des Informationstransfers zum Einsatz kommen. Bei der Implementation des SlideValueObjects wird also nicht nur ein weiteres Dialogobjekt abgebildet, sondern diese neuen Informationstransfermethoden werden erstmalig zum Einsatz kommen.

5.1.3 Optimierte Informationstransfermethoden

Im Zuge der Schaffung neuer Methoden zum Transport von Fließkommazahlen fiel auf, dass die Methoden sowohl zur Behandlung von zu schickenden und zum Umgang mit zu empfangenen Informationen dienlich sind. Dies bezieht sich in erster Linie auf unstrukturierte Methoden zum Lesen und serverseitigen Schreiben von Information. Im Zuge der im Rahmen dieser Arbeit gemachten Änderungen werden auch die Methoden zur Kommunikation server- und clientseitig vereinheitlicht.

Die Vereinheitlichung erfolgt in der Art, dass numerische Felder, wie sie am Server gehalten werden, auch am Client abgebildet werden. Momentan besteht ein numerisches Feld in XploRe aus einem achtdimensionalen Feld, welches je nach Bedarf gefüllt wird. Das heisst, dass ein dreidimensionales Feld auch achtdimensional gehalten wird. Dieses Vorgehen minimiert die Zahl der Methoden dafür, die gehaltene Information abzufragen. Genau dieses eben beschriebene Verfahren wird auch in MD*Crypt abgebildet. Für Textfelder gilt generell dasselbe. Auch hier werden die Daten in derzeit achtdimensionalen Feldern gehalten. Auch ist es wünschenswert, auf Client- und Serverseite gleiche Formen der Informationshaltung zu implementieren.

Diese XploRe-Felder sind generell einsetzbar. Sogar einzelne numerische Werte sowie Textwerte lassen sich deshalb übertragen. Hiervon ausgehend, sind dann nur noch Methoden zum Datentransport für genau diese beiden Typen von Datenfeldern zu implementieren. Die Optimierung besteht hier darin, zukünftigen MD*Crypt-Administratoren das Ändern oder Verbessern zu erleichtern.

5.1.4 Drittprogrammierer

MD*Crypt ist ein Packet, das nicht selbst lauffähig ist. Vielmehr bietet es lediglich Methoden, um zukünftigen Programmierern die Kommunikation mit einem XplRe-Quantlet-Server zu ermöglichen. Hieraus entsteht die Notwendigkeit, besonders gewissenhafte Dokumentationen bereitzustellen. Weiter ist es von höchster Wichtigkeit, die Methoden leicht verständlich zu gestalten. Es darf keinem Programmierer zugemutet werden, sich erst in hochkomplexe Zusammenhänge einzuarbeiten. Um hier Verbesserungen zu erzielen, müssen Methoden grundsätzlich so gestaltet sein, dass sie quasi fertig zur Benutzung sind.

Es ist hierbei besonders darauf zu achten, dass die Information auch hochkomplexer Objekte leicht und unmissverständlich abfragbar ist. Ein XploRe-Grafikkontext hält zum Beispiel hochkomplexe Information. Dazu kommt, dass die Information teils kodiert vorliegt. Diese Kodierung ist ein Verfahren, welches das Schicken von Information auf Bytestreams erst ermöglicht. Es wird darauf zu achten sein, dass solche Information auch für Dritte nutzbar vorliegt. Dies kann durch verbesserte Dokumentation der vorliegenden Ergebnisobjekte und durch die Schaffung verbesserter Abfragemethoden geschehen.

5.2 Bestandsaufnahme, Möglichkeiten

5.2.1 Datentypen

MD*Crypt bietet bisher nicht die Möglichkeit, numerische Felder als Objekte zu halten und weiterzubehandeln. In bisherigen Einsatzfeldern war dies nicht nötig. Numerische Felder wurden nur innerhalb von Grafikkontexten vom Server zum Klient übertragen. Es wird immer dringender, numerische Felder auch abgekoppelt von Grafikkontexten zu halten und - noch wichtiger - diese auch zum Server übertragen zu können. Dieses numerische Datenfeld wird sich in den Rahmen der XQS-Objekte einbinden müssen.

XQS-Objekte sind bisher Ergebnisobjekte, die der Server an den Client schickt. Objekte, die der Client an den Server schickt, bestehen jedoch im Moment aus einfachen Zeichenketten, die die XploRekommandos halten. Damit ist eine Datenübertragung vom Client zum Server nur unzureichend zu realisieren. Der Begriff XQS-Objekt sollte erweitert werden auf alle Objekte, die an Client und Server gehalten werden, um das gewünschte Resultat zu erzielen. Der Client muss in die Lage versetzt werden, alle am Server verwendeten Datenstrukturen zu halten und zu transportieren.

5.3 Ziele der Implementation

Das Ziel der Arbeiten muss es sein, die Datenstrukturen, die der XQS verwendet, auch clientseitig abzubilden. Sowohl numerische Felder als auch Textfelder werden in XploRe als jeweils achtdimensionale Matrizen gehalten. Es wird also jeweils eine Methode zum Lesen und Schreiben von sowohl Text- als auch numerischer Matrix implementiert werden.

Weiter müssen diese Abfragemethoden derart aufbereitet werden, dass es Drittprogrammierern leicht möglich ist, die Information abzufragen, die XploRes komplexe Strukturen halten. Dies gilt sowohl für die neu zu schaffenden Objekte als auch für schon bestehende wie das Grafikobjekt.

5.4 Methoden der Implementation

Um XploRes-Feldstruktur abzubilden, werden zuerst Objekte geschaffen, deren erste Eigenschaft ein achtdimensionales String- beziehungsweise Doublearray ist. Als weitere Eigenschaft halten sie einen Integerwert, der die maximale Dimensionalität (hier: acht) hält. Eine weitere Eigenschaft ist ein eindimensionales Integerfeld, das die eigentlichen Feldlängen pro Dimension hält. Für eine zweidimensionale 2x2 Matrix wären also die ersten beiden Feldwerte zwei und die verbleibenden sechs jeweils eins. Als weiteres Attribut erhält dieses Objekt einen Namen als Zeichenkette. Mit diesen Attributen ist ein solches numerisches oder Textfeld ausreichend beschrieben.

