Hochschule für Angewandte Wissenschaften
Fakultät Informatik und Mathematik
Lehrgebiet Wirtschaftsinformatik

Peter Mandl
Andreas Bakomenko
Johannes Weiß

Datenkommunikation

Studienarbeit im Wintersemester 2011/2012

Bachelorstudiengang Wirtschaftsinformatik

München, September 2011

Inhaltsverzeichnis

Web-Links

www.eclipse.org: Homepage von Eclipse

www.java.sun.com: Java Home Page von Sun Microsystems

www.msdn.microsoft.com: Microsoft Developer Home Page

www.prof-mandl.de: Webseite von Prof. Mandl mit Material zur Studienarbeit

http://ccwi.cs.hm.edu/mediawiki/: Wiki des Technologie- und Toolservice des CCWI

1 Einführung

Im Fach Datenkommunikation ist eine Studienarbeit begleitend zur Vorlesung anzufertigen. Diese wird in kleinen Teams erstellt. Gemäß Studien- und Prüfungsordnung geht die Prüfungsleistung in die Gesamtnote des Moduls Datenkommunikation ein.

Die Studienarbeit befasst sich mit der Konzeption und Implementierung von Kommunikationssoftware und dient der Sammlung von Erfahrungen in der Entwicklung verteilter Anwendungen. Spezielle Aspekte der Entwicklung und des Einsatzes von Kommunikationsanwendungen sowie Problemstellungen des Designs von verteilten Anwendungen sollen vertieft werden. Die Schwerpunkte der Studienarbeit liegen in der Protokollkonzeption und -implementierung sowie in der Validierung (Test) und der Leistungsbewertung einer selbst entwickelten Kommunikationssoftware.

2 Inhalt der Studienarbeit

2.1 Überblick

Konkretes Ziel der Studienarbeit in diesem Semester ist es, die Socket-Programmierung für TCP-Sockets und UDP-Datagram-Sockets zu erlernen und Erfahrungen in der Programmierung von Client-/Server-Anwendungen auf Basis herkömmlicher Transportprotokolle zu sammeln. Als Programmiersprache verwenden wir Java mit den dort vorhandenen Java-Socket-Klassen.

Im Team wird ein einfaches Echo-Client-/Serverprogramm (kurz: Echo-Anwendung) auf Basis verschiedener Transportsysteme entwickelt. In der Echo-Anwendung soll ein Clientprogramm Nachrichten (Echo-Request) an ein Serverprogramm senden, das diese Nachrichten wieder zurücksendet (Echo-Response). Dabei soll der Server mehrere Clients nebenläufig bedienen können und die aktiven Clients in einer Liste verwalten. Nach der letzten gesendeten Nachricht, dies wird vom Client angezeigt, soll der Client wieder aus der Liste im Server entfernt werden. Die serverseitig zu verwaltende Clientliste soll in einer Instant der Klasse *ConcurrentHashMap* abgelegt werden. (siehe *Java-Package java.util.concurrent.ConcurrentHashMap*).

Client und Server sollen jeweils in eigenen Betriebssystemprozessen instanziiert werden und wahlweise auf einem oder auf zwei Rechner verteilt ablauffähig sein.

Für die Nachrichten wird ein festes PDU-Format verwendet, das in der vorgegebenen Java-Klasse *EchoPDU* definiert ist und bei der Kommunikation zwischen Client und Server als serialisiertes Java-Objekt übertragen wird.

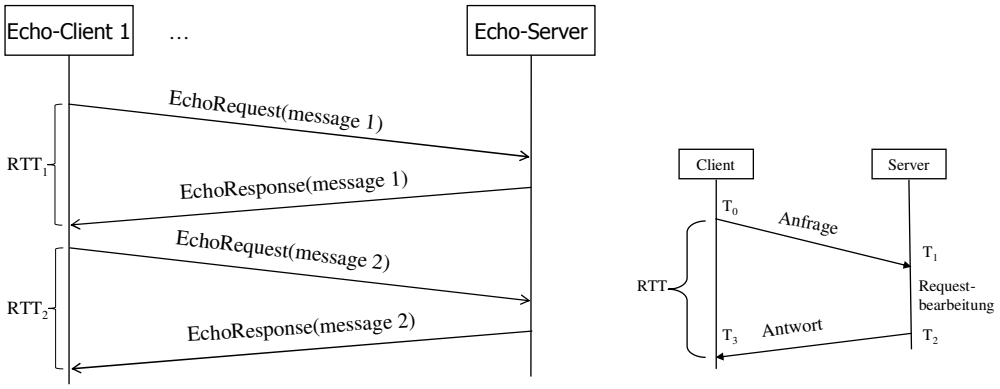

Abbildung 1: Echo-Kommunikation zwischen Echo-Client und Echo-Server

Der einfache Nachrichtenfluss zwischen Client und Server ist in Abbildung 1 skizziert. In der Skizze ist auch die sog. Round Trip Time (RTT) angegeben. Dies ist die Zeit vom Absenden eines Echo-Requests bis zur Ankunft des Echo-Response (beides beim Client):

$$RTT = T_3 - T_0.$$

Die Latenzzeit für die reine Kommunikation (wir nennen sie T_k) lässt sich ebenfalls ermitteln, indem man noch die Bearbeitungszeit für den Echo-Request im Server abzieht. Damit ergibt sich gemäß Skizze aus Abbildung 1

$$T_k = (T_3 - T_0) - (T_2 - T_1).$$

T_k beinhaltet die Bearbeitungszeit in allen Protokollinstanzen der Schicht 1 bis 4 (komplettes Transportsystem) im Client und im Server sowie die Zeit, welche die Nachrichten im Netzwerk benötigen.

