WIRTSCHAFTSUNIVERSITÄT WIEN

DIPLOMARBEIT

Titel der Diplomarbeit:
Auswirkungen von Architektur-Patterns auf das Qualitätsmerkmal Performance am Beispiel eines Web Service Frameworks

Verfasserin/Verfasser: Christoph Bäck

Matrikel-Nr.: 9551597

Studienrichtung: Betriebswirtschaft J151

Beurteilerin/Beurteiler: Univ. Prof. Dr. Gustaf Neumann

Ich versichere:

dass ich die Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

dass ich dieses Diplomarbeitsthema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Datum

Unterschrift

1 Einleitung

In dieser Diplomarbeit wird anhand eines Web Service Frameworks untersucht wie sich das Qualitätsmerkmal Performance durch Implementierung verschiedener Architektur-Patterns beeinflussen lässt, gleichzeitig wird verifiziert, ob die in der Literatur getätigten Vorhersagen bezüglich Performance sich auch so in einem produktiv-verwendbaren Softwaresystem wieder finden lassen.

Zu diesem Zwecke wurde das Open Source Framework Apache Axis[1] herangezogen, das einen einfachen Server zur Verfügung stellt. Dieser Server unterstützt in der ursprünglichen Variante nur eine iterative und eine konkurrente dem *Thread-Per-Request* Pattern [PeSo97] folgende Betriebsart; diese Arbeit erweiterte den Server um drei Varianten, die anderen Architektur-Patterns folgen.

Um die Auswirkungen dieser Architektur-Patterns auf das Qualitätsmerkmal Performance systematisch zu untersuchen, wurde eine Test Suite entworfen und implementiert. Der Nutzen dieses Tools ist, dass ein regressionsfähiger System- und Leistungstest zur Verfügung steht [vgl. GrKoZu04, S. 384ff], der auch zum Test anderer Web Service Frameworks genutzt werden kann, um diese zu vergleichen und auch deren Architektur zu validieren. Die Benchmark Suite ist von dieser Test Suite abgeleitet und verwendet einen Mix an Operationen mit steigender Last und Grad an Nebenläufigkeit für Performance-Messungen, um die verschiedenen Architekturen zu vergleichen.

In den nächsten Absätzen erfolgt ein Überblick der Kapitel und eine kurze Erläuterung über deren Inhalt.

Im Kapitel 2 werden die Begriffe Patterns, Architektur und Qualitätsmerkmal geklärt und der Zusammenhang zwischen diesen dargestellt.

---

[1] Siehe http://ws.apache.org/axis/, Abruf am 2004-08-18.

Im anschließenden Kapitel 3 werden die Grundzüge von verteilten Systemen, Web Services und der Service Oriented Architecture (SOA) dargestellt.

Server-Varianten und die dazugehörenden bzw. im Umfeld dieser liegenden Patterns und die Implementierungsdetails werden im Kapitel 4 beschrieben.

Eine Beschreibung des Test Suite Tools und der implementierten Testfälle für das Benchmarking werden im Kapitel 5 erläutert.

Was der Begriff Benchmark bedeutet, in welcher Testumgebung dieser stattgefunden hat, welche Testfälle für die Benchmark Suite ausgewählt wurden und deren Parametrierung, werden im Kapitel 6 dargestellt.

Die Ergebnisse des Benchmarkings und die Interpretation eben dieser werden in Kapitel 7 abgehandelt.

Verwandte Arbeiten werden in Kapitel 8 betrachtet, der Ausblick und Schlussfolgerungen werden in Kapitel 9 dargestellt.

Diagramme, die die Implementierung verdeutlichen sollen bzw. Entwurfsdiagramme, werden mittels der UML (Unified Modeling Language) (als Referenz dient [UML04], die Spezifikation der UML kann unter [OMG04] eingesehen werden) dargestellt.

2 Architektur-Patterns und Qualitäts-merkmale

Patterns wurden in der Software-Entwicklung Mitte der Neunziger Jahre durch die Publikation des Buches *Design Patterns* (in dt. Fassung [GHJV96]) von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides popularisiert, die Gruppe der Autoren wird auch oft nur als „Gang of Four“ bezeichnet (daher sind die in ihrem Werk publizierten Patterns auch als „Gang of Four“ bzw. GoF Patterns bekannt).

Ursprünglich fußt die Pattern-Idee in der Architektur, wie sie von Christopher Alexander in seinem Werk *Timeless Way of Building* [Alex79] dargelegt wurde, daher folgt auch die Definition von Patterns nach Alexander:

„*Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass man diese Lösung beliebig oft anwenden kann, ohne sie jemals ein zweites Mal gleich auszuführen*“ [siehe Alex79]

Das heißt, dass ein Pattern die Lösung eines in der Umwelt wiederkehrenden Problems darstellt, wobei aber die Ausführung der Lösung nicht immer gleich sein muss. Alexander definierte, dass ein Pattern eine dreiteilige Regel ist, welche die Beziehung zwischen einem Kontext, einem Problem und einer Lösung ausdrückt.

In [GHJV96], in dem Design Patterns vorgestellt werden, wurden die Bestandteile eines Patterns wie folgt definiert:

- **Mustername**
  Ein jedes Pattern besitzt einen Namen, um ein gemeinsames Vokabular zu schaffen.

- **Problem**
  Dieser Bestandteil eines Patterns beschreibt, welches Problem adressiert wird und was der Kontext des Problems darstellt.

- **Lösung**

  In diesem Teil wird die Lösung des Problems beschrieben, es wird aber nicht eine konkrete Implementierung dargestellt, sondern nur eine Schablone, die für viele verschiedenen Situationen anwendbar ist.

- **Konsequenzen**

  Beschreibt, welche Vor- und Nachteile die vorgeschlagene Lösung besitzt, da Entwurfsentscheidungen oft eine Form des Ausbalancierens verschiedener Kräfte darstellen.

Neben den vorgestellten Patterns aus der Architektur (nicht Software-Architektur) und des Software-Designs werden noch folgende Arten von Patterns unterschieden, die eine Art Hierarchie von der niedrigsten Abstraktionsstufe zur höchsten darstellen (diese Darstellung ist angelehnt an der in [POSA1, S.12]):

- **Idiome**

  Sind streng genommen keine Patterns, da sie vergleichbar zu Redewendungen natürlicher Sprache sind. D.h. Idiome sind zum Teil Konventionen (oft informell) wie bestimmte Probleme in einer Programmiersprache gelöst werden. Zum Beispiel das Kopieren eines Strings in C, welches üblicherweise mittels des folgenden Idioms: `while ( *destination++ = *source++ ) ;` bewerkstelligt wird (wenn auf die Verwendung der Standardbibliotheksfunktion `strcpy` verzichtet wird). Bekannte Idiome für C++ können u.a. in [Cope92] und [Mey92] gefunden werden.

- **Design Patterns**

  Sind Patterns, die sich mit Entwurfsproblemen beschäftigen. Das bekannteste Werk darüber ist [GHJV96].

- **Architektur-Patterns**

  Diese Patterns befassen sich mit Problemen der Software-Architektur, diese Art von Patterns sind das Thema dieser Arbeit. Bekannte Quelle

für Architektur-Patterns ist die Serie Pattern-Oriented Software Architecture ([POSA1], [POSA2] und [POSA3]).

Neben den in der vorhergehenden Aufzählung dargestellten Patterns sind u.a. noch Analysis Patterns, Pedagogical Patterns, Organizational Patterns und weitere andere bekannt.

Nachdem der Begriff des Patterns einführend erläutert wurde, muss noch der Begriff Architektur (diesmal für Software) geklärt werden.

Bass, Clements und Kazman definieren Software-Architektur wie folgt: „*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*“ [siehe BCK98, S. 23]

Diese Definition zeigt auf, dass Architektur sich mit den Beziehungen zwischen den Komponenten und der Sichtbarkeit der Eigenschaften ebendieser beschäftigt.

Eine inhaltlich ähnliche Definition findet sich in [POSA1, S. 384]: „*A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. …*“

Der letzte Satz der vorhergehenden Definition erwähnt, dass Subsysteme und Komponenten in verschiedenen Darstellungsformen spezifiziert wurden, um die verschiedenen funktionalen und nicht-funktionalen Merkmale darzustellen.

Funktionale Merkmale sind Merkmale, die sich auf funktionale Anforderungen beziehen, dies wären zum Beispiel Algorithmen, um eine bestimmte Funktion zu errechnen.

Nicht-funktionale Merkmale sind Merkmale, die nicht direkt als Anforderung beschrieben wurden, dies sind Aspekte, die sich auf Zuverlässigkeit, Wartbarkeit, Wiederverwendbarkeit, Performance usw. beziehen. Nicht-

funktionale Merkmale können zu funktionalen Merkmalen werden, wenn sie eine explizite Anforderung darstellen. Nicht-funktionale Merkmale sind auch als Qualitätsmerkmale bekannt.

Qualitätsmerkmale bzw. nicht-funktionale Merkmale können nach [BCK98] in folgende Kategorien eingeteilt werden:

- **Merkmale, die zur Laufzeit wahrnehmbar sind**
  Diese Merkmale können zur Laufzeit festgestellt bzw. gemessen werden, dies sind Merkmale wie Performance, Funktionalität, Verfügbarkeit, Usability etc.

- **Merkmale, die nicht zur Laufzeit wahrnehmbar sind**
  Diese sind nicht zur Laufzeit messbar, dies wären Merkmale wie Änderbarkeit, Portabilität, Wiederverwendbarkeit, Testbarkeit etc.

- **Business Qualitities**
  Merkmale, die sich auf die wirtschaftlichen Komponente beziehen, z.B. Time to Market, Lebenszeit des Systems, Kosten etc.

- **Qualität der Architektur**
  Dies bezieht sich auf Merkmale, wie konzeptuelle Integrität, darunter wird die Vision oder das Thema, das sich durch die Architektur zieht, verstanden, die Korrektheit und Vollständigkeit der Architektur, damit sie die Anforderungen des Systems erfüllen kann und abschließend, dass das System realisierbar ist.

Diese Qualitätsmerkmale müssen von Architekturen ausbalanciert werden, d.h. dass zwischen konfliktären Merkmalen entschieden werden muss, welches Merkmal Vorrang besitzt.

Architektur-Patterns sind also Patterns, die sich auf Architektur beziehen und dadurch Qualitätsmerkmale ausbalancieren (z.B. höhere Wartbarkeit durch Separation of Concerns vs. geringere Performance durch Einführung einer zusätzlichen Indirektionsebene). In dieser Arbeit wird das zur Laufzeit

wahrnehmbare Qualitätsmerkmal Performance betrachtet und getestet, wie die Wahl unterschiedlicher Architektur-Patterns dieses beeinflusst.

3 Web Services

3.1 Verteilte Systeme

Da es sich bei Web Services um ein verteiltes System handelt, muss zuerst erläutert werden was verteilte Systeme sind, hierfür bedienen wir uns der Definition von [StTa02]: *„A distributed system is a collection of independent computers that appears to its users as a single coherent system.“* [siehe StTa02, S. 2]

Diese Definition besitzt zwei Kernaspekte, der erste bezieht sich auf Hardware und lautet, dass es sich um autonome und unabhängige Rechner handelt, der zweite bezieht sich auf Software und lautet, dass es für den Benutzer scheint, dass er es mit einem einzigen System zu tun hat. Das heißt, dass verschiedene Systeme miteinander kommunizieren und dies vor dem Nutzer verborgen wird.

Verteilte Systeme werden oft dem *Layer* Pattern [POSA1] nach organisiert, d.h. in verschiedene Schichten gegliedert; die höhere Schicht besteht aus Benutzern und Applikationen, die niedere Schicht ist das lokale Betriebssystem und eine mittlere Schicht, genannt Middleware, ist die Schnittstelle bzw. der Mediator zwischen Applikationen und der niederen Schicht mit dem Betriebssystem [vgl. StTa02, S. 2f]. Diese Middleware ist u.a. für die Kommunikation zuständig, d.h. sie vereinfacht die Programmierung, da die low-level Programmierung von ihr gekapselt wird. Weitere Zuständigkeiten können Naming, d.h. das Auffinden von Ressourcen über Namen, die Persistenz, d.h. die Speicherung von z.B. Objekten, verteilte Transaktionen und Sicherheit sein.

Verteilte Systeme verfolgen folgende Ziele [vgl. StTa02, S. 4 – 15]:

- **Verbindung von Benutzern und Ressourcen**
  Eines der Hauptziele von verteilten Systemen ist, dass Benutzer Zugriff auf entfernte Ressourcen haben und diese auch mit anderen Benutzern teilen können (z.B. teure Hardware).

- **Transparenz**

  Für den Nutzer von Ressourcen ist es nicht ersichtlich, dass die Prozesse und Ressourcen physisch über mehrere Rechner verteilt sind, für ihn sieht es so aus, als ob er ein einziges System nutzen würde, dies ist unter Transparenz zu verstehen. Eine Übersicht der Transparenzeigenschaften wird in der folgenden Tabelle 1 dargestellt.