Es bleiben nun die Fragen: Wie sollte die Information dieser Felder abfragbar sein? Soll man, um ein einzelnes Feld abzufragen oder zu setzen, immer alle acht Dimensionen angeben müssen? Für Drittprogrammierer wäre dieses Vorgehen nur schwer verständlich. Es müssen vielmehr abgeleitete Objekte geschaffen werden, die es ermöglichen, den Kern, das achtdimensionale Feld, wie ein ein- oder zweidimensionales Feld zu behandeln. In fast allen Szenarien werden ein- und zweidimensionale Felder völlig ausreichen, um die zu haltende Information aufzunehmen. Einstweilen kann - beispielsweise - der XQC-Textoutput höchstens zweidimensional anzeigen. Diese abgeleiteten Objekte bieten nun Abfrage- und Setzmethoden, die dem Programmierer die Möglichkeit geben, XploRes-Felder wie ein- oder zweidimensionale Felder zu behandeln (betrachte Grafik (8) als Ergebnis dieser Implementation). Es wurde das numerische Feld XQCDoubleArray als Beispiel für die abzuleitenden Klassen XQCDoubleMatrix und XQCDoubleVector gewählt.

Bei der Implementation der eigentlichen Methoden zum Transport der Fließkommazahlen zum Server entstanden unter Windows-Plattformen Probleme. Offen-

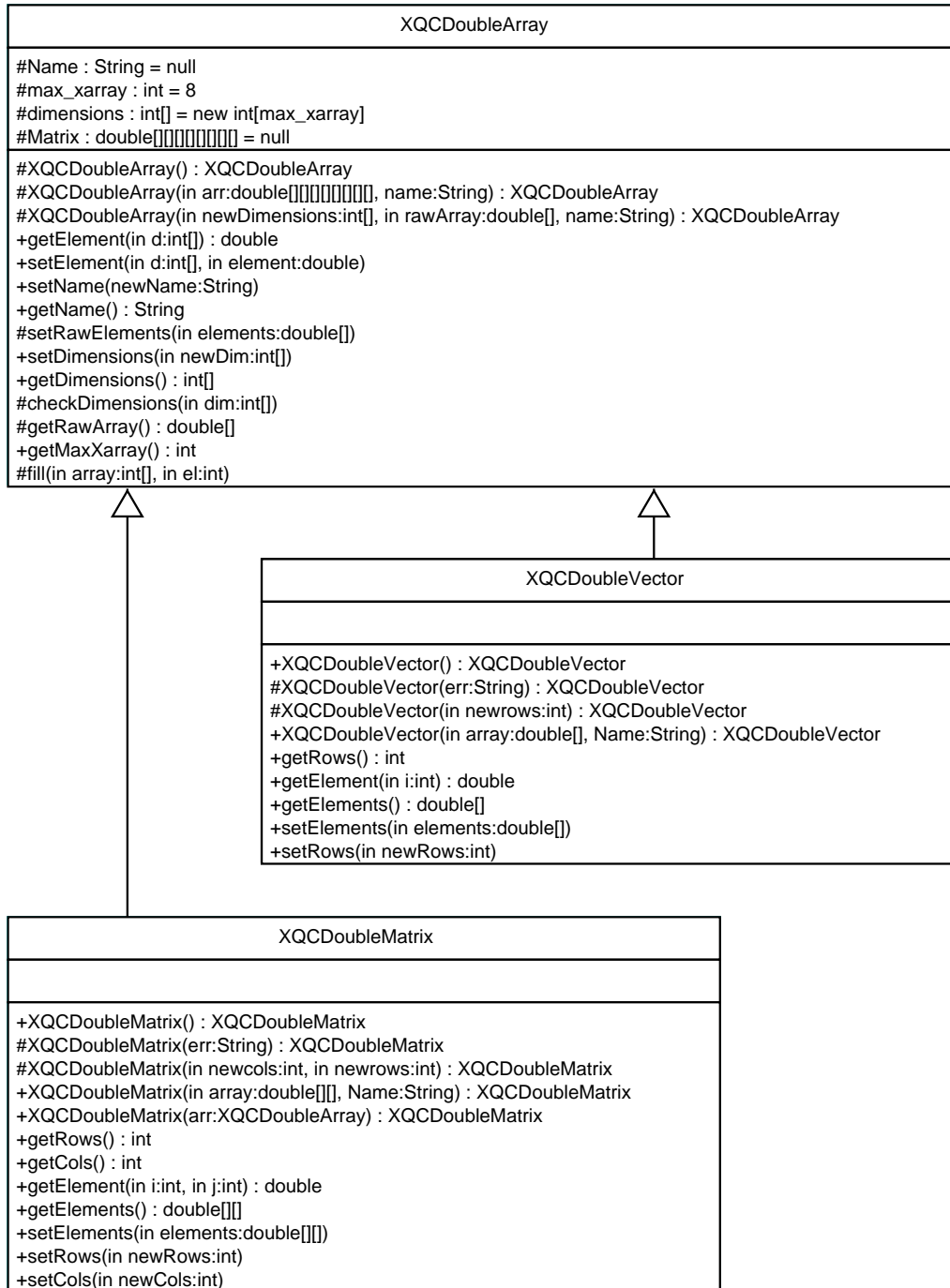


Abbildung 8: XQCDoubleArray-Klasse und ihre Ableitungen

sichtlich erfolgt auf dem Weg von Middleware zu XQS eine Interpretation der gesendeten Bytes (vergleiche Grafik (2)). Einem Byte, das eigentlich Teil eines vier Byte großen Datentyps ist, wird auf dem unterliegenden Standard Input-Output-Stream seine Bedeutung im Kontext eines Characterstreams zugewiesen. Das Byte 23 hieße hier EndOfFile (EOF), hat aber in dem von MD*Crypt verwendeten Kontext eine gänzlich andere Bedeutung. Ein Fließkommawert, der als eines seiner Bytes eine 23 hält, lässt den XQS also abstürzen, weil die lesende Methode unter Microsoft's Windows diesen Bytewert 23 als EOF interpretiert. Es musste also ein Algorithmus entworfen werden, der dafür sorgt, dass keines der Bytes mit besonderer Bedeutung unmaskiert von MD*Serv zum Server geschickt werden.