Für unsere Betrachtung spielt vor allem die RTT eine wichtige Rolle, da wir auf Basis dieser Kennzahl einen Leistungsvergleich durchführen wollen.

Eine Echo-PDU (Java-Klasse *EchoPdu*) enthält folgende Informationen:

- Name des Client-Threads, der den Echo-Request absendet (beliebige, eindeutige Bezeichnung eines Clients-Threads).
- Name des Threads, der den Echo-Request im Server bearbeitet (beliebige, eindeutige Bezeichnung des Server-Threads).
- Echo-Nachricht (eigentliche Nachricht in Textform)
- Kennzeichen, ob es sich um die letzte Nachricht eines Clients handelt, bevor dieser sich beendet. Dieses Kennzeichen dient dem Server dazu, um festzustellen, ob ein Client nach der Bearbeitung einer empfangenen Nachricht aus der Clientliste entfernt werden kann.
- Zeit in Nanosekunden, die der Server zur Bearbeitung des Requests benötigt. Diese Zeit muss der Server messen ($T_2 - T_1$) und vor dem Absenden der Response in die Echo-PDU eintragen.

Um den Leistungsunterschied bei der Bearbeitung auf Basis der Transportprotokolle TCP und UDP berechnen zu können, soll die Echo-Anwendung sowohl mit UDP (UDP-Datagram-Sockets) als auch mit TCP (TCP-Sockets) entwickelt werden.

Um ein Gefühl für die Komplexität einer gesicherten, verbindungsorientierten Kommunikation über TCP im Vergleich zu verbindungslosen Kommunikationsprotokollen wie UDP zu entwickeln, soll die Echo-Anwendung zudem auf Basis eines selbstentwickelten Transportprotokolls implementiert werden. Hier gibt es in der Studienarbeit wahlweise zwei Möglichkeiten. Entweder man programmiert selbst ein gesichertes Transportprotokoll auf Basis von UDP oder man erweitert eine bestehende Implementierung, die mit ausgeliefert wird und als „LWTRT-Protokoll“ (Lightweight Transport Reliable Protocol) bezeichnet wird. LWTRT setzt auf UDP auf und stellt einige Mechanismen zur Unterhaltung einer gesicherten Verbindung, ähnlich wie TCP, aber wesentlich einfacher, zur Verfügung. Dazu muss die in der Aufgabenstellung bereitgestellte LWTRT-Schicht analysiert und an einigen wichtigen Stellen ergänzt werden.

Schließlich soll in der Studienarbeit auch noch eine Implementierung der Echo-Anwendung auf Basis des komfortableren Java RMI (Remote Method Invocation) entwickelt werden, um auch hier die Unterschiede höherer Kommunikationsprotokolle, die schon eine Client-/Server-Kommunikation explizit unterstützen, zur Nutzung klassischer Transportprotokolle wie TCP und UDP einordnen zu können.

Die TCP- und UDP-basierten Echo-Anwendungen sollen in einem Single-threaded und in einem Multithreaded Server realisiert werden. Die Implementierungen mit Java-RMI und LWTRT bzw. der eigenen Implementierung sollen nur als Multi-treaded Server ausgeprägt sein.

Alle Implementierungen sind bezüglich ihrer Leistungsfähigkeit zu untersuchen und miteinander zu vergleichen. Hierzu ist ein Benchmarking-Client zu programmieren, über den eine einfache Möglichkeit der Parametrisierung besteht. Die erforderlichen Parameter sind vordefiniert.

Serverseitig soll als "Service-Port" sowohl für UDP-Datagram-Sockets als auch für TCP-Sockets der Port 50000 verwendet werden. Clientseitig werden die Ports ab 51000 genutzt.

Folgende Einzelaufgaben sind in der Studienarbeit am besten in der angegebenen Reihenfolge auszuführen:

1. Entwicklung einer TCP-basierten Echo-Anwendung mit einem Single-threaded Server
2. Entwicklung einer TCP-basierten Echo-Anwendung mit einem Multi-threaded Server

3. Entwicklung einer UDP-basierten Echo-Anwendung mit einem Single-threaded Server
4. Entwicklung einer UDP-basierten Echo-Anwendung mit einem Multi-threaded Server
5. Entwicklung eines eigenen gesicherten Transportprotokolls auf Basis von UDP einschließlich einer zugehörigen Echo-Anwendung mit Multi-threaded Server oder alternativ die Vervollständigung der LWTRT-Protokollimplementierung und die Entwicklung einer LWTRT-basierten Echo-Anwendung mit Multi-threaded Server
6. Entwicklung einer RMI-basierten Echo-Anwendung mit Multi-threaded Server
7. Leistungsbewertung aller Implementierungen und Vergleich der verschiedenen Lösungen

Die Teilaufgaben sollen im Weiteren kurz erläutert werden.