Transparenzeigenschaft	Beschreibung
Zugriff	Unterschiede in der Datenrepräsentation und wie auf Ressourcen zugegriffen wird, sind für den Benutzer irrelevant
Ort	Der Benutzer muss nicht den Ort, an der die Ressource liegt, kennen
Migration	Dass eine Ressource ihren Ort ändern kann, ist für den Benutzer nicht relevant
Relokation	Eine Ressource kann während der Benutzung den Ort wechseln
Replikation	Es ist für die Nutzung nicht relevant, dass eine Ressource repliziert worden sein kann oder nicht
Nebenläufigkeit	Die Ressource kann von mehreren konkurrierenden Benutzern verwendet werden
Fehler	Das Fehler- und Recoveryverhalten von Ressourcen ist für den Benutzer irrelevant (i.S. dass er sich nicht explizit darum kümmern muss)
Persistenz	In welchem Speichermedium eine Ressource existiert (z.B. Hauptspeicher, Plattenspeicher etc.) ist für den Benutzer irrelevant

Tabelle 1: Transparenzeigenschaften von verteilten Systemen

- **Offenheit**

  Ein offenes verteiltes System ist ein System, das seine Dienste nach

standardisierten Regeln für Syntax und Semantik beschreibt. Dies wird bei verteilten Systemen durch Interfaces erreicht, die Beschreibung ebendieser mit einer Interface Definition Language (IDL) bestimmt nur die Syntax (z.B. Namen der Funktionen, Parameter, Returntyp, Exceptions etc.), aber nicht deren Semantik.

Eine weitere Charakteristik der Offenheit ist die Interoperabilität, dies ist der Grad bis zu dem verschiedene Implementierungen von verteilten Systemen bzw. Komponenten zusammenarbeiten können und sich darauf verlassen können, dass sie sich verhalten wie in den entsprechenden Standards festgelegt wurde.

Als weitere Charakteristik ist die Portabilität zu nennen, diese steht dafür, wie weit eine Applikation bzw. eine Komponente, die für ein bestimmtes verteiltes System erstellt wurde, auf ein anderes übertragen werden kann.

- **Skalierbarkeit**
  Die Skalierbarkeit kann in drei Dimensionen gemessen werden, die erste und bekannteste ist die auf die Größe bzw. Kapazität bezogene, d.h. dass man mehr Ressourcen und mehr Nutzer hinzufügen kann. Die zweite bezieht sich auf die geographische Verteilung, damit ist gemeint, dass die Nutzer und Ressourcen örtlich getrennt sein können und abschließend die dritte, die administrative, d.h. dass ein System auch über getrennte Organisationen existiert.

3.2 Web Service Architektur und Protokolle

Web Services können eine Service Oriented Architecture (SOA) realisieren, welche nach [KoSta05] folgende Ziele verfolgt:

- lose Kopplung von Softwaresystemen,
- höhere Agilität von Geschäftsprozessen,

- verbesserte Wiederverwendung sowohl von Softwaresystemen als auch von Geschäftsprozessen.

Diese zuvor definierten Ziele können grundsätzlich auch mit anderen Technologien für verteilte Systeme (wie z.B. CORBA) auch erreicht werden, d.h. Web Services realisieren nicht zwingend eine Service Oriented Architecture.

Ausführlicher ist folgende Charakterisierung der Eigenschaften einer SOA aus [W3C04b]:

- **Logical View**

  Der Service ist eine abstrakte, logische Darstellung eines Programmes, einer Datenbank, eines Geschäftsprozesses usw. definiert durch das, was seine Aufgabe ist, typischerweise eine Aufgabe aus dem Bereich des Business Computings.

- **Message Orientation**

  Ein Service wird formal definiert durch den Austausch von Messages zwischen den anbietenden und den konsumierenden Diensten. Die interne Struktur eines Services z.B. die Implementierungssprache, Datenbankschemata etc. ist überwiegend irrelevant. Der Vorteil dieses Ansatzes kommt bei der Anbindung von sogenannten Legacy-Systemen (darunter sind u.a. Altsysteme aus der Host-Welt zu verstehen) zu tragen, wenn solche Systeme an neueren Applikationen angebunden werden müssen.

- **Description Orientation**

  Ein Service wird durch maschinen-verarbeitbare Metadaten beschrieben. Diese Beschreibung mittels Metadaten unterstreicht die öffentliche Natur von SOA, nur diese Details sind öffentlich und notwendig, um diesen Service aufzurufen. Die Semantik eines Services sollte direkt oder indirekt auch durch diese Metadaten beschrieben werden.

- **Granularity**

  Services stellen eine geringe Anzahl an Operationen bereit mit relativ großen und komplexen Messages.

- **Network Orientation**

  Services sind überwiegend für die Nutzung in Netzwerken vorgesehen.

- **Platform Neutral**

  Die Messages der Services werden in einem Plattform-neutralen, standardisierten Format ausgeliefert, XML ist ein Format, das diese Anforderung erfüllt.

Nachdem die allgemeinen Eigenschaften von verteilten Systemen und der Service Oriented Architecture (SOA) vorgestellt wurden, wenden wir uns den Web Services, mit denen eine SOA realisiert werden kann, zu.

Die Basistechnologie, auf der Web Services aufbauen, ist XML (eXtensible Markup Language) [W3C04a] zur Beschreibung von Metadaten und anderen damit verbundenen bzw. abgeleiteten Standards, wie z.B. XML Schema [W3C01]. Die Architektur ist wie in Abbildung 1 dargestellt eine Schichtenarchitektur die aus drei Layern besteht.

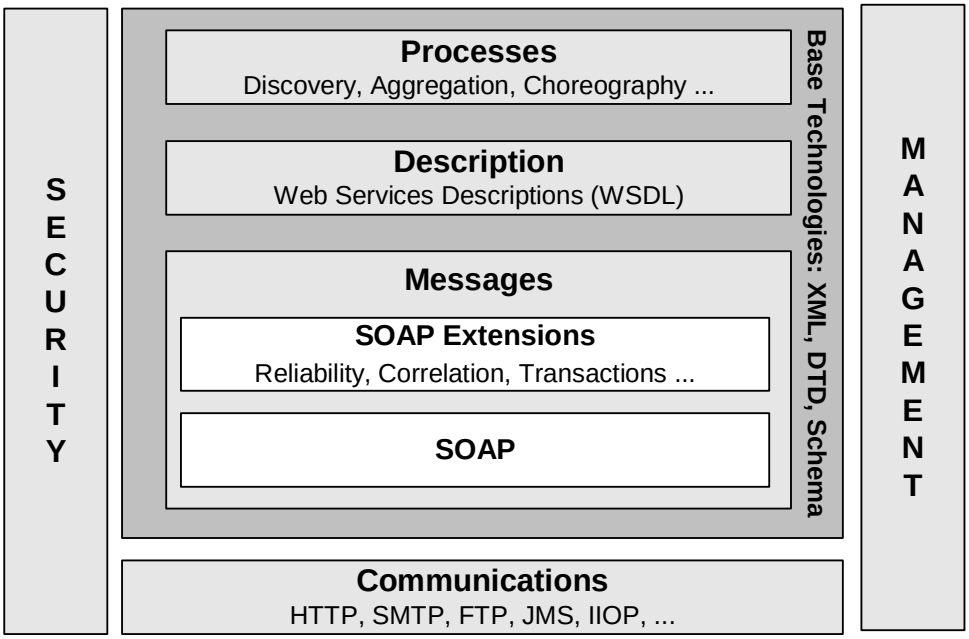


Abbildung 1: Web Services Architektur [vgl. W3C04b]

Der erste Layer, der mit *Processes* bezeichnet ist, beinhaltet u.a. Dienste wie Discovery (also das Auffinden von Web Services z.B. mittels UDDI (Universal Description Discovery Integration) [vgl. EbFi03, S. 300 – 308]) und Choreography, damit sind hauptsächlich Sprachen zur Modellierung des Workflows mittels Sprachen wie z.B. BPEL4WS gemeint [vgl. EbFi03, S. 335f].

Der mit *Description* bezeichnete Layer beinhaltet die Sprache zur Beschreibung der Schnittstellen namens WSDL (Web Service Description Language) [W3C04c]. Diese Schnittstellenbeschreibung ist notwendig, um Web Services aufrufen zu können bzw. Proxy-Objekte zu generieren.

Der letzte Layer mit der Bezeichnung *Messages* beinhaltet mit SOAP (Simple Object Access Protocol) [W3C03a] ein Protokoll für den Austausch von Messages zwischen Aufrufer und Aufgerufenen. Mittels dieses Protokolls werden die Methodenaufrufe durchgeführt, die Parameter und auch der Rückgabewert bzw. die Outputparameter (so welche gegeben bzw. der Aufruf synchron ist) übertragen.

Als Kommunikationsprotokolle stehen Web Services, die im Block *Communications* dargestellten, zur Verfügung; das Protokoll mit der häufigsten Verwendung dürfte HTTP [RFC2616] werden, da damit auch Kommunikation über ein Intranet hinaus möglich ist (d.h. es muss kein eigener Port bei einer Firewall geöffnet werden).

4 Server-Varianten

In dieser Arbeit wurde der *SimpleAxisServer* eine Variante des Axis Servers, der HTTP [RFC2616] zur Übertragung von SOAP Nachrichten [W3C03a] nutzt, modifiziert, um andere Architektur-Patterns für die konkurrente Abarbeitung von Nachrichten zu implementieren und deren Auswirkungen auf die Performance zu messen.

Server können grundsätzlich mittels eines Threads (respektive Prozesses), mittels mehrerer Threads (respektive Prozessen) und als endlicher Zustandsautomat implementiert werden [vgl. StTa02, S. 135 – 144], den ersten Ansatz bezeichnet man als iterativ, den zweiten als konkurrenten. Die dritte Variante, eine Realisierung mittels eines endlichen Zustandsautomatens, wird hier nicht weiter betrachtet.

Es wurden in dieser Arbeit drei weitere konkurrente Varianten implementiert; die ursprüngliche Version des Servers unterstützte nur den iterativen Ansatz und einen konkurrenten Ansatz, der dem *Thread-Per-Request* Pattern [PeSo97] folgend implementiert wurde.

Die folgende Tabelle stellt die implementierten Varianten nach deren Ansätzen (iterativ oder konkurrent) dar.

Iterativer Ansatz	**Konkurrenter Ansatz**
Iterativer Server *	Server dem *Thread-Per-Request* Pattern [PeSo97] folgend *
	Server mit Thread Pooling[2]
	Server dem *Half-Sync/Half-Async* Pattern [POSA2] folgend
	Server dem *Leader/Followers* Pattern [POSA2] folgend

Tabelle 2: Implementierte Ansätze für Server[3]

Wenn Architektur-Patterns präsentiert werden, orientiert sich die Darstellung an der in [POSA1, S. 9ff] vorgestellten. Zusätzlich wird noch der Konsequenzen Teil dargestellt, wie in [POSA1, S. 21] und in [GHJV96, S. 4] beschrieben.

Es folgt nun eine Erläuterung der Teile der Pattern Beschreibung:

- **Kontext**

  Der Kontext der beschreibt in welchen Situationen das durch das Pattern zu lösende Problem auftauchen kann. Da es nahezu unmöglich ist alle Situationen anzugeben, beschränkt man sich nur auf die bekannten, die von einem Pattern adressiert werden.

- **Problem**

  Welches konkrete Problem ist durch dieses Pattern zu lösen, die Darstellung des Problems wird durch Angabe der Forces (Kräfte) komplettiert.

  Die Forces sind Aspekte, die bei der Lösung des Problems zu berücksichtigen sind, wie zum Beispiel:

  - Requirements, die die Lösung zu erfüllen hat.
  - Constraints, die zu berücksichtigen sind, wie zum Beispiel das Einhalten eines bestimmten Protokolls.
  - Erwünschte Eigenschaften der Lösung, wie zum Beispiel die leichte Änderbarkeit (Changeability) der Software.

  Diese Forces können komplementär wie auch konfliktär sein, dadurch müssen diese ausbalanciert werden. Bei Systemen mit eingeschränkt zur Verfügung gestellten Ressourcen (wie Embedded Systems) kann die Kraft Erweiterbarkeit (z.B. durch Einsatz der Objekttechnologie) im Gegensatz zur Schonung der Ressourcen wirken.

---

[2] Realisiert das *Pooling* Pattern [POSA3] für Threads

- **Lösung**

  Der Solution Teil zeigt wie das Problem zu lösen ist oder genauer wie die Forces zu balancieren sind.

  Es werden die statischen Aspekte der Architektur (welche Komponenten daran teilnehmen) wie auch die dynamischen Aspekte (wie interagieren die Komponenten untereinander) beschrieben.

- **Konsequenzen**

  Beschreibt die Vorteile wie auch Nachteile, die die Lösung mittels dieses Patterns mit sich bringt. Die Konsequenzen beeinflussen wiederum die Forces. Zum Beispiel, dass ein mehr an Speicherplatz (z.B. beim Caching) belegt wird (Nachteil), aber dafür die Performance (Vorteil) besser ist.

Anhand der konkreten Implementierung des *SimpleAxisServer*s wird dargestellt wie diese Architektur-Patterns in einer Applikation realisiert werden können. Es wurde versucht, eine möglichst einfache Implementierung ohne Mikrooptimierungen vorzunehmen. um eine gute Vergleichbarkeit zu erreichen. Des Weiteren soll dadurch erreicht werden, dass die Implementierung exemplarisch die eigentliche Grundidee des zugrunde liegenden Patterns reflektiert.