Dies wurde durch eine Maskierung nach folgendem Muster erreicht: Repräsentiere jedes Problembyte als Sequenz aus zwei Bytes. Das erste sei das Führungsbyte, welches der demaskierenden Methode anzeigt, dass hier ein Problembyte maskiert wurde; im MD*Serv/XQS-Fall wurde das Byte 200 gewählt. Als zweites Byte wird das Führungsbyte plus das zu maskierende Byte geschickt. Für das Byte 23 wird also nun die Sequenz 200 223 geschickt. Das Byte 200 selbst muss nun aber auch maskiert werden, denn es übernimmt ja die Rolle des Führungsbytes. Als 200 wird eine 200 200 geschickt. Auf diese Weise wird keines der Problembytes geschickt, und es bleibt dem Gegenüber möglich, den korrekten Zustand wieder herzustellen.

Der Maskierungsalgorithmus wird in der write(byte[]) Methode des XPLInputStreams und des XPLOutputStreams implementiert. Dieser kapselt einen von Java bereitgestellten Input- beziehungsweise OutputStream und kapselt dessen write(byte[]) Methode derart, dass erst der Maskierungsalgorithmus gerufen und dann die unterliegende write Methode verwendet wird. Zum Verständnis betrachte das folgende Quelltextfragment; es zeigt, wie die maskierende Methode die eigentliche Schreibmethode kapselt.

```
...
import java.io.DataOutputStream;
import java.io.OutputStream;

public class XplDataOutputStream {

private DataOutputStream dout;

...
...

```

```
public void write(byte[] bytes, int off, int len) throws java.io.IOException{

    byte[] newBytes = null;

    /*
    Maskierungsalgorithmus
    legt ein neues Bytefeld an (newBytes) und füllt es mit
    den maskierten Werten aus dem übergebenen Feld (bytes)
    */

    // Aufruf der unterliegenden Writemethode,
    //Beachte, dass die Syntax beider Methoden die gleiche ist

    dout.write(newBytes, off, newBytes.length);
    dout.flush();
    return;
}
...
}
```

5.5 Ergebnis der Implementation

Im Rahmen dieser Arbeit wurden Objekte geschaffen, die die Informationshaltung in XploRe auch für den Client verfügbar machen. Diese wurden dahingehend erweitert, dass sie möglichst leicht auch von Drittprogrammierern zu benutzen sind.

Durch die Vereinheitlichung der Ein- und Ausgabemethoden gelingt es, fast alle zu transportierenden Datenstrukturen mit einer minimalen Anzahl von Methoden zu bewältigen. Die Verbesserung liegt auf der Hand. Mit diesen wenigen Methoden werden sich in Zukunft neue Objekte schneller implementieren lassen. Dies wird möglich, weil sich basierend auf den XploRe-Datenstrukturen, Algorithmen zum Transport von XploReobjekten leichter definieren lassen.

Diese Änderungen ermöglichen eine Erweiterung der Einsatzgebiete von MD*Crypt. Die bisherigen Einsatzszenarien wie die Abbildung von XploRes Grafikkontexten innerhalb eines Applets etwa in den elektronischen Büchern, die auf XploRe-Technologie basieren, können erweitert werden. Es sind nun auch Szenarien denkbar, in denen größere numerische Datenmengen ohne Genauigkeits-

verlust zum Server transportiert werden müssen. MD*Crypt wird somit zu einem Packet, das die gebotenen XploRe-Funktionalitäten, die bisher nur dem Nutzer von Stalaloneversionen zur Verfügung standen, umfangreicher ausschöpfen kann.

6 Protokolloptimierung

6.1 Geschwindigkeit

Das MD*Crypt-Protokoll selbst bietet Möglichkeiten der Geschwindigkeitsoptimierung. Diese sollen im Folgenden identifiziert werden; weiterhin sollen Wege zur Implementation verbesserter Algorithmen aufgezeigt werden.

Im Folgenden soll das MD*Crypt-Protokoll auf Möglichkeiten untersucht werden, sowohl Geschwindigkeit als auch Funktionalität der Kommunikation zu verbessern. Bisher wurde in fast allen Fällen eher die Möglichkeit diskutiert, die bestehende Kommunikation effizienter zu gestalten. Einige Möglichkeiten der Optimierung liegen aber in der Optimierung der Kommunikationsspezifikationen selbst. Es wird in erster Linie um die Optimierung der Geschwindigkeit gehen.

Bei der Analyse des MD*Crypt-Protokolls fällt auf, dass speziell bei der Spezifikation von Grafikkontexten redundante Information übertragen wird. Die grafische Repräsentation jedes einzelnen Datenpunktes wird übertragen, auch wenn mehrere Punkte die gleiche Repräsentation haben. Es mussten Wege gefunden werden, um die Übertragung dieser redundanten Information zu verhindern oder sie zumindest zu minimieren.

6.2 Komprimierungsalgorithmus

Zu jeder Datenmatrix, die in einem Grafikkontext übertragen wird, wird ein Feld von Integerwerten übertragen, das seine grafische Repräsentation hält. Dieses Feld hält bisher genau so viele Grafikrepräsentationen, wie Datenpunkte anzuzeigen sind. Viele dieser Punkte enthalten die gleiche Repräsentation, einfach weil viele Punkte eines Grafikkontexts die gleiche grafische Repräsentation haben. Es muss also ein Weg gefunden werden, bei dem für gleichartig repräsentierte Punkte nur ein Wert übertragen wird.