2.2 Teilaufgabe 1: Single-threaded TCP-Server

In dieser Teilaufgabe soll ein Echo-Server und ein passender Echo-Client auf Basis von TCP-Sockets entwickelt werden, der alle Echo-Requests in einem einzigen Thread (Single-threaded) bearbeitet.

Für jeden Echo-Request baut ein Client eine TCP-Verbindung zum Server auf, sendet eine Echo-PDU, wartet auf das Ergebnis und baut anschließend die Verbindung wieder ab.

In dieser Teilaufgabe der Studienarbeit ist auch die Basis eines parametrisierbaren "Benchmarking-Clients" für einen Leistungstest zu programmieren. Dieser Client kann später auf die anderen Implementierungsvarianten angepasst werden.

Folgende Parameter sollen beim Starten des Benchmarking-Clients eingestellt werden können:

- Anzahl der Clients, die jeweils in einem eigenen Threads gestartet werden sollen
- Anzahl der Nachrichten, die jeder Client senden soll
- Denkzeit eines Clients zwischen zwei Echo-Requests (in ms)
- Länge der Nutzdaten einer Echo-Nachricht (konstante Länge für alle Nachrichten eines Testlaufs in Byte)

Der Benchmarking-Client soll vor allem RTT-Mittelwerte zur Bearbeitungszeit für einen Echo-Request ermitteln und sowohl für den Funktionstest des Servers als auch für den Leistungstest dienen. Weitere Informationen zu den Leistungsmessungen sind in Kapitel 2.9 zu finden.

In Abbildung 1 sind die Systembestandteile der ersten Teilaufgabe skizziert:

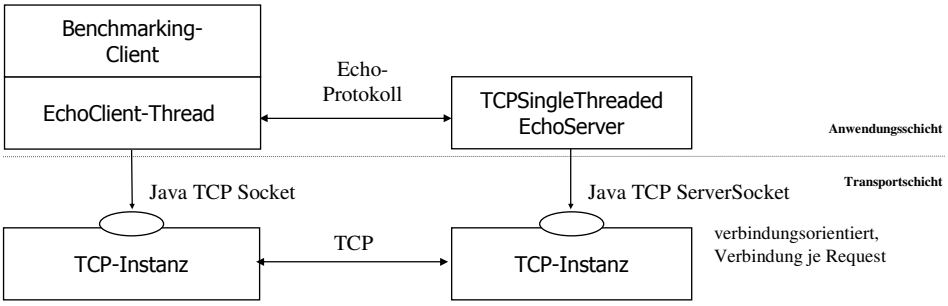

Abbildung 1: Systembestandteile für 1. Teilaufgabe

Als Basis für die Sammlung statistischer Informationen kann die mit der Aufgabenstellung ausgelieferte Klasse *SharedClientStatistics* dienen und bei Bedarf modifiziert werden. An geeigneten Stellen im Benchmarking-Client sind die Methoden dieser Klasse *SharedClientStatistics* zu nutzen. Die Entwicklung einer grafischen Oberfläche für den Benchmarking-Client ist optional. Für eine Lösung reicht auch die Parametrisierung über Java-Konstanten im Sourcecode, da die Entwicklung der Datenkommunikation im Vordergrund steht.

2.3 Teilaufgabe 2: Multi-threaded TCP-Server

In dieser Teilaufgabe soll ein Echo-Server und ein passender Echo-Client auf Basis von TCP entwickelt werden, in dem für jeden Client-Thread ein (serverseitig) eigener "Worker-Thread" bereitgestellt wird, der die Verbindung zum jeweiligen Client-Thread bedient. Jeder Client-Thread baut also eine Verbindung zum Echo-Server auf. Diese wird solange gehalten, bis der Client-Thread alle Echo-Requests gesendet und die Echo-Responses erhalten hat.

Als Basis der Clientsoftware für diese Teilaufgabe soll der Benchmarking-Client aus Teilaufgabe 1 dienen. Eine entsprechende Anpassung der Clientseite ist erforderlich. In Abbildung 2 sind die Systembestandteile der zweiten Teilaufgabe skizziert:

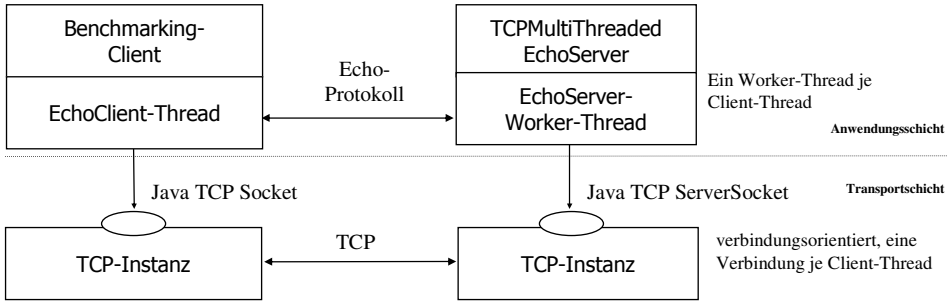

Abbildung 2: Systembestandteile für 2. Teilaufgabe

2.4 Teilaufgabe 3: Single-threaded UDP-Server

In dieser Teilaufgabe soll ein Echo-Server und ein passender Echo-Client auf Basis von UDP-Datagram-Sockets entwickelt werden, der die Echo-Requests aller Client-Threads in einem einzigen Thread (Single-threaded) ausführt (siehe Abbildung 3). Der Benchmarking-Client ist entsprechend anzupassen bzw. zu erweitern.