4.1 Iterativer Server

Der iterative Server ist die einfachste Variante der Realisierung eines Servers. Es findet keine konkurrente Abarbeitung von Requests statt, es wird jeder Request sequenziell abgearbeitet.

[3] Mit * markierte Implementierungen sind in der ursprünglichen Version des Servers schon implementiert worden

Das folgende Code-Beispiel der Klasse `SimpleAxisServer` stellt die Realisierung eines iterativen Servers dar und ist die Basis aller anderen Server-Varianten.

```
.
.
package org.apache.axis.transport.http;
.
.
public class SimpleAxisServer implements Runnable {
     .
     .
     public void run() {
          .
          .
          // Accept and process requests from the socket
          while (!stopped) {
               Socket socket = null;
               try {
                    socket = serverSocket.accept();
               } catch (java.io.InterruptedIOException iie) {
               } catch (Exception e) {
                    log.debug(Messages.getMessage("exception00"), e);
                    break;
               }
               if (socket != null) {
                    SimpleAxisWorker worker = new SimpleAxisWorker(this, socket);
                    if (doThreads) {
                         .
                         .
                    } else {
                         worker.run();
                    }
               }
          }
          .
          .
     }
     .
     .
}
```

Abbildung 2: *SimpleAxisServer* des iterativen Servers

Die Schleife `while (!stopped)` stellt den Kern des Servers dar, in dieser werden die Verbindungen akzeptiert (in der Zeile `socket = serverSocket.accept();`).

Das Objekt `socket` stellt die Verbindung mit dem Client dar, es wird dann in der Zeile `SimpleAxisWorker worker = new SimpleAxisWorker(this, socket);` ein Objekt `worker` instanziert, das die Verarbeitung des Requests durchführt.

Wenn die boole'sche Variable `doThreads` auf `false` gesetzt wurde, d.h. der Server wurde für die iterative Verarbeitung konfiguriert (mittels eines Kommandozeilenparameters), wird die Verarbeitung des Requests mittels `worker.run()` angestoßen.

Wie anhand der beschriebenen Implementierung zu sehen ist, besitzt ein iterativer Server den Vorteil, dass dieser leicht zu implementieren ist. Es müssen keine Synchronisationsmechanismen verwendet werden wie bei den konkurrenten Varianten, welches eine zusätzliche Fehlermöglichkeit eliminiert.

Nachteilig erweist sich, dass auf Grund der Nicht-Verwendung von Threads die Performance geringer als bei konkurrenten Varianten sein kann (u.a. werden Multi-Prozessor Systeme nicht ausgenutzt; der Server blockiert solange ein Request nicht beendet ist etc.).

4.2 Server dem Thread-Per-Request Pattern folgend

4.2.1 Das Thread-Per-Request Pattern

Die Beschreibung des *Thread-Per-Request* Pattern orientiert sich an der in [PeSo97] erschienen, die Erläuterungen zu diesem Pattern wurde mit Elementen aus dem *Forking Server* Pattern, wie in [GrTa03] dargestellt, ergänzt.

Wie aus den Namen der Pattern hervorgeht, ist der Hauptunterschied zwischen den beiden Pattern, dass das erste Threads kreiert und das zweite Prozesse (zum Unterschied zwischen Threads und Prozesse siehe Standardwerke über Betriebssysteme wie z.B. [Tan02]).

Kontext

Ein Server muss komplexe Requests von mehreren Clients abarbeiten. Die Abarbeitung eines Requests bedarf einer längeren Zeitdauer.

Problem

Um die Kapazität eines Servers zu steigern, will man Multi-threading einsetzen.

Forces:

- Eine einfach zu realisierende Lösung ohne zuviel Overhead für das Thread Management. Obwohl jeder Thread die Abarbeitung eines einzelnen Requests übernimmt, benötigen die Threads Synchronisationsmechanismen, um auf gemeinsam genutzte Daten zuzugreifen, dies stellt eine erhöhte Komplexität dar. Diese erhöhte Komplexität kann dazu führen, dass die Performance-Vorteile der konkurrenten Abarbeitung durch diesen Overhead aufgefressen werden.

- Die Erhöhung der Anzahl von Threads führt zu einen Mehrverbrauch an System-Ressourcen (Hauptspeicher, File-Deskriptoren etc.) und zu einem Overhead bei der Ausführung (Thead-Erzeugung, Kritische Regionen (dieser Synchronisationsmechanismus wird beispielsweise in [Tan02, S. 119] erläutert) etc.).

- Die Allokation von Ressourcen (Threads, Hauptspeicher etc.) soll nur wenn notwendig durchgeführt werden, die Deallokation sollte so früh wie nur möglich durchgeführt werden.

Lösung

Wie Abbildung 3 illustriert besteht die Lösung darin, dass ein Prozess bzw. Thread der *Listener* (auch als *Dispatcher* [vgl. Tan02, S. 103] bekannt) genannt wird die Requests entgegennimmt (d.h. die Verbindung akzeptiert) und für die Abarbeitung *Worker*-Threads [vgl. Tan02, S. 103] kreiert, die die eigentliche Verarbeitung durchführen.

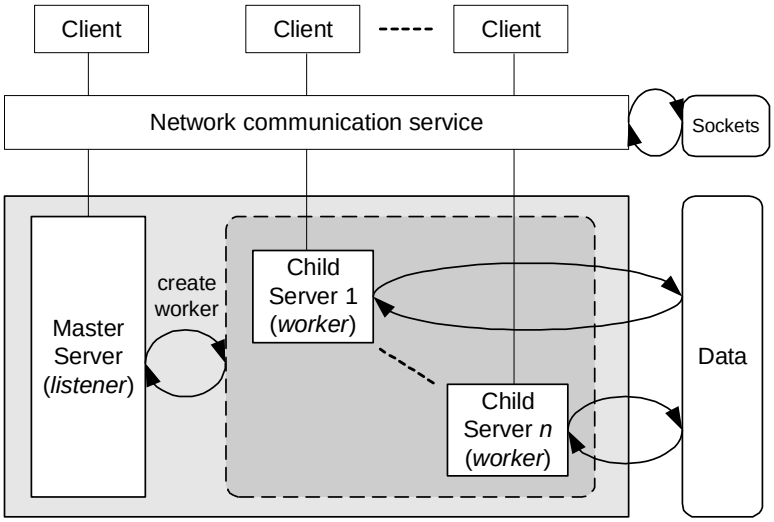


Abbildung 3: Forking Server [vgl. GrTa03, S. 9]

Nachdem die Abarbeitung des Requests durchgeführt wurde, wird der Thread beendet. Der Programmieraufwand zur Realisierung dieser Lösung ist gering und System-Ressourcen werden nur für die Dauer der Abarbeitung eines Requests belegt.

Konsequenzen

Vorteile:

- *Performance* - Bei langandauernden Requests wird eine bessere Performance erreicht, d.h. dass ungenutzte CPU-Zeit nicht brachliegt (z.B. während ein Thread auf ein Event wartet, kann die CPU-Zeit von den anderen Threads genutzt werden kann).
- *Ressourcen* - Wenn der Server nicht bzw. unter niedriger Last steht, werden wenig Ressourcen belegt, da Threads nur on-demand erzeugt werden.

Nachteile:

- *Kein Threadlimit (1)* - Da es keine Begrenzung der Threads gibt, kann das System unerwünschterweise ausgelastet werden.

- *Kein Threadlimit (2)* - Erhöhung der Anzahl von Threads über eine bestimmte Anzahl ist auf Grund von abnehmenden Grenzerträgen nicht sinnvoll (d.h. der Overhead durch die Allokation und Deallokation von System Ressourcen und der Erzeugung von Threads größer als der Performance Gewinn durch die konkurrente Verarbeitung ist).

- *Overhead* - Threads werden immer neu gestartet und nicht wieder verwendet, d.h. es ergibt sich ein Erzeugungs-Overhead.

- *Komplexität* - Implementierung ist komplexer als bei iterativen Server, Fehlermöglichkeiten wegen Synchronisation.

4.2.2 Implementierungsdetails

```
.
.
package org.apache.axis.transport.http;
.
.
public class SimpleAxisServer implements Runnable {
     .
     .
     public void run() {
          .
          .
          // Accept and process requests from the socket
          while (!stopped) {
               Socket socket = null;
               try {
                    socket = serverSocket.accept();
               } catch (java.io.InterruptedIOException iie) {
               } catch (Exception e) {
                    log.debug(Messages.getMessage("exception00"), e);
                    break;
               }
               if (socket != null) {
                    SimpleAxisWorker worker = new SimpleAxisWorker(this, socket);
                    if (doThreads) {
                         Thread thread = new Thread(worker);
                         thread.setDaemon(true);
                         thread.start();
                    } else {
                         .
                         .
                    }
               }
          }
          .
          .
     }
     .
     .
}
```

Abbildung 4: *SimpleAxisServer* des dem *Thread-Per-Request* Pattern folgenden Servers

Der einzige Unterschied in der Implementierung zu der des iterativen Servers ergibt sich daraus, dass die Variable `doThreads` auf `true` gesetzt wurde und im entsprechenden Block ein `Thread`-Objekt für jeden Request instanziert wird,

das die Methode `run` des `worker`-Objekts, das auch für diesen Request neu erzeugt wurde, aufruft. Die Methode `run` des `worker`-Objekts arbeitet dann als eigener Thread den Request ab.

Die untenstehende Graphik verdeutlicht den oben genannten Zusammenhang, dass für einen jeden Request ein `Thread`-Objekt instanziert wird, das genau eine Instanz eines `SimpleAxisWorker` als Referenz erhält.

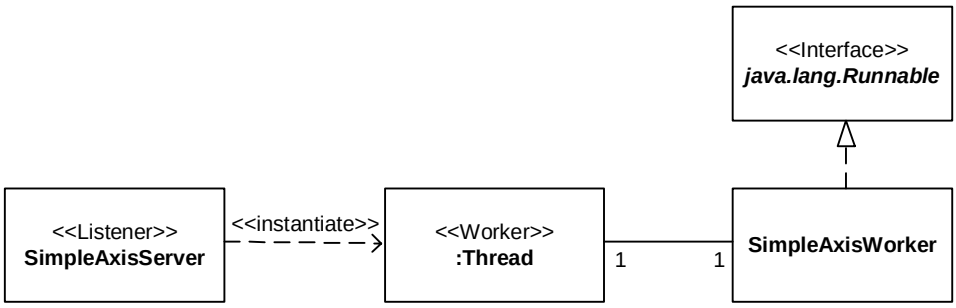


Abbildung 5: Klassendiagramm *SimpleAxisServer* nach dem *Thread-Per-Request* Pattern

4.3 Server mit Thread Pooling

4.3.1 Das Pooling Pattern

Die Basis der Beschreibung des *Pooling* Pattern, stellt die in [POSA3] publizierte Darstellung des Patterns dar.

Kontext

Systeme, die kontinuierlich Ressourcen belegen und freigeben, die dies aber effizient durchführen müssen.

Problem

Für viele Systeme müssen Ressourcen schnell und vorhersagbar (im Sinne der Zugriffszeit) verfügbar sein. Typische solche Ressourcen wären Netzwerk-Verbindungen, Threads, Speicher und andere Objekte. Der Zugriff auf solche Ressourcen soll über die Anzahl der zu verwaltenden Ressourcen wie auch der darauf zugreifenden Clients skalieren. Die Abweichung der Zugriffszeiten für unterschiedliche Clients soll so klein wie nur möglich sein.

Forces:

- *Performance* – Die Verschwendung von CPU-Zeit für den Zugriff auf Ressourcen soll minimiert werden
- *Vorhersagbarkeit* – Die Zugriffszeit auf Ressourcen soll vorhersagbar sein.
- *Einfachheit* – Die Lösung sollte einfach sein, um die Komplexität der Applikation minimal zu halten.
- *Stabilität* – Die Akquisition und die Freigabe von Ressourcen kann das Risiko von System Instabilitäten erhöhen. Beispielsweise kann die Allokation und Deallokation von Hauptspeicher zur Fragmentierung ebendieses führen (bekannt unter dem Begriff Heap Fragmentation).
- *Reuse* – Nicht-benutzte Ressourcen sollten wiederverwendet werden, d.h. Akquisition (im Sinne einer Instanzierung bzw. Allokation) sollte nur einmal stattfinden.

Lösung

Das *Pooling* Pattern besitzt folgende Kollaborateure:

- Einen *Resource User* der Ressourcen akquiriert und verwendet.
- Die *Resource*, die Ressource z.B. Speicher oder ein Thread.
- Der *Resource Pool* verwaltet Ressourcen und stellt sie den *Resource User*n zur Verfügung.
- Die *Resource Environment*, wie zum Beispiel das Betriebssystem, welches die Ressourcen ursprünglich verwaltet und besitzt.

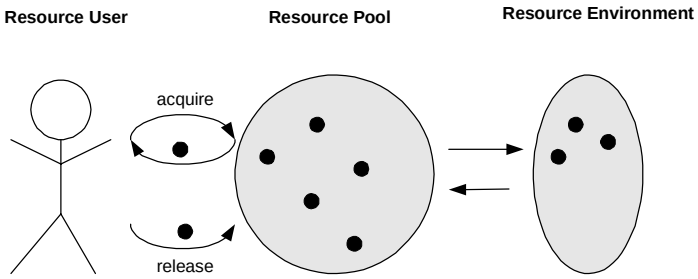


Abbildung 6: Interaktionen beim *Pooling* Pattern [vgl. POSA3]

Abbildung 6 verdeutlicht die Beziehung zwischen den einzelnen Teilnehmern am *Pooling* Pattern. Der *Resource Pool* entnimmt und verwaltet *Resource*s aus der *Resource Environment* und stellt diese den *Resource User*n zur Verfügung.