Ein Weg der Laufzeitkomprimierung folgt folgendem Algorithmus: Man betrachte die Kodierung für die grafische Repräsentation eines Datenpunktes in einem Grafikkontext. Derzeit wird ein Integerwert übertragen. Dieser enthält die Kodierung innerhalb seiner Vierbyte-Repräsentation. Die oberen drei Bytes enthalten jeweils die Rot-, Grün- und Blaufarbwerte des Datenpunktes. Das untere Byte enthält Werte für Layout und Größe des Datenpunktes. Das untere Byte kann auf Grund der technischen Spezifikation des Layout- und Größenwertes nie den Wert 255 annehmen. Dadurch entsteht die Möglichkeit, auf diesem Layout- und Größenwert weitere Information zu transportieren. Für die Grafikrepräsentatio-

nen innerhalb des MD*Cryptprotokolls wird dieses Byte benutzt, um die Information über die Häufigkeit der Wiederholung einer grafischen Repräsentation für eine Menge von Datenpunkten zu übermitteln. Zuerst wird hierzu das Layout- und Größenbyte zum Oberen der Vierbyterepräsentation gemacht. Liest man nun einen Repräsentationspunkt, prüft man zuerst, ob das obere Byte den Wert 255 hat. Hat es diesen Wert nicht, verfährt man wie gehabt. Das obere Byte wird dem entsprechenden Datenpunkt als Layout- und Größeninformation zugewiesen. Die drei unteren Bytes werden dem korrespondierenden Datenpunkt als Farbinformation zugewiesen. Hat das obere Byte aber den Wert 255, interpretiert man die unteren beiden Bytes als Integerwert, der die Anzahl der folgenden Datenpunkte mit der gleichen Repräsentation angibt wie der zuletzt gelesene. Auf diese Weise erhält man die Information über die grafische Repräsentation jedes einzelnen Datenpunktes, ohne sie für jeden Punkt übertragen zu müssen.

Als Beispiel soll die Übertragung von 10000 zweidimensionalen Datenpunkten gleicher grafischer Repräsentation dienen. Zuerst sind die 10000×2 Doublewerte der Datenpunkte zu übertragen. Als nächstes ist die grafische Repräsentation zu übertragen. Bisher war das ein Integerfeld der Länge 10000. Mit Laufzeitkomprimierung bleibt noch ein Integerfeld der Länge 2 zu übertragen. Das erste hält die grafische Repräsentation des ersten Datenpunktes und das zweite die Information darüber, dass die nächsten 9999 Datenpunkte die gleiche Repräsentation haben.

Man erkaufte die geringere Menge an zu übertragender Information aber mit einem erhöhten Rechenaufwand sowohl auf Server- als auch auf Clientseite. Serverseitig muss ermittelt werden, wie oft sich eine grafische Repräsentation wiederholt, und auf Clientseite muss wieder ein Feld errechnet werden, das gleiche Länge wie das Datenfeld hat und alle korrespondierenden grafischen Werte korrekt hält.

Betrachte auch folgende Quelltextfragmente:

```
...
if(corgData != null){
    for(int k=0 ; k<numberOfRows; k++){
        PointColors[k] = new Color( (int)(0xfffff00 & corgData[k] >> 8);
        PointLooks[k] = (int) (0x00000f0 & corgData[k] >> 4);
        PointSizes[k] = (int) (0x0000000f & corgData[k]);
    }
}
...
```

Im obigen Quelltextfragment ist zu erkennen, dass `corgData`, das Feld, das die grafischen Repräsentationen hält, genau so lang ist wie das Datenfeld (hier durch die Eigenschaft `numberOfRows` beschrieben). Die Dekodierung der grafischen

Eigenschaften erfolgt innerhalb einer *for*-Schleife. Im nachfolgenden Fragment ist der Algorithmus zur Dekomprimierung der komprimierten Grafikrepräsentationen beschrieben. Man beachte, dass die *for*-Schleife nur über die Länge des Feldes der grafischen Repräsentationen läuft. Für jedes Element des Feldes *datastyles* wird zuerst abgeprüft, ob das obere Byte 255 ist, um dann entsprechend die Felder der grafischen Repräsentationen zu füllen.

```
int wdh, swdh;
```

```
...
```

```
if(this.datastyles != null){
    for(k=0 ; k<datastyles.length; k++){
        a =(byte)((0xff000000 & datastyles[k])>>24) ;
        if(a == (byte)255){
            wdh = (int)(0x0000ffff & datastyles[k]);
            Color tmpc = new Color( (int)(0x00ffffff & datastyles[k-1]));
            Arrays.fill(PointColors, k+swdh, k+swdh+wdh-1, tmpc);
            Arrays.fill(PointLooks, k+swdh, k+swdh+wdh-1,
                (int)(0xf0000000 & datastyles[k-1]) >> 28);
            Arrays.fill(PointSizes, k+swdh, k+swdh+wdh-1,
                (int) (0x0f000000 & datastyles[k-1]) >> 24);
            swdh += wdh-1;
        }
        else{
            PointColors[k+swdh] = new Color( (int)(0x00ffffff & datastyles[k]));
            PointLooks[k+swdh] = (int) (0xf0000000 & datastyles[k]) >> 28;
            PointSizes[k+swdh] = (int) (0x0f000000 & datastyles[k]) >> 24;}
        }
    }
}
...
```

Um die Ergebnisse der Änderungen zu testen, wurden die MD*Cryptklassen so verwendet, wie sie im Verlauf dieser Arbeit entstanden. Das heißt: Das Lesen und Schreiben ist weitgehend optimiert. Die Ergebnisse wurden auf einer Maschine mit AMD Athlon 1.2GHz und Linux2.4 Kernel getestet. Der XploRe-Quantlet-Server läuft auf derselben Maschine. Es entstehen also keine technischen Engpässe durch Verbindungen zu entfernten Maschinen.

Des Weiteren ist zu bemerken, dass ein direkter Vergleich der geänderten mit den herkömmlichen Methoden schon deshalb mit Vorsicht zu betrachten ist, weil auch die Kommunikationsklassen an sich Änderungen unterworfen waren. Ein Feld von

Doublewerten wird bei den Klassen mit Laufzeitkomprimierung nicht auf gleiche Weise übertragen wie noch vor der Laufzeitkomprimierung (siehe hierzu das Kapitel Funktionalitätserweiterung). Zeitnahmen können also keine echten Vergleichswerte liefern. Weiter ist anzumerken, dass unter den zugegeben sehr günstigen Testbedingungen das Lesen eines Bytefeldes der Längen 10000 im Schnitt nicht länger als zwei Millisekunden dauert. Es konnten für die Testumgebung schließlich keine signifikanten Geschwindigkeitsverbesserungen gemessen werden. Nichtsdestoweniger werden in Netzwerken die Durchsatzgeschwindigkeiten nicht immer derart günstig sein. Man sollte also auf keinen Fall auf die beschriebene Form der Komprimierung verzichten.