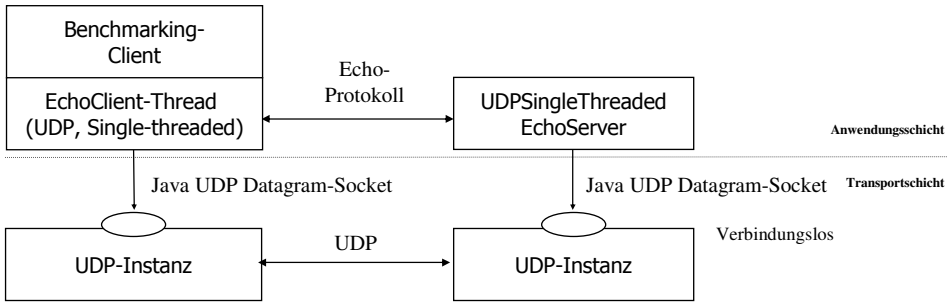

Abbildung 3: Systembestandteile für 3. Teilaufgabe

2.5 Teilaufgabe 4: Multi-threaded UDP-Server

In dieser Teilaufgabe soll ein Echo-Server und ein passender Echo-Client auf Basis von UDP-Datagram-Sockets entwickelt werden (Abbildung 4), der jeden Request in einem eigenen Thread bedient (Multi-threaded). Der Benchmarking-Client ist entsprechend anzupassen bzw. zu erweitern.

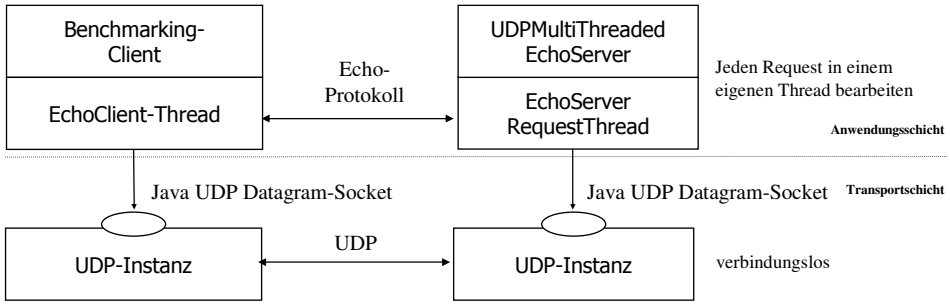

Abbildung 4: Systembestandteile für 4. Teilaufgabe

2.6 Teilaufgabe 5: Alternative 1 - Eigene Implementierung

Diese Teilaufgabe befasst sich mit der Entwicklung eines eigenen gesicherten Transportprotokolls auf Basis von UDP sowie eines entsprechenden multi-threaded Echo-Servers und eines passenden Echo-Clients. Die Erweiterung bzw. Anpassung des Benchmarking-Clients ist auch hier erforderlich.

Die Architektur ist in Abbildung 5 skizziert:

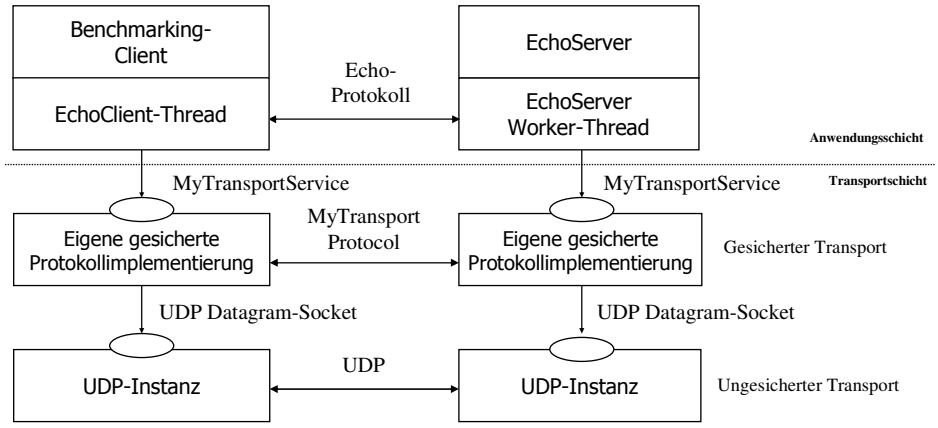

Abbildung 5: Systembestandteile für 5. Teilaufgabe (Alternative 1)

Folgende Protokollmechanismen zur Erhöhung der Zuverlässigkeit sollten implementiert werden:

- Einfaches Stop-and-Go-Protokoll mit positiv-selektivem Quittungsverfahren. Jede Nachricht an einen Empfänger sollte erst von diesem bestätigt werden, bevor die nächste gesendet werden darf. Bis zur Empfang der Antwort blockiert der Sender-Thread. Dabei sollen auch Duplikate vermieden werden.
- Eine Timerüberwachung für jede Nachricht ist einzubauen. Nach Ablauf von n Sekunden (konfigurierbar) ohne Bestätigung kann entweder die Nachricht oder die Bestätigung verloren gegangen sein und es wird eine Neuübertragung ausgeführt. Nach k-maliger Wiederholung (konfigurierbar) einer Nachricht wird die Übertragung eingestellt und ein Fehler gemeldet.
- Der Ausfall eines Partners sollte erkannt werden und es sollte auf eine derartige Ausnahme sinnvoll reagiert werden. Ein Garbage-Collection zur Bereinigung der client- und serverseitigen Datenstrukturen (Kontextbereinigung) ist zu realisieren.