Die *Resource User* können *Resource*s akquirieren und wieder freigegeben (besser gesagt sie müssen die *Resource*s wieder freigeben, damit kein Resource Leak entsteht, außer das *Leasing* Pattern [POSA3] wird verwendet, um dieses mögliche Problem zu adressieren).

Die Idee des *Leasing* Patterns besteht darin, dass eine Ressource für eine bestimmte Zeitspanne an einen *Resource User* vergeben wird. Dieser kann die Ressource nutzen, optional das Lease erneuern (d.h. erneut eine Zeitspanne anfordern). Ist die Zeitspanne abgelaufen, so kann die Ressource automatisch dem *Resource User* entzogen werden und sie wird wieder freigegeben bzw. einem Pool zurückgegeben.

Die Interaktion zwischen den Kollaborateuren variiert, abhängig davon, ob Ressourcen nach dem *Eager Acquisition* Pattern [POSA3] während des Start-ups akquiriert wurden oder nicht. Wurden Ressourcen während des Start-ups (für den Pool) akquiriert, werden die ersten Akquisitionen der *Resource User* durch diese befriedigt. Darüber hinaus benötigte Ressourcen werden on-demand (dem *Lazy Acquisition* Pattern [POSA3] folgend) akquiriert (für den Pool).

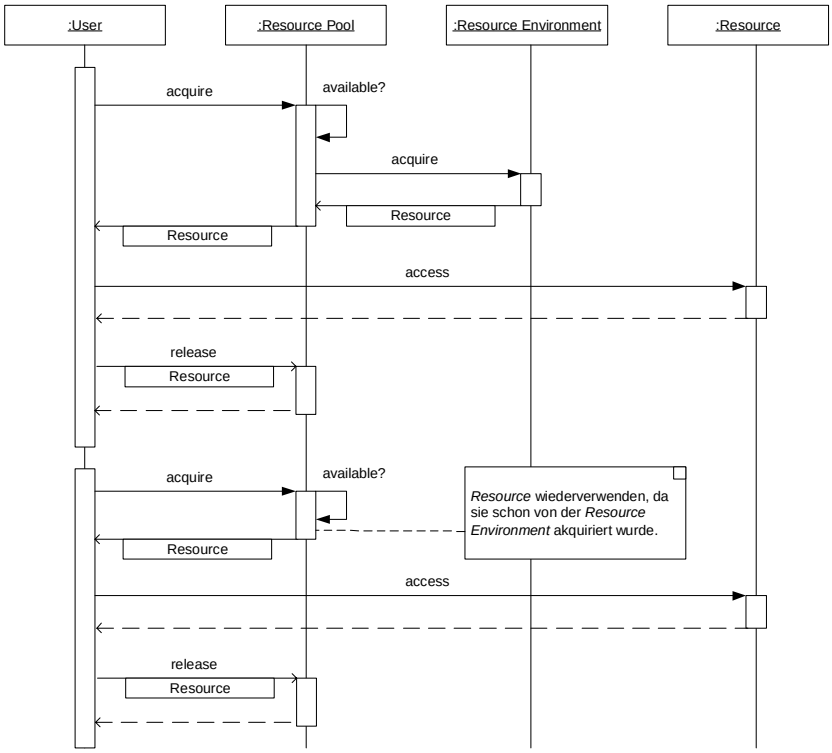


Abbildung 7: Sequenzdiagramm für das *Pooling* Pattern [vgl. POSA3]

Das Sequenzdiagramm in Abbildung 7 stellt einen typischen *acquire*/*release* Zyklus dar, nach einem *acquire* wird im Pool geprüft, ob eine Ressource vorrätig ist, falls nicht, wird sie aus der *Resource Environment* akquiriert; ist sie vorrätig, so wird sie wieder verwendet.

Ein *Resource Pool* kann noch zusätzlich statistische Daten, wie z.B. letztmalige Verwendung einer Ressource, Häufigkeit der Verwendung etc., führen, um beispielsweise die Verwendung des *Evictor* oder *Leasing* Patterns (beide aus [POSA3]) zu ermöglichen.

Konsequenzen

Vorteile:

- *Performance* – Der Aufwand für Akquisition und Freigabe von Ressourcen wird reduziert.
- *Vorhersagbarkeit* – Das Lookup und die Akquisition von schon im Pool enthaltenen Ressourcen ist vorhersagbarer als bei einer Akquisition aus dem Umfeld der Ressource, insbesonders wenn *Eager Acquisition*

[POSA3] angewandt wird. D.h. eine bestimmte Anzahl an knappen Ressourcen wird bei der Initialisierung des Pools akquiriert und nicht on-demand. Die on-demand Akquisition von Ressourcen, die größer als die initiale Anzahl ist, folgt dem *Lazy Acquisition* Pattern [POSA3].

- *Sharing* – Ressourcen können von verschiedenen Nutzern verwendet werden. Beispielsweise stellt ein Datenbank-Verbindungspool eines Application Servers seine Dienste allen Applikationen, die in diesem Server residieren, zur Verfügung.

Nachteile:

- *Memory Footprint* – Mehr Speicher als benötigt kann belegt sein. Dies kann durch Anwendung des *Evictor* Patterns (ein Pattern, das nicht bzw. selten genutzte Ressourcen aus einem Pool entfernt), *Leasing* Patterns und / oder *Lazy Acquisition* Pattern (alle genannten Patterns stammen aus [POSA3]) minimiert werden.

- *Overhead* – Es wird zusätzliche CPU-Zeit für das Management der Ressourcen im Pool aufgewandt. Dieser Overhead ist typischerweise geringer als für die Akquisation und Freigabe der Ressourcen außerhalb des Pools von Nöten wäre.

- *Komplexität* – Clients, die Ressourcen nutzen, müssen diese wieder in den Pool zurückgeben. Mittels des *Leasing* Patterns [POSA3] kann dieses Problem gemindert werden.

- *Synchronisation* – Applikationen, die Multi-threading nutzen, müssen Synchronisationsmechanismen verwenden, um Race Conditions (eine Erläuterung zu Race Conditions findet sich in [Tan02, S. 117f]) zu vermeiden.

4.3.2 Implementierungsdetails

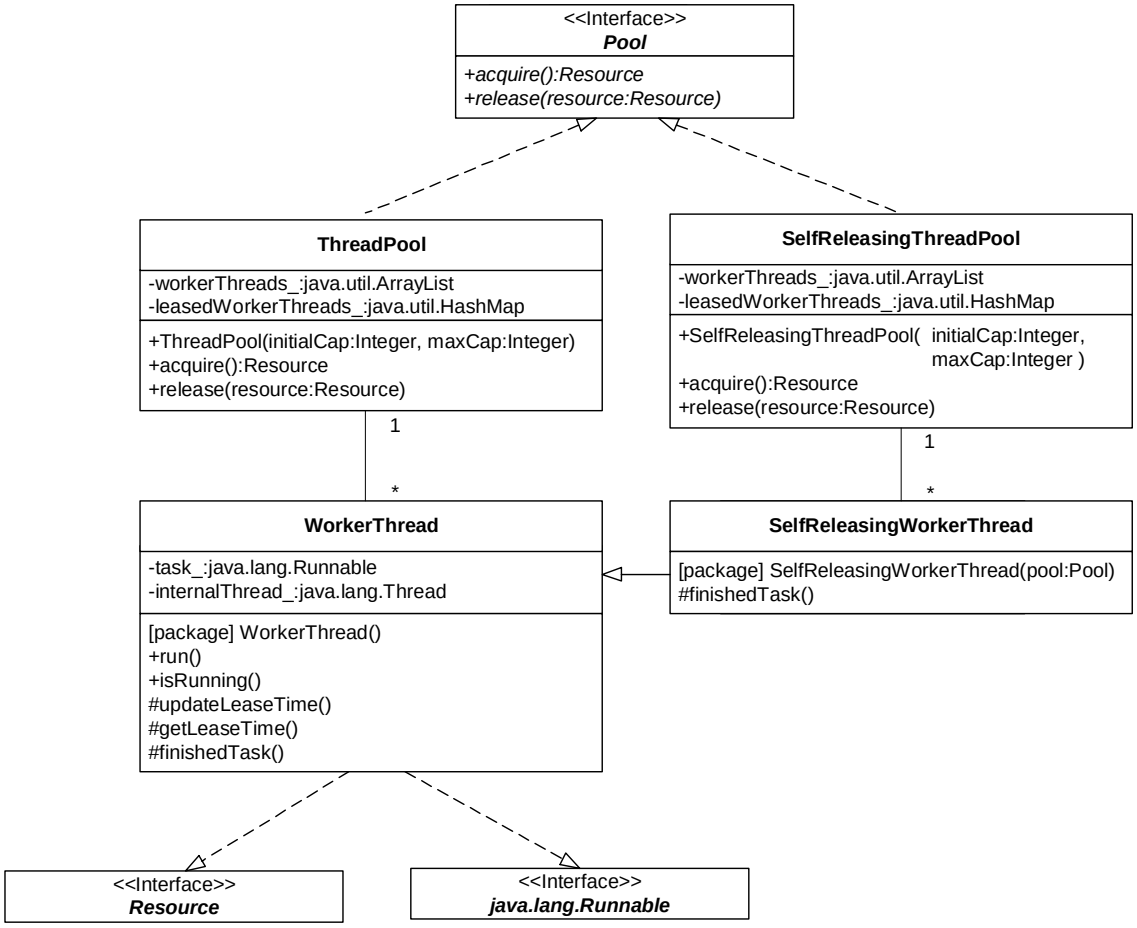


Abbildung 8: Thread Pool Klassen im Paket da_pattern_axis.threads

Es gibt zwei Klassen, die das `Pool` Interface implementieren, nämlich `ThreadPool` und `SelfReleasingThreadPool`. Erstere Klasse realisiert einen Thread Pool der `WorkerThread`-Objekte verwaltet (die mittels `release` in den Pool zurückgegeben werden müssen), der `SelfReleasingThreadPool` verwaltet `SelfReleasingWorkerThread`-Objekte, diese Objekte rufen die `release` Methode des Pools in `SelfReleasingWorkerThread.finishedTask` automatisch, nachdem die `run`-Methode von `task_` aufgerufen wurde, auf.

Der garantierte Aufruf von `finishedTask` auch im Falle einer auftretenden Exception wird mittels des *try-finally* Idioms [vgl. Bloc02, S. 33 - 37] sichergestellt.

Die Konstruktoren der Thread Pool Klassen besitzen die Formalparameter `initCap` und `maxCap`, `initCap` gibt an, wie viele Threads der Pool bei Start-up vorhalten soll (wenn größer 0 findet *Eager Acquisition* [POSA3] statt), `maxCap` gibt die Maximalanzahl der zu verwaltenden Threads an.

Wenn keine Threads vorrätig sind, werden Threads bis zur Maximalanzahl instanziert (wenn die Maximalanzahl erreicht wurde, muss der Client auf verfügbare Threads warten), dies stellt eine Form von *Lazy Acquisition* [POSA3] dar.

Bei einer jeden Akquisition ruft der Pool `updateLeaseTime` auf, um den Zeitpunkt der Vergabe der Ressource festzuhalten.

Die von einem Thread aus dem Pool auszuführende Aufgabe muss ein Objekt sein, dass das Interface `Runnable` implementiert. Die auszuführende Aufgabe wird mittels `setTask` gesetzt, wobei der Parameter `task_` ein *Command*-Objekt ist [Vgl. das gleichnamige Pattern in GHJV96].

Die unten folgende Abbildung des *SimpleAxisServer*s illustriert den konkreten Einsatz des `SelfReleasingThreadPool` Objekts.

```
.
.
package org.apache.axis.transport.http;
.
.
// Importing the thread pool
import da_pattern_axis.threads.SelfReleasingThreadPool;
import da_pattern_axis.threads.SelfReleasingWorkerThread;
.
.
public class SimpleAxisServer implements Runnable {

     /**
      * Initial capacity of thread pool
      **/
     private static int INIT_CAP = 33;

     /**
      * Maximum capacity of thread pool
      **/
     private static int MAX_CAP = 100;

     /**
      * The thread pool
```

```
     **/
    private SelfReleasingThreadPool pool_ = null;
    .
    .
    public void run() {
         .
         .
         // Setting up thread pool
         pool_ = new SelfReleasingThreadPool( INIT_CAP, MAX_CAP );

         // Accept and process requests from the socket
         while (!stopped) {
              Socket socket = null;
              try {
                   socket = serverSocket.accept();
              } catch (java.io.InterruptedIOException iie) {
              } catch (Exception e) {
                   log.debug(Messages.getMessage("exception00"), e);
                   break;
              }
              if (socket != null) {
                   SimpleAxisWorker worker = new SimpleAxisWorker(this, socket);
                   if (doThreads) {
                        .
                        .
                        SelfReleasingWorkerThread workerThread = \
                        (SelfReleasingWorkerThread) pool_.acquire();
                        workerThread.setTask( worker );
                        workerThread.start();

                   } else {
                        .
                        .
                   }
              }
         }
         .
         .
    }
    .
    .
}
```

Abbildung 9: *SimpleAxisServer* des Servers mit Thread Pooling

In dieser Variante des *SimpleAxisServer*s besitzt die Klasse `SimpleAxisServer` die Klassenvariablen `INIT_CAP` und `MAX_CAP`, die mittels Kommandozeilenparameter gesetzt werden können (so nicht die Default-Belegung genutzt werden soll). Diese Klassenvariablen dienen als Argumente für den Konstruktor des `SelfReleasingThreadPool`-Objekts, das als Thread

Pool verwendet wird. Der Pool wird in der `run` Methode mittels der Zeile `pool_ = new SelfReleasingThreadPool( INIT_CAP, MAX_CAP );` instanziert.