7 Ergebnisanalyse

Die Änderungen und Neuimplementationen, die während dieser Arbeit entstanden, versuchen in erster Linie, Verbesserungen des MD*Crypt-Protokolls auf den Gebieten Geschwindigkeit und Funktionalität zu erreichen. Nicht unerhebliche Geschwindigkeitserhöhungen konnten im Bereich der Ein- und Ausgabe von MD*Crypt erreicht werden. Weiterhin konnten sowohl neue Objekte zu Datenhaltung als auch neue Ergebnisobjekte implementiert werden, welche die möglichen Einsatzgebiete MD*Crypts ausweiten können. Es kann allerdings auch festgehalten werden, dass weitere Anstrengungen nötig sein werden, um ein Maximum an Nutzbarkeit aus der gegebenen Technologie zu ziehen. Dieses Kapitel versucht, die Ergebnisse zusammenzufassen und verbliebene Handlungsfelder aufzuzeigen. Hierzu wird zuerst eine kurze Bestandsaufnahme der Ergebnisse in den Bereichen Funktionalität und Geschwindigkeitsverbesserung angestellt, um dann an Hand der Referenzprojekte die neu entstandenen Möglichkeiten aufzuzeigen.

7.1 Neue Funktionalitäten

Mit dieser Arbeit werden als erstes die Objekte XQCDoubleArray und XQCTextArray sowie ihre Kindklassen eingeführt. Diese dienen dazu, auf MD*Cryptseite eine Datenhaltung wie auf Seiten des XploRe-Quantlet-Servers zu ermöglichen. Mit der Schaffung dieser Grundvoraussetzung können nun auch clientseitig Daten gehalten werden, die die XploRefunktionalitäten auch auf Seiten des Clients zur Verfügung stellen.

Mit der Implementierung des XQSAddDataObjects wird ein weiteres Ergebnisobjekt zur Verfügung gestellt, welches es ermöglicht, komplexere Grafikkontexte auch clientseitig zu verwalten. Das SlideValueObject als Ergebnis dieser Implementation ermöglicht es, weitere interaktive Elemente in XploRe Quantlets zu verwenden. Speziell in Einsatzgebieten der Lehre statistischer Methoden kann dies eine Verbesserung darstellen.

Insgesamt versetzen die genannten Ergebnisse MD*Crypt in die Lage, in erheblich mehr Einsatzszenarien als bisher zu agieren.

7.2 Geschwindigkeit

Die Ein- und Ausgaberroutinen MD*Crypts konnten während dieser Arbeit erheblich verbessert werden. Auf Plattformen wie Microsoft's Windows und unter Linuxsystemen wurden erhebliche Geschwindigkeitsgewinne erzielt. Dies schafft

die Möglichkeit, auch Kontexte mit hohem Datentransferbedarf zu verarbeiten. Es wird möglich, Quantlets zu bearbeiten, die früher schon ihrer reinen Datentransportzeit wegen nicht sinnvoll zu bearbeiten waren.

Diese Verbesserung wirkt sich auf sämtliche bestehenden und zukünftigen Clients aus. Die Menge der zu transportierenden Daten ist kein Hinderungsgrund für die Bearbeitung von statistischen Problemen in XploRe Quantlet Server/MD*Crypt-Architektur mehr.

7.3 Fehlertoleranz

Die clientseitig verfügbare Information über den Zustand der Kommunikation zwischen XploRe-Quantlet-Server und MD*Crypt wurde verbessert. Sowohl die zur Verfügung stehende Information über die beteiligten Programme, deren Versionen und unterliegende Betriebssysteme als auch die Information über Ausnahmen im Programmablauf, soweit sie nicht innerhalb von MD*Crypt behandelt werden können, werden nun generell an den Client durchgereicht. So ist dem Clientprogrammierer die Möglichkeit gegeben, alle verfügbare Information zu nutzen, um einen fehlertoleranteren Ablauf seines XQSListeners zu gewährleisten.

7.4 ReX

Für den Referenzclienten ReX ist vor allem die Möglichkeit, die gehaltenen Daten wirklich als Fließkommawert zu senden und zu empfangen, von größter Bedeutung. Es wird für ihn nun möglich, die in Excel gehaltenen Daten präziser und mit weniger Rechenaufwand durch Datentypumwandlung an den XploRe Quantlet-Client zu senden.

Die Geschwindigkeitsverbesserungen wirken sich selbstverständlich auch positiv auf die Anwenderfreundlichkeit des ReX-Clients aus. Mit den neuen Funktionalitäten kann ReX zu einem anwenderfreundlichen, hochperformanten Programm werden, um XploRe-Funktionalitäten innerhalb Excels zu nutzen.

7.5 XQC und nahe Projekte

Für den derzeit höchstentwickelten aller Clientimplementationen ergeben sich verschieden neue Möglichkeiten. Die XploRe-ähnliche Datenhaltung und die Geschwindigkeitserhöhung werden sich sehr positiv auf die Verwendbarkeit des XQC auswirken. Weiter wird es ihm mehr noch als dem ReX-Client möglich sein,

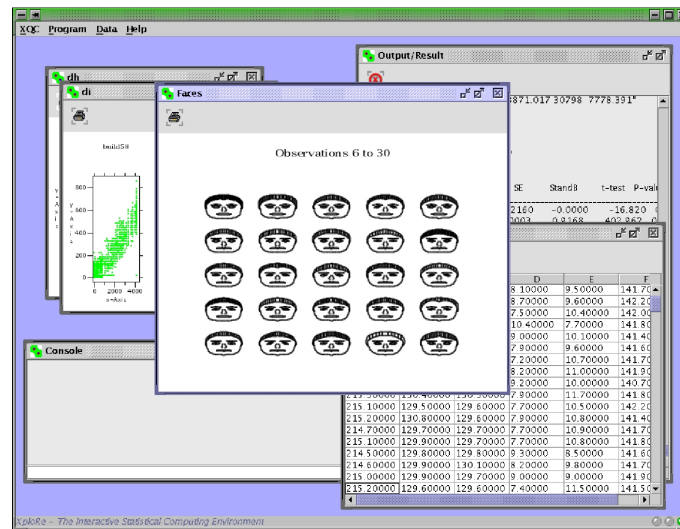


Abbildung 9: Auch grafische Kontexte mit hohem Datentransfer werden möglich

die Verbesserungen im Bereich der Fehlertoleranz zu nutzen, da ihm sämtliche Information über den Zustand der Kommunikation zur Laufzeit zur Verfügung gestellt wird. Schnelleres und präziseres Reagieren auf etwa auftretende Kommunikationsprobleme werden ihm - nach einiger Implementationsarbeit - zur Verfügung stehen. Schon jetzt ist es möglich, mit Hilfe der Geschwindigkeitszuwächse im Ein- und Ausgabebereich Aufgaben zu lösen, die vorher schlicht an der Bearbeitungszeit scheiterten.