Auf das Erkennen von Übertragungsfehlern (Bitfehler), z.B. durch das Senden eines CRC-Codes, in den Nachrichten-Headern kann verzichtet werden. Hier verlassen wir uns auf die Mechanismen von UDP.

2.7 Teilaufgabe 5: Alternative 2 - LWTRT-Implementierung

Diese Teilaufgabe befasst sich als Alternative zu Kapitel 2.6 mit der Entwicklung eines Echo-Servers auf Basis des vorgegebenen LWTRT-Protokolls, der für jeden Client einen eigenen Worker-Thread für die Verbindung bereitstellt. Die Erweiterung bzw. Anpassung des Benchmarking-Clients ist auch hier erforderlich. Ein passender Echo-Client ist ebenfalls zu programmieren.

Das vorgegebene LWTRT-Protokoll stellt ein stark vereinfachtes, verbindungsorientiertes Transportprotokoll auf Basis von UDP dar. Die Protokollimplementierung ist unabhängig von der nutzenden Anwendung und stellt für die Anwendungsschicht einen LWTRT-Dienst (bereitgestellt über die Java-Interfaces *LWTRTService* und *LWTRTConnection*) bereit, über den eine Registrierung möglich ist, eine Verbindung auf- und abgebaut werden kann und beliebige Nachrichten übertragen werden können.

Die Implementierung des Kommunikationsprotokollautomaten nutzt eine Form des sog. *State-Patterns*. Der Sourcecode ist vorgegeben, jedoch fehlen an einigen Stellen der Implementierung Sourcecode-Teile, die zu ergänzen sind, damit das Protokoll ordnungsgemäß funktioniert.

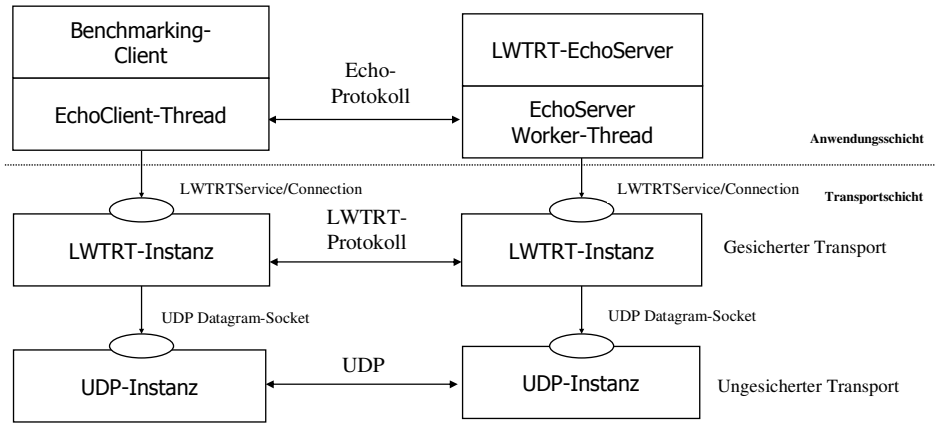

Abbildung 6: Systembestandteile für 5. Teilaufgabe (Alternative 2)

Die Schichtenarchitektur ist in Abbildung 6 skizziert. Die verschiedenen Schichten haben folgende Funktionalität (von unten nach oben):

- *Die UDP-Instanz* stellt einen verbindungslosen und unzuverlässigen Dienst zur Verfügung, der in Java mit Datagram-Sockets verwendet werden kann
- *Die LWTRT-Instanz* stellt einen zuverlässigeren Transportdienst bereit, der unabhängig von den darüberliegenden Schichten für beliebige Anwendungen einen „leichtgewichtigen" (lightweight) Transportdienst bereitstellt, der zwar nicht so zuverlässig wie TCP ist, aber doch eine gewisse Zuverlässigkeit bietet. Die Dienste an der Dienstschnittstelle und auch der genaue Protokollablauf sind vorgegeben. Weiterhin werden Java-Interfaces und auch einige Java-Klassen bereits vorgegeben. Diese sind zu nutzen.
- Die *Echo-Anwendung* bestehend aus dem Echo-Client (inkl. Benchmarking-Client) und dem Echo-Server (Multi-threaded) nutzt den LWTRT-Dienst zur gesicherten Kommunikation.

Die Spezifikation des LWTRT-Protokolls wird in der Übung diskutiert und in einem eigenen Foliensatz bereitgestellt.

Bei der Realisierung des Protokolls muss berücksichtigt werden, dass UDP nicht zuverlässig ist. Die Kommunikation über LWTRT wird daher durch einige Protokollmechanismen abgesichert. Beispielsweise werden Nachrichtenverluste entdeckt, verlorene Nachrichten erneut angefordert und es werden Duplikate vermieden. Folgende Protokollmechanismen zur Erhöhung der Zuverlässigkeit sind zum Teil implementiert und an den fehlenden Stellen zu ergänzen:

- Einfaches Stop-and-Go-Protokoll mit positiv-selektivem Quittungsverfahren. Jede Nachricht an einen Empfänger sollte erst von diesem bestätigt werden, bevor die nächste gesendet werden darf. Der Sender-Thread blockiert solange. Dabei sollen auch Duplikate vermieden

werden.