Worker Threads aus diesem Pool werden in der Schleife, die die Verbindungen akzeptiert mittels der Zeile `SelfReleasingWorkerThread workerThread = (SelfReleasingWorkerThread) pool_.acquire();` aus dem Pool entnommen.

Als auszuführendes *Command*-Objekt wird das `SimpleAxisWorker`-Objekt `worker` gesetzt, dies geschieht in der Zeile `workerThread.setTask( worker );`. Die konkurrente Ausführung des Threads wird mittels `workerThread.start();` angestoßen.

4.4 Server dem Half-Sync/Half-Async Pattern folgend

4.4.1 Das Half-Sync/Half-Async Pattern

Die folgende Darstellung des *Half-Sync/Half-Async* Pattern basiert auf der in [POSA2] publizierten.

Kontext

Ein konkurrentes System, welches synchrone wie auch asynchrone Dienste besitzt, die miteinander kommunizieren müssen.

Problem

Konkurrente Systeme sind oft eine Kombination aus synchroner wie auch asynchroner Verarbeitung.

Es ist für Entwickler oft interessant, asynchrone Verarbeitungsmodelle aus Performance Gründen zu verwenden, da Dienste oft von asynchronen Mechanismen, wie z.B. Interrupts (beispielsweise durch I/O Events ausgelöst) oder Signalen abhängen.

Das synchrone Verarbeitungsmodell bietet den Vorteil der einfachen Handhabung, da es für gewöhnlich weniger komplex in der Realisierung ist.

Forces:

- Die Architektur soll so designt sein, dass Anwendungs-Entwickler, die die Einfachheit des synchronen Verarbeitungsmodells nutzen wollen, nicht mit der Komplexität des asynchronen konfrontiert werden. Umgekehrt sollen System-Entwickler, die die Performance maximieren wollen, nicht mit den Ineffizienzen des synchronen Verarbeitungsmodells konfrontiert werden.
- Die synchronen und asynchronen Verarbeitungsdienste sollten, ohne deren Programmierung unnötig zu verkomplizieren oder die Performance zu reduzieren, miteinander kommunizieren können.

Lösung

Die Dienste des Systems werden in zwei Schichten (Layers), einer synchronen und einer asynchronen Schicht gegliedert. Diese zwei Schichten kommunizieren über eine dritte, der Queueing-Schicht, miteinander. Dieses Architekturkonzept entspricht dem *Layer* Pattern aus [POSA1].

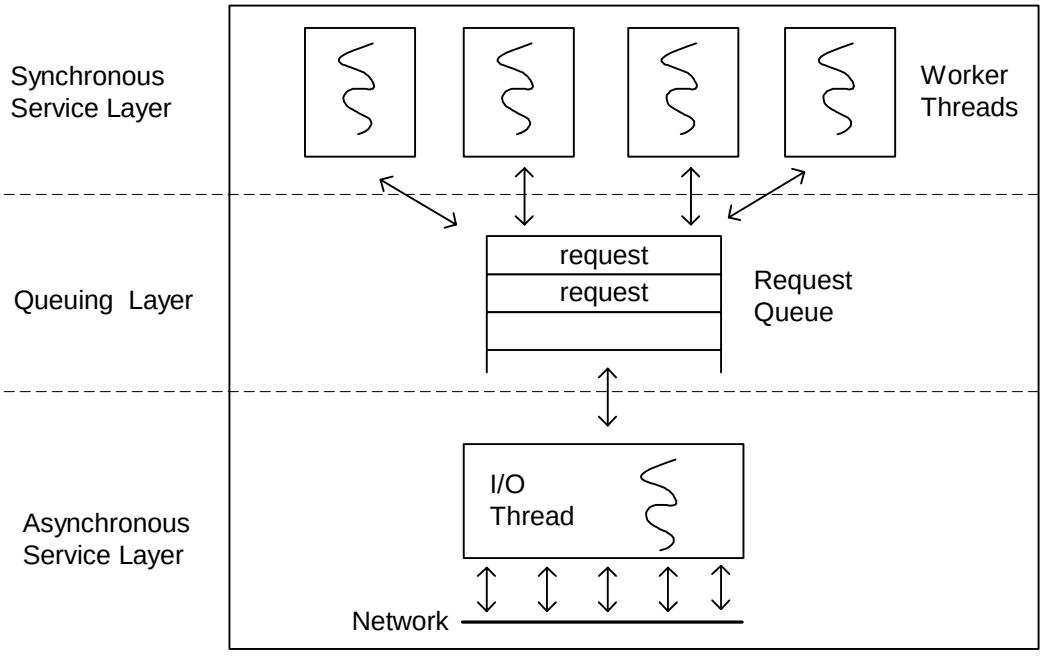


Abbildung 10: Layer des *Half-Sync/Half-Async* Pattern [vgl. POSA2, S. 448]

Langandauernde, in der Abstraktion höhere Dienste, wie zum Beispiel Datenbank-Abfragen oder Datei-Transfers, werden in der synchronen Schicht mittels eigener Threads (*Worker Threads* in Abbildung 10) oder Prozesse durchgeführt.

Kurzandauernde, in der Abstraktion vom System niedere Dienste, wie zum Beispiel Handler, die durch Interrupts ausgelöst werden, werden in der asynchronen Schicht angesiedelt, um eine höhere Performance zu erreichen.

Da diese beiden Schichten miteinander kommunizieren müssen, wird zusätzlich eine Queueing-Schicht zu diesem Zwecke eingezogen (*Request Queue* in Abbildung 10). Diese Queuing-Schicht ist ein *Mediator* zwischen den beiden anderen Schichten [vgl. das gleichnamige Design Pattern in GHJV96].

Konsequenzen

Vorteile:

- *Einfachheit und Performance* – Die Programmierung von high-level synchronen Diensten ist vereinfacht, ohne die Performance von low-level asynchronen System Diensten zu beinträchtigen.
- *Separation of Concerns* – Die Strategie, ob eine synchrone oder asynchrone Verarbeitung stattfindet, ist für jede Schicht getrennt.

Nachteile:

- *Komplexität des Debuggens und des Testens* - Durch die Trennung von Ort und Zeit zwischen dem Aufruf einer Operation und deren Abschluss (sprich der Notifikation, z.B. durch Aufruf eines Callbacks bzw. durch Anwendung des *Asynchronous Completion Token* Patterns [POSA2]) beim asynchronen Verarbeitungsmodell [vgl. POSA2, S. 258] ist das Debuggen und Testen erschwert.

- *Performance Overhead* – Es entsteht durch die Queueing-Schicht ein Overhead (Context Switches, Entnahme/Einfügen der Elemente in die Queue etc.).

4.4.2 Implementierungsdetails

Die Beschreibung der Implementierung beginnt mit der Darstellung der Queueing-Schicht. Zum besseren Verständnis der Implementierung werden die Forces des *FIFO Queue* Patterns [vgl. Herz03, S. 2] vorgestellt:

- *Zwischenspeicherung* – Da die Nachrichten nicht sofort verarbeitet werden können, müssen sie gespeichert werden.

- *Anzahl an Producer und Consumer* – Es können mehrere Producer und Consumer existieren.

  Producer legen Nachrichten in der Queue ab, die Consumer entnehmen Nachrichten der Queue und arbeiten diese ab.

- *Over- / Underflow* – Die Produktion von Nachrichten kann zu schnell oder zu langsam für die Consumer sein.

- *Concurrency* – Synchronisation für den Zugriff auf die Nachrichten in der Queue muss sichergestellt werden.

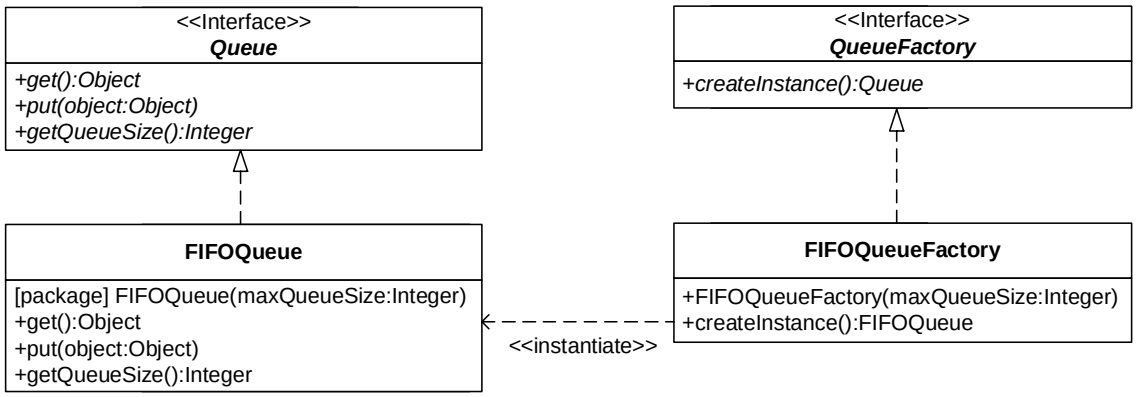


Abbildung 11: Queue Klassen im Paket da_pattern_axis.container

Für Queues im Allgemeinen wurde das Interface `Queue` definiert, das die Methode `get` zum Entnehmen und `put` zum Ablegen von Elementen in eine Queue vorsieht. Des Weiteren wurde die Methode `getQueueSize`, die die Anzahl der Elemente in der Queue zurückliefert, eingeführt.

Da in dieser Implementierung eine FIFO-Queue verwendet wird, existiert eine Klasse `FIFOQueue`, die ein `Queue`-Interface implementiert. Eine FIFO-Queue ist eine Queue, die dem First-In-First-Out (FIFO) Prinzip folgend Elemente zurückliefert, d.h. das Element das zuerst abgelegt wurde (mittels `put`), wird auch als Erstes (mittels `get`) zurückgeliefert und aus der Queue entfernt.

Damit die Queue nicht unendlich viele (oder besser bis zur Maximalgröße des verwendeten Containers als Zwischenspeicher) Elemente aufnimmt, besitzt die `FIFOQueue` Klasse einen Konstruktor mit dem Formalparameter `maxQueueSize`, der die Maximalgröße angibt.

Wenn `put` von einem Producer aufgerufen wird und die Queue voll ist (d.h. die Anzahl der Elemente entspricht der Maximalgröße), blockiert der Aufrufer bis die Anzahl der Elemente wieder kleiner der Maximalgröße ist. Dieses Vorgehen löst die *Overflow*-Problematik, die bei den Forces angegeben wurde.

Die *Underflow*-Problematik wird dadurch aufgelöst, dass, wenn `get` aufgerufen wird und keine Elemente vorhanden sind, der Consumer blockiert bis wieder Elemente in der Queue abgelegt worden sind.

Um Race Conditions zu vermeiden, sind alle Methoden mittels eines Monitor Objekts synchronisiert. Damit wird die *Concurrency* und auch die *Anzahl an Producer und Consumer*-Problematik adressiert.

Zur *Zwischenspeicherung* der Elemente wird intern eine verkettete Liste, wie auch in [Herz03, S. 4] dargestellt, verwendet. Diese Liste wird von der Laufzeitumgebung der Java Runtime bereitgestellt (`java.util.LinkedList`). Alternativ könnte auch eine Implementierung als Array vorgenommen werden [vgl. Herz03, S. 4f]. Eine weitere Möglichkeit zur Implementierung einer Queue wäre die einer Priority Queue, welche Elemente nach ihrer Priorität ordnet.

Um zukünftig eine Wahl der zu verwendenden Queue Art zur Laufzeit zu ermöglichen (und somit eine Anwendung des *Strategy* Patterns [GHJV96] zu realisieren) wird das *Abstract Factory* Pattern aus [GHJV96] verwendet. Dieses Pattern wird mittels des Interfaces `QueueFactory` und der das Interface implementierenden Klasse `FIFOQueueFactory` realisiert. Derzeit wird immer nur eine `FIFOQueue` via einer *Factory* erzeugt und kein *Strategy* Pattern implementiert.

Am Beispiel von Code werden in den folgenden Absätzen und Abbildungen die zwei anderen Schichten beschrieben. Die asynchrone Schicht wird mittels der Klasse `SimpleAxisServer` realisiert und die synchrone mittels der Klasse `SimpleAxisWorkerDecorator`.