Eng verbunden mit dem XQC sind Projekte, die XQC als Kerntechnologie verwenden. Hier sind zunächst die MD*Book e-books zu nennen. Wenn XQC die neuen Funktionalitäten implementiert, wird es möglich sein, weit komplexere Beispiele mit höherer Interaktivität in die Bücher einzubinden. Auch Projekte, die MD*Crypt heute schon verwenden, werden von sowohl dem Laufzeitverhalten als auch den neuen Funktionalitäten profitieren. Zu nennen sind GraFitI und das ambitionierte E-Stat Projekt, in dem versucht wird eine statistische Wissensbasis aufzubauen (siehe auch hier 'Ressourcen im World Wide Web'). Auch hier kann MD*Crypt nun mehr leisten.

Bleibt abschließend festzuhalten, dass die Änderungen quasi unter 'Reinraumbedingungen' erfolgten. Wie Drittprogrammierer die Änderungen annehmen wird die Veröffentlichung der neuen MD*Crypt-Version zeigen. Erfahrungswerten zeigen, dass oft noch Anpassungsbedarf besteht. Einfach weil Drittprogrammierer mehr sehen und Anderes versuchen, als der Paketentwickler vorhersehen kann.

8 Ausblick

Dieser Abschnitt soll klären, wo zukünftig Handlungsfelder zur Weiterentwicklung der XQS-/MD*Crypt-Architektur zu suchen sein werden. Sowohl strategische als auch operative Optionen sollen skizziert werden. Hierbei werden auch Grenzen der XQS-/MD*Crypt-Architektur erkannt werden können.

8.1 MD*Crypt

MD*Crypt ist bereits ein weit entwickeltes Programmpaket, welches weite Teile der XploRe-Funktionalität ansteuern kann.

Mit MD*Crypt ist man heute in der Lage, weite Teile der XploRe-Objekte abzubilden und zu behandeln. Es bietet die Möglichkeit, Clients zu entwickeln, die von allgemeinen Anwendungen wie dem Abbilden einer XploRe-Nutzerschnittstelle (die die XploRe-Standalone-Nutzerschnittstelle nachbildet) bis hin zu speziellen Anwendungen (in denen nur sehr begrenzte XploRe-Funktionalitäten für sehr spezielle Zwecke benutzt werden müssen) angewandt werden können. Mit MD*Crypt stehen die Funktionalitäten von XploRe an jedem Computer mit Netzanschluss zur Verfügung.

Mit der Entwicklung der XploRe-Sprache - wie jede andere nichttote Sprache entwickelt sich auch diese - wird es immer nötig sein, neue oder verbesserte Objekte aus XploRe den Clientprogrammierern zur Verfügung zu stellen. Hier wird weiter und ständig Handlungsbedarf bestehen. Auch Fragen der Performanz wird in Zukunft gleich bei der Erstimplementierung mehr Beachtung geschenkt werden können.

Mit den Veröffentlichungen der ersten Java1.4 Beta-Versionen stellt sich wiederum die Frage, ob und wie man frühe Implementierungen der Java-Virtual-Machine, hier namentlich der Microsoft Java-Virtual-Machine, unterstützt. Man verzichtet dabei auf weite Funktionalitäten aktueller Javamaschinen. Das Verwenden unabhängiger Teilprogramme (Threads) ist beispielsweise eingeschränkt; noch problematischer ist die Tatsache, dass andere Pakete, welche gut ausimplementierte Methoden für verschiedene Funktionalitäten bereitstellen würden, von MD*Crypt nicht importiert werden können. 'Das Fahrrad ein zweites Mal zu erfinden', und dies womöglich schlechter, ist erstens arbeitsintensiv und zweitens fehlerträchtiger, weil alle bereits gefunden Fehler in häufig benutzten Paketen mit hoher Wahrscheinlichkeit in der Nachimplementierung wieder gemacht werden. Speziell im Fall der ReX-Clients bleibt die Frage offen, ob es andere Wege gibt, eine COM DLL zu generieren, als gerade die Nutzung der Microsoft-Java-Virtual-Machine.

Eine Entwicklungslinie MD*Crypts wird seine Verwendung serverseitig werden. Hierzu muss die spezielle Art der Datenübertragung in diesen Umgebungen beachtet werden. JavaServerPages können zwar die Methoden von Javaklassen aufrufen, aber umgekehrt kann eine Javaklasse nur sehr eingeschränkt Methoden an JavaServerPages aufrufen. Es wird nötig werden, Objekte anzubieten, die Ergebnisobjekte vorhalten und so für JavaServerPages abrufbar machen.

8.2 Interaktive Grafik

Um den folgenden Abschnitt leicht zu verstehen, betrachte man folgendes Beispiel: Ein Grafikkontext an einem XploRe-Quantlet-Client existiere. Er zeige eine Wolke von Datenpunkten und eine Regressionslinie durch diese hindurch. Man fasse mit einem Eingabegerät einen der Datenpunkte und bewege ihn. Die korrespondierende Regression wird zur Laufzeit neu berechnet und angezeigt. Dies ist für größere Datenmengen mit den heute zur Verfügung stehenden Datenobjekten kaum zu realisieren, da man dazu den kompletten, geänderten Datensatz an den Server schicken muss. Dieser muss die Regression neu berechnen und dann den geänderten Datensatz und die geänderte Regression komplett zurückschicken. Die zu transportierende Datenmenge ist zu hoch; es dauert zu lang.