- Eine Timerüberwachung für jede Nachricht ist einzubauen. Nach Ablauf von n Sekunden (Wartezeiten festgelegt) ohne Bestätigung kann entweder die Nachricht oder die Bestätigung verloren gegangen sein und es wird eine Neuübertragung ausgeführt. Nach k-maliger Wiederholung (Anzahl der Wiederholungen ist festgelegt) einer Nachricht wird die Übertragung eingestellt und ein Fehler nach oben gemeldet.
- Der Ausfall eines Partners sollte erkannt werden und es sollte auf eine derartige Ausnahme sinnvoll reagiert werden. Ein Garbage-Collection zur Bereinigung der client- und serverseitigen Datenstrukturen (Kontextbereinigung) ist in der vorgegebenen Implementierung enthalten.

2.8 Teilaufgabe 6: Multi-threaded RMI-Server

In dieser Teilaufgabe soll ein Echo-Server und ein entsprechender Echo-Client auf Basis von Java RMI entwickelt werden. Java RMI ermöglicht eine höherwertige Client-/Server-Kommunikation und verfügt bereits implizit über einen Multi-threaded Server.

Ein Echo-Dienst wird über eine Remote-Schnittstelle definiert und einer RMI-Registry, die in unserem Beispiel in den Echo-Server integriert werden soll, bekanntgemacht. Die RMI-Registry dient dem Client dazu, die Remote-Adresse des zuständigen Serverobjekts zu ermitteln. Die Abfrage einer Remote-Adresse erfolgt über einen vordefinierten Port (standardmäßig der TCP-Port 1099).

Die Architektur der RMI-Lösung ist in Abbildung 7 skizziert.

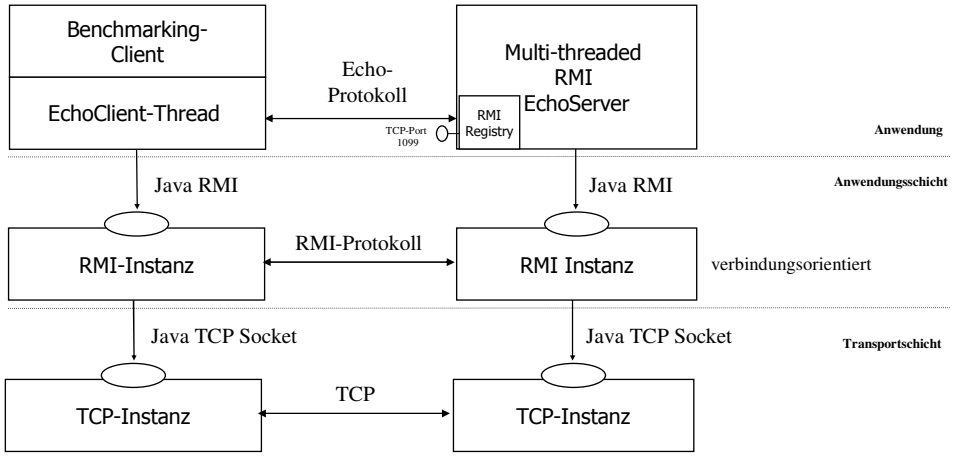

Abbildung 7: Systembestandteile für 6. Teilaufgabe

2.9 Teilaufgabe 7: Leistungsbewertung und Vergleich

Für alle Implementierungen sind Messungen durchzuführen, um vor allem die RTT-Werte vergleichen zu können.

Neben einer tabellarischen Darstellung der Ergebnisse der Lasttests sind zwei grafische Darstellungen (siehe Abbildung 8) zu erarbeiten, die folgendes aufzeigen sollen:

- Entwicklung der durchschnittlichen RTT bei Veränderung der Anzahl an Threads von 100 auf 1000 in 100er-Schritten und bei konstanter Nachrichtenlänge von 50 Byte und einer Denkzeit von 100 ms.
- Entwicklung der durchschnittlichen RTT bei Veränderung der Größe der Nachrichtenlänge von 100 auf 1000 Byte in 100er-Schritten und bei konstanter Anzahl an Threads (500 Threads) und einer Client-Denkzeit von 100 ms.

Die Messungen sind mindestens fünf Mal zu wiederholen. Um statistisch vernünftige Aussagen zu erhalten, sind die RTT-Mittelwerte über die Wiederholungen zu berechnen.

Mit Hilfe der vorgegebenen Statistik-Klasse können die Werte ermittelt und in eine Datei geschrieben werden. Am Ende jeder Einzelmessung soll ein Datensatz erzeugt werden, der u.a. die Clientanzahl, die durchschnittliche RTT und die Anzahl verlorener Echo-Responses enthält. Die Daten können mit MS Excel oder einem anderen Tabellenkalkulationsprogramm nachbearbeitet und ausgewertet werden.