Es folgt nun der Code-Ausschnitt des *SimpleAxisServer*, der dem *Half-Sync/Half-Async* Pattern folgend modifiziert wurde.

```
.
.
package org.apache.axis.transport.http;
.
.
// Importing classes for the queue
import da_pattern_axis.container.Queue;
import da_pattern_axis.container.FIFOQueueFactory;.
.
.
public class SimpleAxisServer implements Runnable {

     /**
      * Number of threads processing requests from the queue
      **/
     private static int THREAD_COUNT = 100;

     /**
      * Maximum size of the queue
      **/
     private static int MAX_QUEUE_SIZE = 100;

     /**
      * Array of threads with the size THREAD_COUNT which
      * process requests from the queue
      **/
     private Thread[] workers_ = null;

     /**
      * The queue
      **/
```

```
    private Queue queue_ = null;
    .
    .
    public void run() {
        .
        .
        // Setting up queue
        queue_ = ( new FIFOQueueFactory( MAX_QUEUE_SIZE ) ).createInstance();

        // Setting up worker threads which process requests from the queue
        workers_ = new Thread[ THREAD_COUNT ];
        for ( int i = 0; i < THREAD_COUNT; i++ ) {
            workers_[i] = new Thread(
                                new SimpleAxisWorkerDecorator( this, queue_ )
                                );
            workers_[i].start();
        }

        // Accept and process requests from the socket
        while (!stopped) {
            Socket socket = null;
            try {
                socket = serverSocket.accept();
            } catch (java.io.InterruptedIOException iie) {
            } catch (Exception e) {
                log.debug(Messages.getMessage("exception00"), e);
                break;
            }
            if (socket != null) {
                // SimpleAxisWorker worker = new SimpleAxisWorker(this, socket);
                if (doThreads) {
                    .
                    .
                    queue_.put( socket );
                } else {
                    .
                    .
                }
            }
        }
        .
        .
    }
    .
    .
}
```

Abbildung 12: *SimpleAxisServer* des dem *Half-Sync/Half-Async* Pattern folgenden Servers

In dieser Variante des *SimpleAxisServer*s besitzt die Klasse `SimpleAxisServer` die Klassenvariablen `THREAD_COUNT` und `MAX_QUEUE_SIZE`, die mittels Kommandozeilenparameter gesetzt werden können (so nicht die Default-

Belegung genutzt werden soll). Die Klassenvariable `MAX_QUEUE_SIZE` bestimmt die Maximalgröße der verwendeten Queue (die Queue wird via einer *Factory* mittels der Zeile `queue_ = ( new FIFOQueueFactory( MAX_QUEUE_SIZE ) ).createInstance();` instanziert).

Dann werden die Threads, die als Consumer agieren, instanziert; die Anzahl der zu instanzierenden Threads wird durch die Klassenvariable `THREAD_COUNT` festgelegt.

Als Einsprungspunkt für die Threads dient die Klasse `SimpleAxisWorkerDecorator`. Diese Klasse realisiert einen *Decorator* [vgl. das gleichnamige Pattern in GHJV96], der der Klasse `SimpleAxisWorker` neue Funktionalität hinzufügt.

In der Schleife, die die Verbindungen akzeptiert, wird jetzt nicht die Verarbeitung angestoßen, sondern der Request (in Form des Sockets der die Verbindung darstellt) mittels der Zeile `queue_.put( socket );` in der Queue abgelegt.

```
.
.
package org.apache.axis.transport.http;
.
.
public class SimpleAxisWorkerDecorator extends SimpleAxisWorker {

     .
     .
     public SimpleAxisWorkerDecorator(SimpleAxisServer server, Queue queue) {

          assert server != null;
          assert queue != null;

          this.server = server;
          queue_ = queue;
     }
     .
     .
     public void run() {
          assert queue_ != null;

          for ( ;; ) {

               socket = (Socket) queue_.get();

               assert socket != null;
```

```
                // Delegate to parent
                super.run();
            }

        }

        /**
         * Queue for the sockets
         **/
        private Queue queue_ = null;
}
```

Abbildung 13: *SimpleAxisWorkerDecorator* des dem *Half-Sync/Half-Async* Pattern folgenden Servers

In der `run` Methode der Klasse `SimpleAxisWorkerDecorator` wird mittels einer Endlosschleife realisiert, dass Sockets aus der Queue mittels der Zeile `socket = (Socket) queue_.get();` entnommen werden (falls keine Sockets zur Abarbeitung vorhanden sind, blockiert der Aufruf bis welche wieder vorhanden sind).

Nachdem ein Socket aus der Queue entnommen wurde, wird die weitere Verarbeitung mittels `super.run()` an die ursprüngliche Implementierung des `SimpleAxisWorker`s delegiert.

4.5 Server dem Leader/Followers Pattern folgend

4.5.1 Das Leader/Followers Pattern

Die folgende Darstellung des *Leader/Followers* Pattern ist wiederum an der in [POSA2] publizierten angelehnt.

Kontext

Eine Event-getriebene Applikation muss Requests effizient mittels mehreren Threads verarbeiten.

Problem

Multi-threading ist eine weitverbreitete Technik, um Events konkurrent zu verarbeiten. Es ist aber schwierig, Server Applikationen mit höchsten Performanceanforderungen, die einen hohen Durchsatz besitzen, mittels eines Multi-threaded Servers zu realisieren.

Forces:

- Um die Performance zu maximieren, müssen Overheadkosten, die durch die konkurrente Verarbeitung auftreten (z.B. Context Switches, Synchronisation etc.), minimiert werden.
- Threads müssen synchronisiert werden, um u.a. Race Conditions vorzubeugen.

Lösung

Der Hauptbestandteil der Lösung ist ein Thread Pool (siehe `LeaderFollowersThreadPool` in Abbildung 14), der die Methoden `join` und `promoteNewLeader` zur Verfügung stellt.

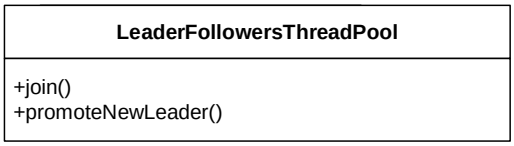


Abbildung 14: Die `LeaderFollowersThreadPool` Klasse

Threads, die in diesem Thread Pool abgelegt werden sollen, können dies mittels Aufrufs von `join` bewerkstelligen. Der erste Thread, der in diesem Pool abgelegt wird, übernimmt die *Leader*-Rolle, d.h. er wird nicht auf Wait gesetzt. Alle folgenden Threads übernehmen die *Follower*-Rolle, d.h. sie werden auf Wait gesetzt.

Der Thread, der die *Leader* Rolle übernommen hat, reagiert auf Events und arbeitet diese dann ab, er übernimmt während der Abarbeitung die Rolle eines *Processing* Threads. Bevor dieser Thread die Abarbeitung beginnt, wird mittels der Methode `promoteNewLeader` ein *Follower*-Thread zum nächsten *Leader*

„befördert“ (d.h. er wird geweckt). Wenn der *Processing*-Thread die Abarbeitung vollendet hat, nimmt er die Rolle eines *Followers*-Threads ein, falls ein anderer Thread die *Leader*-Rolle einnimmt, ansonsten übernimmt er die *Leader*-Rolle (dies geschieht durch Aufruf von `join`).

Zu den einzelnen Rollen und deren Übergänge siehe die Abbildung 15.

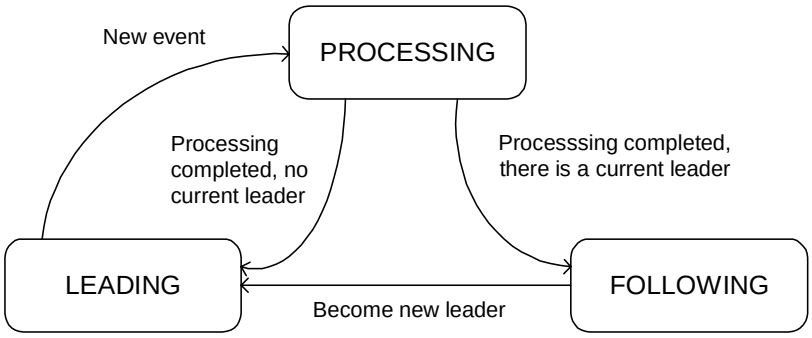


Abbildung 15: Zustandsdiagramm für das *Leader / Followers* Pattern [vgl. POSA2, S. 456]

Die Synchronisation geschieht über ein *Monitor*-Objekt [POSA2], welches sich die Methoden `join` und `promoteNewLeader` teilen.

Consequences

Vorteile:

- *Performance Verbesserungen* – Verglichen mit dem *Half-Sync/Half-Async* Pattern wird der Bedarf nach Allokationen von Heap Memory minimiert (da keine Queue verwendet wird), der Synchronisations-Overhead ist kleiner, da Threads keine Daten miteinander austauschen müssen und die Anzahl an Context Switches wird reduziert.

- *Einfachheit der Implementierung* – Die Implementierung dieses Patterns ist einfach und übersichtlich; und vereinfacht dadurch die Implementierung von konkurrenten Systemen.

Nachteile:

- *Mangelnde Flexibilität* – Architekturen, die auf dem *Half-Sync/Half-Async* Pattern beruhen, sind flexibler, z.B. können Elemente in der Queue entfernt oder neu priorisiert werden; das System kann mehrere Queues für verschiedene Prioritäten unterhalten. Diese Möglichkeiten stehen einer Implementierung dem *Leader/Followers* Pattern folgend nicht zur Verfügung, da keine explizite Queue besteht. Um ähnliche Funktionalität nachahmen zu können, müssen mehrere *Leader/Followers*-Gruppen mit z.B. unterschiedlichen Prioritäten von der Applikation unterhalten werden.

4.5.2 Implementierungsdetails

Die Implementierung des *SimpleAxisServer*s (wie in Abbildung 16 dargestellt), die dem *Leader/Followers* Pattern folgt, gleicht in den meisten Belangen der Variante, die das *Half-Sync/Half-Async* Pattern realisiert.

Die Implementierung benutzt auch das *Decorator* Pattern um Funktionalität dem ursprünglichen `SimpleAxisWorker` hinzuzufügen. Nur wird diesem `SimpleAxisWorkerDecorator` im Konstruktor ein Objekt der Klasse `ServerSocket` übergeben, um zu ermöglichen, dass der Thread, der die *Leader* Rolle besitzt, mittels Aufruf von `accept` auf eine Verbindung warten kann. Es existiert keine Schleife im *SimpleAxisServer* die Verbindungen iterativ akzeptiert.

```
.
.
package org.apache.axis.transport.http;
.
.
// Importing LeaderFollowersThreadPool
import da_pattern_axis.threads.LeaderFollowersThreadPool;
.
.
public class SimpleAxisServer implements Runnable {

     /**
      * Number of threads in the LeaderFollowersThreadPool
      **/
     private static int THREAD_COUNT = 100;
```

```
    /**
     * Array of threads with the size THREAD_COUNT. These threads
     * process requests by using the LeaderFollowersThreadPool.
     **/
    private Thread[] workers_ = null;

    /**
     * The LeaderFollowersThreadPool
     **/
    private LeaderFollowersThreadPool lfThreadPool_ = null;
    .
    .
    public void run() {
         .
         .
         // Setting up thread pool
         lfThreadPool_ = new LeaderFollowersThreadPool( THREAD_COUNT );

         // Setting up worker threads which process requests by using
         // the LeaderFollowersThreadPool
         workers_ = new Thread[ THREAD_COUNT ];

         for ( int i = 0; i < THREAD_COUNT; i++ ) {
              workers_[i] = new Thread(
                                   new SimpleAxisWorkerDecorator(
                                             this,
                                             serverSocket,
                                             lfThreadPool_
                                   )
                                   );
              workers_[i].start();
         }
         .
         .
    }
    .
    .
}
```

Abbildung 16: *SimpleAxisServer* des dem *Leader/Followers* Pattern folgenden Servers

Die Klasse `SimpleAxisWorkerDecorator` (siehe Ausschnitt in Abbildung 17) ruft in `run` die `join` Methode eines `LeaderFollowersThreadPool`-Objekts auf. Falls dieser Thread die *Leader*-Rolle innehat, wird der Kontrollfluss fortgesetzt und der Thread blockiert beim Methodenaufruf von `accept`.

Hat der Thread die *Follower*-Rolle inne, so blockiert der Aufruf von `join` bis der Thread die *Leader*-Rolle erhält.