Welche Information muss dem Server eigentlich geschickt werden? Die Information über genau einen geänderten Punkt in einem Grafikkontext. Hätte man ein Objekt zur Verfügung, das die Information eines Datenpunktes und die Information über seine Zugehörigkeit zu einem bestimmten Grafikkontext überträgt, wäre es möglich, Änderungsinformation redundanzfrei zu übertragen. Es müssen dann nur noch die Information über den geänderten Punkt an den Server gesandt werden; dieser kann dann die Berechnungen ausführen und lediglich die geänderte Regression zurücksenden. Auch für große Datensätze können so Grafikkontexte interaktiver werden.

Um in Szenarien der Lehre hohe Interaktivitätsgrade in der Verwendung grafischer Kontexte zu erreichen, müssen also neue Objekte der Datenhaltung definiert werden, um auch Fragmente von Grafikkontexten übertragen zu können. Zusammen mit dem neu implementierten XQSAddDataObject kann es dann möglich sein, solche hochinteraktiven Grafikkontexte auch in Client/Server-Architekturen umzusetzen. In der Lehre wäre damit eine der Hauptforderungen auch für Client-/Serversysteme erreicht.

8.3 MD*Serv

MD*Serv hat bisher die Aufgabe, auf Anfrage einen XQS zu starten und die Kommunikation zwischen MD*Crypt und XploReQuantletServer sicherzustellen. Grundsätzlich hat MD*Serv aber eine zentrale Position zwischen XQS und MD*Crypt. Folgendes Szenario möge zur Veranschaulichung dienen. Sei der XQS-Listener ein Applet, welches im Browser läuft. Es hat dann aus Sicherheitsgründen nur die Möglichkeit, XploRe-Quantlet-Server auf der Maschine zu kontaktieren, auf der auch die appletrufende HTML-Seite liegt. Kann dieser Server aber genügend Ressourcen oder die angefragten Methoden zur Verfügung stellen? Wenn MD*Serv in die Lage versetzt wird, Information über benachbarte XQS zu halten, kann es wie ein Ressourcendispatcher wirken. Es ist grundsätzlich in der Lage, andere MD*Serv zu kontaktieren und so Informationen mit diesen über den Zustand ihrer jeweiligen Server zu erhalten. Weiter wird es möglich sein, Methoden anderer XQS auf diesen Weg anzusprechen.

Um solche Funktionalitäten zu implementieren, müssen die Methoden der MD*Serv erheblich erweitert werden. Methoden zum MD*Serv/MD*Serv-Kontakt werden implementiert werden müssen.

Diese Weiterentwicklung der Middleware ist einer der meistversprechenden Entwicklungspfade der XQS/MD*Crypt-Architektur.

8.4 Drittprogrammierer

Da MD*Crypt nur in der Lage ist, XploRe-Funktionalitäten zur Verfügung zu stellen, wenn Dritte die Schnittstelle zu ihrer Plattform und zu ihren Einsatzszenarien implementieren, wird der Spezifikation und Dokumentation des MD*Crypt-Protokolls verstärkte Aufmerksamkeit gewidmet werden müssen.

Bisher bestand die Dokumentation aus einer javadoc-generierten Sammlung von HTML-Seiten. Diese sind für Javaprogrammierer lesbar, bieten aber keine ausreichende Information über Möglichkeiten und Grenzen der MD*Crypt-Technologie. Zudem sind derzeit schon nicht alle MD*Crypt-Nutzer in Javaumgebungen genügend geschult; bei der steigenden Komplexität von Javaumgebungen kann nicht davon ausgegangen werden, dass sich dies in Zukunft ändert. Auch den Drittprogrammieren, die weniger firm in Java sind, muss die Möglichkeit gegeben werden, MD*Crypt leicht zu verwenden.

Wie kann dies erreicht werden? Für Neunutzer von MD*Crypt ist der MD*Crypt-Webdienst erste Informationsquelle. Ressourcen und Dokumentationen sowie der Download des Packetes werden hier angeboten. Alle Erweiterungen der Doku-

mentation müssen hier abgelegt werden. Welche Erweiterungen sind wünschenswert? Zuerst wäre es nützlich, einen simplen Client, der beispielhaft die Hauptfunktionalitäten MD*Crypts implementiert, als Javaquelltext abzulegen. Zudem muss dafür gesorgt werden, dass neben der Java-Dokumentation Informationen auf einen tiefer liegenden Level bereitstehen. Hier kann auf beispielhafte Projekte verwiesen werden, wie das MD*Book, das MM*Stat oder eben den XQC selbst, welche zeigen, wie MD*Crypt eingesetzt werden kann. Für die eigentliche Entwicklerarbeit wird es zunehmend wichtig werden, weitere Informationen bereitzustellen wie etwa eine erweiterte Dokumentation der eigentlichen Kommunikation zwischen MD*Crypt und dem XQS. Hier müssen Fragen behandelt werden, wie das Verhalten von MD*Crypt bei bestimmten Ausnahmen reagiert, die während des Informationsaustauschs auftreten können, welche Fehlerbezeichnungen mit welchen Fehlern korrespondieren etc.

Mit steigenden Nutzerzahlen wird der Support via E-mail wichtiger werden. Je mehr Programmierer MD*Crypt benutzen, desto mehr Einsatzszenarien werden entstehen. Nicht alle werden von der gegebenen XQS-/MD*Crypt-Technologie fehlerfrei behandelt werden können. Um schnelle und unkomplizierte technische Unterstützung bieten zu können, werden personelle Ressourcen für die Pflege der Nutzerbeziehungen bereitstehen müssen.

Die Pflege der Beziehungen zu den Cliententwicklern wird zunehmend wichtiger für den Erfolg eines Projektes wie MD*Crypt sein.