Die Lasttests sollen mit einem Server- und einem Clientrechner, die miteinander über ein Ethernet-LAN vernetzt sind, durchgeführt werden. Die Laborrechner sind zu verwenden. Ein dediziertes Netzwerk wäre für die Messungen sinnvoll, kann aber nicht bereitgestellt werden.

Bei der Beschreibung der Messergebnisse sind die verwendete Rechnerausstattung und die Softwarestände sowie die Netzwerkanbindungen genau anzugeben.

Die Entwicklung der Heap-Größe und der CPU-Auslastung während der Tests sollen zur Laufzeit auch mit dem Tool *JConsole* beobachtet und dokumentiert werden, um ein Gefühl für den Ressourcenverbrauch zu bekommen.

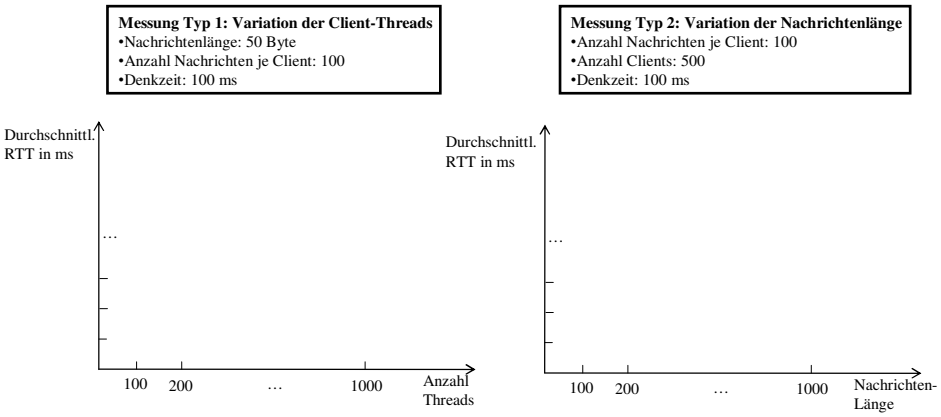

Abbildung 8: Grafische Darstellung der Messergebnisse

3 Hinweise

3.1 Programmierung und Test

Die Programmierung erfolgt in Java mit Eclipse als Entwicklungswerkzeug, als Java-Version wird 1.6 (siehe URL www.java.sun.com) verwendet.

Für die Versionsverwaltung der Sourcen wird jeder Gruppe auf Anfrage ein *svn*-Repository angelegt, in dem die Sourcen abgelegt werden können (*svn* kann in Eclipse verwendet werden).

Zum Test ist der Einbau einer Trace-/Logging-Funktionalität an geeigneten Stellen des Codings auf Basis von Log4j erforderlich. Ziel ist die Erreichung einer verbesserten Test- und Diagnosemöglichkeit, die gerade in verteilten Anwendungen wichtig ist.

Schreiben Sie die Software so, dass Sie die Echo-Anwendung auch auf einem einzigen Rechner testen können.

Vor den Lasttests sollten Sie Funktionstests durchführen, um eine ausreichende Stabilität sicherzustellen. Testen Sie auch Ausfallsituationen wie den Absturz eines Servers, eines Clients oder den Ausfall einer Netzverbindung.

3.2 Verfügbare Basisklassen

In der Vorgabe zur Studienarbeit sind einige Java-Klassen bereits mit ausgeliefert, die Sie verwenden sollten. Im Java-Package *hm.edu.dako.EchoApplication.Basics* finden Sie u.a. die Klassen *SharedClientStatistics* und *EchoPdu*. Erstere dient der Sammlung von Messdaten und kann im

Benchmarking-Client genutzt werden. Letztere ist zwingend für die Kommunikation zwischen Client und Server zu verwenden.

In den Java-Package *hm.edu.dako.EchoApplication.Lwtrt.** sind alle Java-Klassen und -Interfaces der vorgegebenen LWTRT-Schicht enthalten. Diese Klassen enthalten an einigen markierten Stellen Lücken, die zu ergänzen sind.

Weiterhin sollte die vorhandene Java-Basisklasse *ConcurrentHashMap* für die Implementierung der Clientliste im Server verwendet werden. Für die einzelnen Implementierungen sind vor allem die Java-Packages *java.io, java.net. java.rmi* erforderlich.

Für den Einbau der Logging-Funktionalität soll die Apache-Lösung *Logj4* verwendet werden (Java-Packages *org.apache.commons.logging.*, org.apache.log4j.**).

3.3 Sonstige Hinweise

Folgende Hinweise sollten für die Erstellung der Studienarbeit noch beachtet werden:

- Jedes Team besteht aus maximal vier Entwicklern.
- Die Dokumentation wächst mit jeder Teilaufgabe und sollte je Teilaufgabe aus maximal 3 Powerpoint-Folien bestehen (Erläuterung der Architektur und der wichtigsten Designentscheidungen. Lasttestergebnisse, wichtige Sourcecode-Teile, Resümee). Bitte verwenden Sie vor allem Bilder, Skizzen und evtl. UML-Diagramme (Klassendiagramme, …), sofern diese schon aus anderen Veranstaltungen bekannt sind.
- Der Sourcecode ist ausreichend nach Java-Standard zu dokumentieren. Die gesamte Dokumentation zu den Sourcen sollte im Coding untergebracht werden.
- Der komplette Sourcecode und die Dokumentation sind in elektronischer Form im ZIP-Format an den zuständigen Dozenten zu übergeben.
- Die Vorstellung der Ergebnisse (Workshop = Kolloquium) erfolgt gegen Ende des Semesters. Als Diskussionsgrundlagen sind vor allem die lauffähigen Echo-Anwendungen und die vorbereiteten Powerpoint-Folien (Handout) zu nutzen.
- Es muss klar ersichtlich sein, welches Teammitglied sich mit welchen Arbeiten im Team befasst hat. Dies sollte während des Workshops kurz dargelegt werden. Jedes Teammitglied muss seinen Teil selbstständig verteidigen können.