Nachdem ein *Leader*-Thread eine Verbindung erhalten hat, befördert er mittels `promoteNewLeader` einen Thread (der mittels `accept` auf eine Verbindung warten soll) und delegiert den Aufruf an die `run`-Methode des `SimpleAxisWorker`.

```
.
.
package org.apache.axis.transport.http;
.
.
public class SimpleAxisWorkerDecorator extends SimpleAxisWorker {
     .
     -
     public SimpleAxisWorkerDecorator(SimpleAxisServer server, ServerSocket
serverSocket, LeaderFollowersThreadPool lfThreadPool) {

          assert server != null;
          assert serverSocket != null;
          assert lfThreadPool != null;

          this.server = server;
          serverSocket_ = serverSocket;
          lfThreadPool_ = lfThreadPool;
     }
     .
     .
     public void run() {

          assert serverSocket_ != null;
          assert lfThreadPool_ != null;

          for ( ;; ) {

               // Join the pool and wait until we become the leader thread
               lfThreadPool_.join();

               // We synchronously accept a connection the pattern in POSA 2
               // is more complicated because it uses async IO and waits on
               // more handles (e.g. select() and an fd_set with more than
               // one file descriptor)
               try {
                    socket = serverSocket_.accept();
               } catch ( Exception e ) {
                    LogConfig.getLogger().log( Level.INFO, "Exception in
accept!\n\nMessage:\n\n", new Object[] { e.getMessage() } );
               }

               assert socket != null;

               // Promote a new leader thread
               lfThreadPool_.promoteNewLeader();

               // Delegate to parent
               super.run();
          }
```

```
    }

    /**
     * ServerSocket on which to accept a connection
     **/
    private ServerSocket serverSocket_ = null;

    /**
     * LeaderFollowersThreadPool
     **/
    private LeaderFollowersThreadPool lfThreadPool_ = null;
}
```

Abbildung 17: *SimpleAxisWorker* des dem *Leader/Followers* Pattern folgenden Servers

5 Test Suite Tool

Um die Auswirkungen auf die Performance zu testen, wurde ein kleines generisches Testframework und eine Applikation für ebendieses erstellt. Die Applikation, die den Namen *Test Suite* trägt, dient hauptsächlich dem Ausführen von in einer Konfiguration definierten Testfällen. Zur Bedienung wird eine graphische Benutzeroberfläche kurz auch als GUI (Graphical User Interface) bezeichnet und eine Kommandozeilen-Schnittstelle bereitgestellt.

Um die Plattformunabhängigkeit zu gewährleisten, wurde zur Implementierung, wie auch schon beim Axis Server, Java herangezogen.

5.1 Benutzerschnittstelle

Wie schon zuvor erwähnt kann die Bedienung der Applikation auf Basis von zwei Schnittstellen erfolgen, die eine wäre die Kommandozeilen-Schnittstelle, die einen eingeschränkten Funktionsumfang besitzt, die im folgenden Unterkapitel erläutert wird. Im Rahmen der Beschreibung der Kommandozeilen-Schnittstelle wird auch auf Parameter eingegangen, die auch auf den Betrieb mit der graphischen Benutzerschnittstelle Einfluss haben.

Der volle Funktionsumfang steht dem Anwender durch Verwendung der graphischen Benutzeroberfläche zur Verfügung, die im übernächsten Unterkapital beschrieben wird.

5.1.1 Kommandozeilen Schnittstelle

Damit die Applikation auch von Skripten bzw. auch ohne graphische Benutzeroberfläche (zum Beispiel in der Kommandozeile eines Unix-Derivats ohne gestarteten X Window) genutzt werden kann, wird eine Kommandozeilen Schnittstelle zur Verfügung gestellt.

Um die möglichen Parameter und eine Beschreibung hierfür für die Kommandozeile zu bekommen, muss in das Verzeichnis, in dem *test_suite.jar*

liegt, gewechselt werden und mittels folgender Anweisung die Applikation gestartet werden:

```
java -cp test_suite.jar:$CLASSPATH da_pattern_axis.test_suite.Main -?
```

Unter Windows muss `:$CLASSPATH` durch `;%CLASSPATH%` ersetzt werden!

Danach wird die unten folgende Usage-Information dargestellt.

```
Usage:
java -cp test_suite.jar:$CLASSPATH da_pattern_axis.test_suite.Main
[ [ [-h] | [-?] | [-help] | [-v] ] |
[ [-d] [ [-t testSuiteConfiguration [testCase]] |
[testSuiteConfiguration [testCase]] ] ] ]

-d                      ... Starts application with debug logging
-h, -?, -help           ... Prints this text
-v                      ... Prints version information
-t                      ... Starts command line version
                            (testSuiteConfiguration is a mandatory
                            parameter)
testSuiteConfiguration ... Test suite configuration to start
testCase               ... Test case to start
```

Abbildung 18: Usage Information für die *Test Suite*-Applikation

Mit dem Parameter `-d` wird das Logging der Applikation aktiviert, dieser Parameter gilt auch für einen Betrieb mit graphischer Benutzeroberfläche. Um die Usage-Information darzustellen, sind folgende Parameter möglich `-h`, `-?`, `-help`. Mittels `-v` wird Information über die Version, den Zeitpunkt des Builds und ähnliches ausgegeben.

Mit dem Parameter `-t` wird angezeigt, dass der Kommandozeilenmodus verwendet werden soll. Nach diesem Parameter ist zumindest der Parameter `testSuiteConfiguration` (Dateiname einer Konfiguration) anzugeben. Darauf folgend kann als Parameter `testCase` der Name des Testfalls, in der Konfiguration, der ausgeführt werden soll, festgelegt werden.

Ist `testCase` nicht angegeben, so werden alle Testfälle in `testSuiteConfiguration` ausgeführt, bei angegebenen `testCase` wird nur dieser ausgeführt. Alle Parameter nach `-t` gelten auch für den Betrieb mit

graphischer Benutzeroberfläche (natürlich ohne vorhergehendes `-t`), wobei der Parameter `testSuiteConfiguration` fakultativ ist.

Nachdem die Test Suite bzw. ein Testfall ausgeführt wurde (und auch währenddessen), finden sich die Daten der Testausführung in den entsprechenden Dateien, die als *Output File* für die Testfälle in der Konfiguration (siehe Kapitel „*Konfiguration und Erstellung eigener Testfälle*“) bestimmt wurden. Der Inhalt einer *Output File* stellt sich, wie in der Beschreibung für das *Anzeigefenster für Output Files* im Kapitel „*Graphische Benutzeroberfläche*“ dar.

5.1.2 Graphische Benutzeroberfläche

In den folgenden Seiten wird der Aufbau der graphische Benutzeroberfläche dargestellt. Das erste Element der Benutzeroberfläche, mit dem der Benutzer in Kontakt tritt, ist das Hauptfenster, das als Schaltzentrale der Applikation dient.

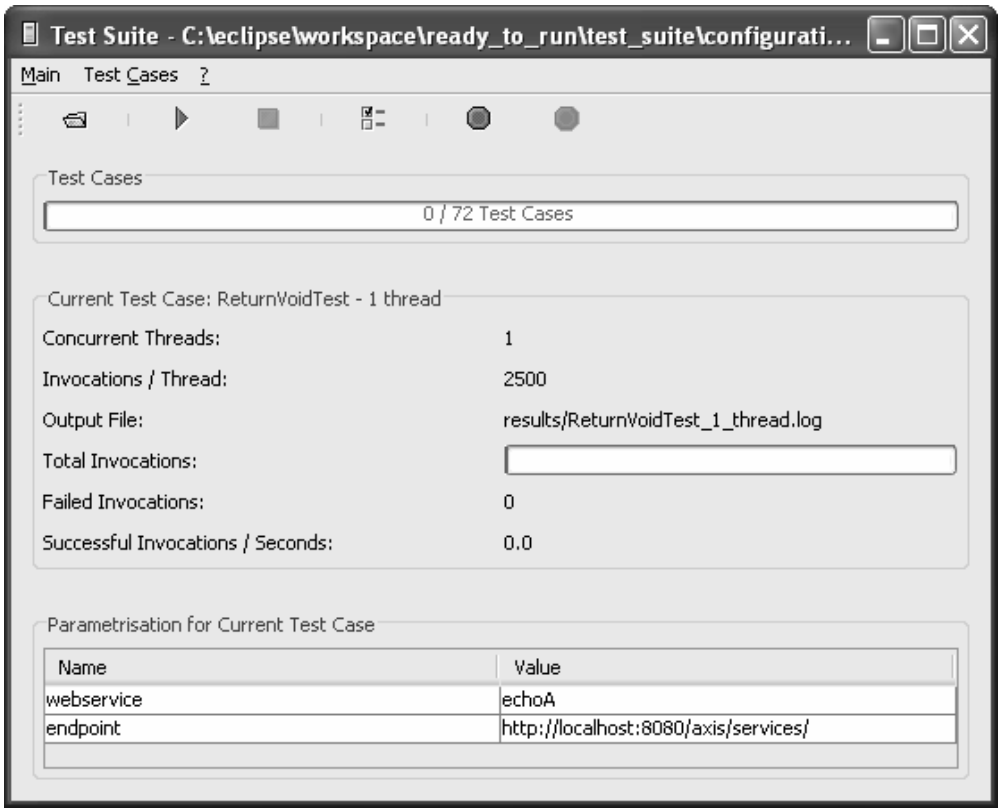


Abbildung 19: Hauptfenster der *Test Suite*-Applikation

Die Darstellung der relevanten Information im Hauptfenster wurde in drei Gruppen geteilt. Zuerst das Gruppenfeld *Test Cases*, das mittels einer Fortschrittsanzeige darstellt, wie viele Testfälle in einer Suite zur Ausführung ausgewählt wurden und bei welchem Testfall sich die Applikation gerade befindet.

Im nächsten Gruppenfeld mit der Bezeichnung *Current Test Case* wird Information zum aktuellen Testfall dargestellt. Neben der Bezeichnung *Current Test Case* ist der Name des aktuellen Testfalls dargestellt. Im Feld *Concurrent Threads* wird die Anzahl der konkurrenten Threads angegeben, *Invocations / Thread* gibt an wie oft der Testfall pro Thread aufgerufen werden soll.

Output File stellt den absoluten Dateinamen der Datei dar, die Information, wie z.B. Aufrufdauer eines Testfalls, Aufrufe pro Sekunde beinhaltet. Diese Datei wird bei einem laufenden Testfall kontinuierlich aktualisiert.

Total Invocations stellt eine Fortschrittsanzeige zur Verfügung, die die Anzahl der Aufrufe eines Testfalls, wie auch die Gesamtanzahl an Aufrufen wiedergibt. *Failed Invocations* gibt die Anzahl der fehlgeschlagenen Aufrufe an. Das Feld *Successful Invocations / Seconds* gibt die erfolgreichen Aufrufe pro Sekunde wieder.

Das letzte Gruppenfeld *Parametrisation for Current Test Case* stellt die Parametrierung des aktuellen Testfalls dar.

Zur Auslösung von Funktionen werden drei Menüs bereitgestellt, diese wären *Main*, *Test Cases* und *?*.

Im Menü *Main* kann unter anderem eine Konfiguration ausgewählt werden, auch das Starten und Stoppen der Ausführung einer Test Suite kann von hier aus erfolgen.

Alle zur Verfügung stehenden Menüpunkte und deren Icons in der Toolbar, so sie von der Toolbar aufrufbar sind, wie auch die Tastenkombinationen für ein Auslösen mittels der Tastatur sind in der folgenden Tabelle angeführt.

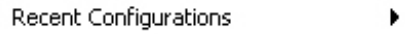
	Durch Auswahl dieses Menüpunkts wird ein Dialog zum Wählen einer Konfigurationsdatei für eine Test Suite dargestellt.
	Falls sich der Mauszeiger über diesem Menüpunkt befindet, wird ein weiteres Menü dargestellt, das die 10 letzten Konfigurationen zur Auswahl stellt[4]. Als Tooltip für jeden Eintrag wird der absolute Dateiname dargestellt.
	Startet die Ausführung der in der Test Suite definierten Testfälle.
	Stoppt die Ausführung einer Test Suite bzw. der zur Ausführung selektierten Testfälle.
	Beendet die Applikation.

Tabelle 3: Menüpunkte des Menüs Main

[4] Mit STRG+0 wird die zuletzt benutzte Konfiguration geöffnet (analoges gilt für STRG+1 bis STRG+9 um ältere Konfigurationen zu benutzen). Die letzten 10 Konfigurationen werden in der Datei *test_suite.properties* im Verzeichnis, das durch das Java Property *user.home* bestimmt wird, gespeichert.

Menüpunkte u.a. zum Starten und Stoppen von einzelnen Testfällen und zur Ansicht der *Output File* eines Testfalls finden sich im Menü *Test Cases*. Nach dem gleichen Schema wie zuvor folgt eine Tabelle mit allen Menüpunkten.

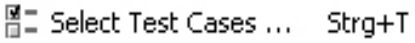
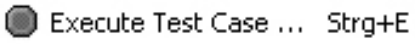
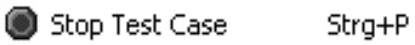
	Öffnet den Dialog *Select Test Cases*, in dem die in der Test Suite auszuführenden Testfälle gewählt werden können.
	Durch Auswahl dieses Menüpunkts wird der Dialog *Execute Test Case* geöffnet.
	Der gerade laufende Testfall wird mittels dieses Menüpunkts gestoppt.
	Falls sich der Mauszeiger über diesem Menüpunkt befindet, wird ein weiteres Menü dargestellt, das die 150 letzten Ergebnisse eines Testfalls zur Auswahl stellt, wird eines dieser Ergebnisse eines Testfalls ausgewählt so wird sie im *Anzeigefenster für Output Files* dargestellt[5]. Die maximal 150 letzten Ergebnisse werden immer in Tranchen von 25 Einträgen dargestellt, um die nächste Tranche darzustellen, muss der Menüpunkt *More Results* gewählt werden. Als Tooltip für jeden Eintrag wird der absolute Dateiname dargestellt.

Tabelle 4: Menüpunkte des Menüs Test Cases

[5] Mit ALT+0 wird die aktuelle bzw. aktuellste *Output File* geöffnet (analoges gilt für ALT+1 bis ALT+9 um ältere *Output Files* anzuzeigen).