8.5 Sicherheit

Implementationen von Serveranwendungen werden immer hohe Anforderungen an die Sicherheit stellen. Immerhin bietet MD*Crypt Methoden an, um entfernte Server zu kontaktieren und dort Prozesse laufen zu lassen. Dies ist per se ein Sicherheitsrisiko. So kann schon eine Endlosschleife, als Quantlet implementiert, zur Schadroutine werden, wenn sie serverseitig so viele Ressourcen beansprucht, dass andere Dienste der Servermaschine darunter leiden. Grundsätzlich wird vor allem serverseitig sehr viel Wert darauf gelegt werden müssen, Ausnahmestände zu erkennen und schadensminimierend zu behandeln.

8.6 Java 1.4

In den Dokumentationen zu den ersten Veröffentlichungen von Java 1.4 wurde immer wieder darauf hingewiesen, dass erhebliche Anstrengungen unternommen wurden, um die Performanz der Java-Klassen erheblich zu verbessern. Die Prüfung dieser Behauptung sollte mit MD*Crypt unbedingt erfolgen.

Weiter wird die Java-Cryptographic-Extension Teil des Java1.4 Software Development-Kits sein. Diese wird es ermöglichen, vergleichsweise einfach und fehlersicher die Verschlüsselung der Datenströme zu implementieren. Dies ist eine Anforderung, die mit weiteren Einsatzszenarien MD*Crypts immer wichtiger werden wird.

Literatur

- Aydinli, G. 'Netbased Spreadsheets in Quantitative Finance' (Netzbasierende Tabellenkalkulationen in der Statistik der Finanzmärkte). 2002. In: "Applied Quantitative Finance", Berlin: Springer Verlag.
- Aydinli, G. , Härdle, W. , Kleinow, T. und Sofyan, H. 'Linking Office Applications with XploRe' (Verbinden von Office Anwendungen mit XploRe). erscheint in 2002. In: *Computational Statistics; special edition*, Heidelberg: Physica-Verlag.
- Donehower, R., Schatzman, J., 'Hindernisse auf der Überholspur – Teil 1: Performanzprobleme' September/Oktober 2001. In: *Java Spektrum* bei 101communications (Deutschland) GmbH. S. 38 - 53.
- Gülcü, C., 'Short Introduction to log4j' (Eine kurze Einführung in log4j) [online] Juli 2001 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
<http://jakarta.apache.org/log4j/docs/manual.html>
Vollständig im Anhang.
- Harold, E. R. 'Java I/O' (Java Ein- und Ausgabe), März 1999. O'Reilly & Associates Inc.
- Hemrajani, A. 'Programming with Java I/O Streams' (Mit Java Eingabe-/Ausgabe Strömen programmieren) [online] November 1998 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
<http://developer.java.sun.com/developer/technicalArticles/Streams/ProgIOStreams/>
Vollständig im Anhang.
- Joy, B., Bracha, G., Gosling, G., Steele, G. 'The Java Language Specification, Second Edition'. 5. Juni 2000. Addison-Wesley Pub Co. Kapitel 7.2.
- Kleinow, T. und Lehmann, H. 'Client/Server based Statistical Computing' (Client-/Serverbasiertes statistisches Rechnen). erscheint in 2002. In: *Computational Statistics*, Heidelberg: Physica-Verlag.

-
- Kluge, K. 'The Experts Talk: Thirteen Great Ways to Increase Java Performance' (Das Expertengespräch: Dreizehn gute Wege die Leistung von Java zu verbessern) [online] Februar 1997 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
<http://developer.java.sun.com/developer/technicalArticles/Programming/Performance/>
Vollständig im Anhang.
- McCluskey, G. 'Tuning Java I/O Performance' (Verbessern der Java Ein-/Ausgabe Leistung) [online] März 1999 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
<http://developer.java.sun.com/developer/technicalArticles/Programming/PerfTuning/>
Vollständig im Anhang.
- Reilly, D. 'Simple Handling of Network Timeouts' (Einfaches Behandeln von Zeitüberschreitungen im Netz) [online] Oktober 1999 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
<http://developer.java.sun.com/developer/technicalArticles/Programming/Performance/>
Vollständig im Anhang.
- Shirazi, J. 'Java Performance Tuning' (Java Leistungsverbesserung), September 2000. O'Reilly & Associates Inc.
- Sun Java 'Java Cryptographic Extension 1.2.1 API Specification & Reference' (Java Kryptografieerweiterung 1.2.1, API Spezifikation und Referenz), [online] Juli 2001 [zitiert am 19. März 2002]. HTML. Im World Wide Web verfügbar:
http://java.sun.com/products/jce/doc/guide/API_users_guide.html
- Object Management Group 'OMG Unified Modelling Language Specification' (OMG Vereinheitlichte Modellierungssprache, Spezifikation), [online] September 2001 [zitiert am 19. März 2002]. PDF. Im World Wide Web verfügbar:
<http://www.omg.org/cgi-bin/doc?formal/01-09-67>

Resourcen im World Wide Web

MD*Crypt Homepage:

<http://www.md-crypt.com>

Homepage des XQC-Projektes:

<http://www.xplore-stat.de/java/java.html>

Homepage MD-Books:

<http://www.md-book.com>

Homepage des VisAD Projektes:

<http://www.ssec.wisc.edu/~billh/visad.html>

Homepage des Microsoft Excel AddIns ReX:

<http://www.md-rex.com>

Sun Java Homepage:

<http://java.sun.com>

Unified Modelling Language:

<http://www.uml.org>

GraFitI Projekt:

<http://www.stat.uni-muenchen.de/blauth/GraphFitI/graphFitI.html>

Rweb html-interface:

<http://www.math.montana.edu/Rweb/>

Homepage des E-Stat Projekts:

<http://www.emilea.de>

Abbildungsverzeichnis

1	Konzept der Bytestreamserialisierung	7
2	Die dreistufige XQS-/MD*Crypt-Architektur	10
3	Zwei grundlegende Fehlerquellen der Bytestreamkommunikation .	19
4	Klasse LogFile und ihre abgeleitete TimeLogFile	23
5	Engpass in der XQS/MD*Serv Kommunikation	27
6	Für dasselbe Quantlet gemessene Verweilzeiten nach gelesenen Bytes	34
7	Vergleich der neuen und alten Lesemethoden	36
8	XQCDoubleArray-Klasse und ihre Ableitungen	41
9	Auch grafische Kontexte mit hohem Datentransfer werden möglich	51