4 Meilensteine, Termine und Bewertung

Die Ergebnisse der Studienarbeit mit allen Programmen einschließlich der Dokumentation sind nach Abstimmung mit dem Dozenten termingerecht abzugeben. Arbeiten Sie sich also zügig in die Thematik. Der folgende Zeitplan gerechnet ab der 1. Oktoberwoche sollte bei sequentieller Bearbeitung als grober Anhalt dienen:

- Teilaufgabe 1: Abschluss bis zum Ende der 2. Woche (Basis des Benchmarking-Clients hier auch gleich entwickeln)
- Teilaufgaben 2 bis 4: Abschluss bis zum Ende der 4. Woche
- Teilaufgabe 5: Abschluss bis zum Ende der 8. Woche (Achtung: Wesentlich aufwändiger als die anderen Teilaufgaben)
- Teilaufgabe 6: Abschluss bis zum Ende der 9. Woche
- Teilaufgabe 7: Abschluss bis zum Ende der 11. Woche (Vorsicht: Lasttests erfordern Zeit und gründliches Arbeiten)

Die Studienarbeit erfordert ein kontinuierliches Arbeiten an der Lösung. Jedes Team sollte sich einen Projektplan entwerfen, der zu jeder Teilaufgabe einen Meilenstein definiert. Eine Parallelisierung der Bearbeitung ist durchaus sinnvoll.

Die Termine für die Abgabe der Arbeiten sowie für die Präsentation der Ergebnisse werden im Laufe des Semesters abgestimmt.

4.1 Bewertungssystem

Für die Bewertung der Studienarbeit werden Punkte je Teilaufgabe vergeben. Insgesamt können während der Studienarbeit maximal 100 Punkte gesammelt werden:

- Teilaufgabe 1: max. 15 Punkte
- Teilaufgabe 2: max. 5 Punkte
- Teilaufgabe 3: max. 5 Punkte
- Teilaufgabe 4: max. 5 Punkte
- Teilaufgabe 5: max. 30 Punkte (Alternative 1 oder 2 wahlweise)
- Teilaufgabe 6: max. 5 Punkte
- Teilaufgabe 7: max. 15 Punkte
- Abschlusspräsentation: max. 20 Punkte

Als Mindestpunkteanzahl zum Bestehen der Studienarbeit sind 50 Punkte festgelegt.

Folgender Notenschlüssel wird zugrundegelegt:

Punkte	**Note**
96 – 100	1,0
93 – 95	1,3
90 – 92	1,7
86 – 89	2,0
82 – 85	2,3
78 – 81	2,7
73 – 77	3,0
65 – 72	3,3
59 – 64	3,7
50 – 58	4,0
< 50	5,0

4.2 Hinweis zur Abgabe und Präsentation

Die Abgabe der Studienarbeiten muss bis **spätestens Donnerstag, den 15.12.2011, 16:45 Uhr** (Ende der Übungen) erfolgen. Bitte Puffer einbauen! Die Abgabe erfolgt per E-Mail direkt an den zuständigen Dozenten unter Angabe des Gruppennamens und der Gruppenteilnehmer. Eine Bestätigung der Abgabe erfolgt ebenso per E-Mail.

Abgegeben wird ein lauffähiges Eclipse-Projekt, das importiert werden kann (bitte testen) mit dem gesamten Sourcecode zum Projekt inkl. einer Inline-Dokumentation im Coding und den Präsentationsfolien. Sollte die Software nicht oder nicht vollständig funktionieren, ist dies bei der Abgabe zu dokumentieren. Schreiben Sie in der E-Mail auch dazu, welche Teile durch das Team bearbeitet wurden und welche nicht.

Die Termine für die Präsentation und Diskussion der Ergebnisse werden in der Vorlesung und/oder im Web (www.prof-mandl.de) bekanntgegeben. Über das schwarze Brett wird darüber informiert. Wie Sie präsentieren, ist Ihnen freigestellt. Anfangs sollten auf alle Fälle das Team und die Aufgabenverteilung vorgestellt werden.

5 E-Mail-Adressen der Dozenten und Tutoren

Dozenten:

- LBA Andreas Bakomenko: a.bakomenko@xt-ag.com
- LBA Johannes Weiß: johannes.weiss@hm.edu
- Prof. Dr. Peter Mandl: mandl@cs.hm.edu

Tutoren (voraussichtlich):

- Björn Rottmueller: b.rottmüller@isys-software.de
- Christian Holdschuh: c.holdschuh@isys-software.de
- Mathias Dolag: m.dolag@isys-software.de
- Najum Ali: n.ali@isys-software.de
- Johannes Forster: johannes.forster@googlemail.com