Abschließend ist das Menü *?* darzustellen, welches nur einen Menüpunkt zur Darstellung des Dialogs *Info* besitzt, in diesem Dialog werden Versionsinformation und andere allgemeine Information über die Applikation, die Laufzeitumgebung etc. dargestellt.

Nach Betätigung des Menüpunkts *Select Test Cases* erscheint der in Abbildung 20 gezeigte Dialog zur Auswahl der auszuführenden Testfälle einer Suite.

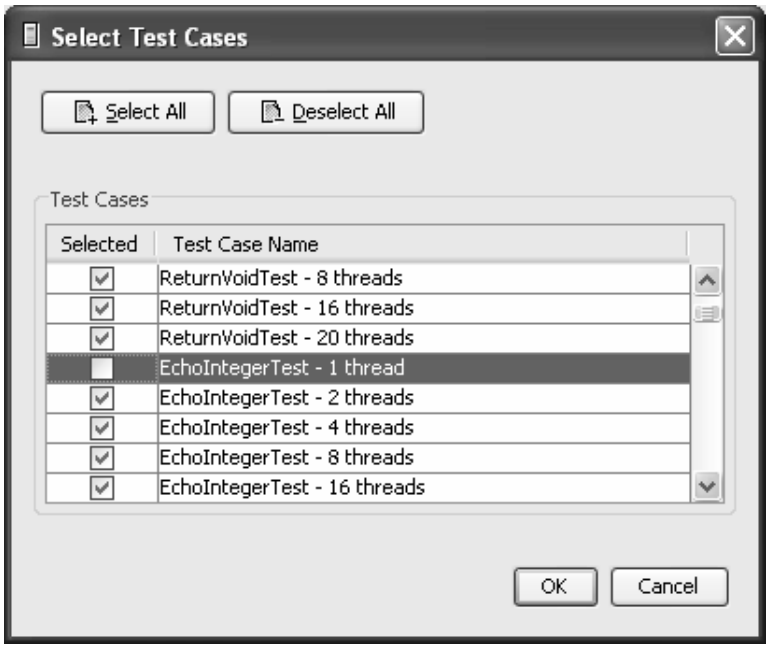


Abbildung 20: Select Test Cases Dialog

In diesem Dialog kann mittels der Checkbox in der Spalte *Selected* gewählt werden, ob ein Testfall bei Ausführung der Test Suite inkludiert sein soll oder auch nicht.

Mit den Buttons *Select All* werden alle Testfälle zur Ausführung ausgewählt, mittels *Deselect All* werden alle deselektiert.

Um die Auswahl zu übernehmen und zum Hauptfenster zurückzukehren, muss der Button *OK* betätigt werden. Die Auswahl wird nur übernommen, wenn mindestens ein Testfall ausgewählt ist, wenn nicht erfolgt eine Fehlermeldung nach Betätigung von *OK*.

Soll die Selektion nicht übernommen werden und zum Hauptfenster zurückgekehrt werden, so kann dies mittels des Buttons *Cancel* bewerkstelligt werden.

Soll ein nur einzelner Testfall ausgeführt werden, so kann dies mit dem Menüpunkt *Execute Test Case* erreicht werden.

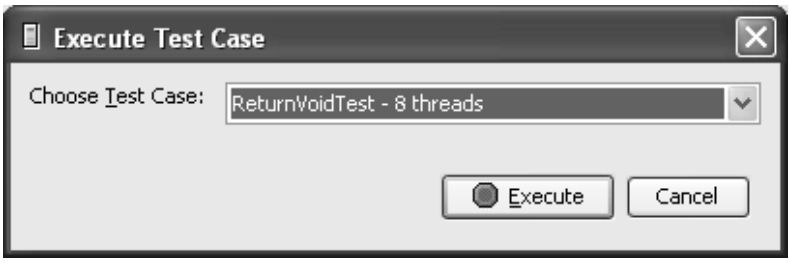


Abbildung 21: Execute Test Case Dialog

Um einen bestimmten Testfall auszuführen, kann in diesem Dialog aus einer Drop-Down Listbox ein Testfall gewählt werden und mittels des Buttons *Execute* zur Ausführung gebracht werden.

Durch Betätigung des Buttons *Cancel*, wird ohne einen Testfall auszuführen, zum Hauptfenster zurückgekehrt.

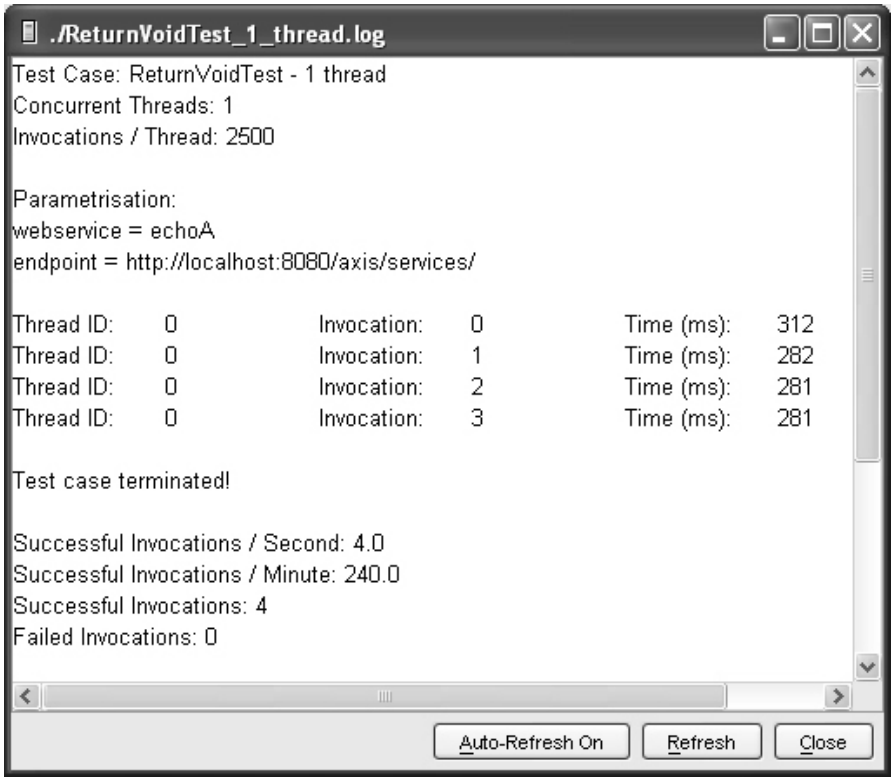


Abbildung 22: Anzeigefenster für Output Files

Wenn ein Testfall gestartet wird, protokolliert dieser Information in die *Output File*. Um die protokollierten Daten während bzw. nach einem Testlauf zu betrachten, kann über das Menü *Results* ein Anzeigefenster aktiviert werden, dessen Titel der absolute Dateiname des *Output Files* ist.

Im Feld *Test Case* wird der Name des Testfalls, im Feld *Concurrent Threads* die Anzahl der konkurrenten Threads und im Feld *Invocations / Thread*, wie oft der Testfall pro Thread aufgerufen wird, dargestellt.

Als nächstes wird der Abschnitt *Parametrisation* dargestellt, in diesem sind alle Parameter für den Testfall in der Form Key = Value aufgelistet (zur Definition von Parametern siehe das Kapitel „*Konfiguration und Erstellung eigener Testfälle*“).

Im nächsten Abschnitt sind die Testfall Aufrufe dargestellt, *Thread ID* ist die Nummer des Threads (0 bis *Concurrent Threads* – 1), *Invocation* die Nummer des Aufrufs (0 bis *Invocations / Thread* – 1) und *Time (ms)* die Dauer des Aufrufs in Millisekunden.

Wenn der Testfall beendet wurde, erscheint der Text *Test case terminated!* und danach Auswertungen wie *Successful Invocations / Second* die erfolgreichen Aufrufe pro Sekunde, *Successful Invocations / Minute* erfolgreiche Aufrufe pro Minute, *Successful Invocations* erfolgreiche Aufrufe und schließlich *Failed Invocations* die fehlgeschlagenen Aufrufe.

Durch Betätigung des Buttons *Auto-Refresh On* wird der Inhalt des Fensters automatisch mit neu hinzugekommenen Einträgen der *Output File* ergänzt, dieser Modus kann durch Betätigung des Buttons *Auto-Refresh Off* deaktiviert werden. Falls *Auto-Refresh On* aktiviert ist, ist der Button *Refresh* deaktiviert, da ja nun periodisch eine Aktualisierung (sprich ein Refresh) durchgeführt wird.

Eine einmalige Ergänzung der Einträge kann durch Betätigung des Buttons *Refresh* erreicht werden.

5.2 Konfiguration und Erstellung eigener Testfälle

```
<?xml version="1.0" encoding="UTF-8"?>

<testSuite>

    <!-- Globale Parameter die fuer alle Testfaelle gelten -->
    <param name="endpoint" value="http://localhost:8080/axis/services"/>
    <param name="webservice" value="echoA"/>

    <testCase
    name="echoDateTest"
    class="da_pattern_axis.test_suite.test_cases.echoDateTest"
    threads="20"
    invocationsPerThread="200"
    outputFile="echoDate.log"/>

    <testCase
    name="echoStringTest with 64kb"
    class="da_pattern_axis.test_suite.test_cases.echoStringTest"
    threads="20"
    invocationsPerThread="200"
    outputFile="echoString.log">
        <!-- Lokaler Parameter gilt nur fuer diesen Testfall -->
        <param name="dataFile" value="test64kb.dat"/>
    </testCase>

</testSuite>
```

Abbildung 23: Beispielkonfiguration einer Test Suite

Das oben abgebildete XML-Dokument [vgl. W3C04] ist ein Beispiel für eine typische Konfiguration. Im Folgenden wird der Aufbau dieser XML-Dokumente zur Konfiguration beschrieben und auch die Erstellung und Einbindung eigener Testfälle.

Auf Grund der Einfachheit des XML-Dokuments wird auf eine Validierung mittels Document Type Definition (DTD) [W3C04a] oder Schema [W3C01] verzichtet.

Das Root-Element des Dokuments ist `<testSuite>`, es besitzt als Child-Elemente `<param>` und `<testCase>`-Elemente.

Ein `<param>`-Element beschreibt einen Parameter für einen bzw. alle Testfälle, dessen Name durch das Attribut `name` und der Wert durch `value` bestimmt wird.

`<param>`-Elemente, die ausschließlich ein Child-Element des Root-Elements sind, werden globale Parameter genannt. D.h. die Parameterdefinition gilt für alle Testfälle (im Dokument für alle `<testCase>`-Elemente).

Falls ein `<param>`-Element das Child-Element eines `<testCase>`-Elements ist, wird dieser Parameter lokal genannt, d.h. er gilt nur für sein Parent-`<testCase>` Element.

Die `<testCase>`-Elemente definieren die eigentlichen Testfälle, ein `<testCase>` besitzt folgende Attribute (alle zwingend):

- `name`, ist der Name des Testfalls.
- `class`, die Klasse des Testfalls (diese muss im Classpath sein!).
- `threads`, die Anzahl an konkurrenten Threads, die den Testfall ausführen.
- `invocationsPerThread`, wie oft der Testfall aufgerufen werden soll.
- `outputFile`, Datei, in der die Ergebnisse (z.B. Dauer eines Aufrufs) geschrieben werden.

Folgende Einschränkungen werden beim Parsing des Dokuments programmatisch geprüft:

- Das Attribut `name` des Elements `<testCase>` muss innerhalb von `<testSuite>` eindeutig sein.
- Es muss mindestens ein `<testCase>` in `<testSuite>` definiert worden sein.

- Das Attribut `threads` muss ein Integer (in Java, d.h. signed 32-Bit [vgl. JLS00, § 3.10.1 und § 4.2.1]) und größer Null sein.

- Das Attribut `invocationsPerThread` muss ein Integer (in Java, d.h. signed 32-Bit) und größer Null sein.

Wurde ein globaler Parameter nach einem `<testCase>`-Element definiert, so gilt er trotzdem für dieses Element, d.h. der Ort der Definition eines globalen Parameters ist unerheblich.

Eigene Testfälle können implementiert werden, indem eine Klasse von `da_pattern_axis.test_suite.suite.LoggingTestCase` abgeleitet wird, d.h. dass sich diese Klasse auch im Classpath befinden muss. Nach erfolgreicher Kompilation können die Testfälle über die Konfiguration eingebunden werden, wobei wichtig ist, dass bei Ausführung der *Test Suite*-Applikation die neuen Testfälle sich wiederum im Classpath befinden.

Es folgt eine Beispielimplementierung eines Testfalls, der keine Operation ausführt, dieser ist nach dem Assembler Mnemonik NOP (No Operation) benannt.

```
// Testfall der keine Operation ausführt
// (NOP nach dem Assembler Mnemonik)

import java.util.Properties;

import da_pattern_axis.test_suite.suite.LoggingTestCase;

import da_pattern_axis.test_suite.suite.IrrecoverableTestCaseException;
import da_pattern_axis.test_suite.suite.TestCaseException;

public class NOP extends LoggingTestCase {

    public void setParams(Properties params)
        throws IrrecoverableTestCaseException {
        // Pruefen der zu setzenden Parameter, falls nicht
        // in Ordnung wird eine IrrecoverableTestCaseException
        // geworfen

        // Alles OK Parameter setzen
        super.setParam( params );
    }

    public void executeTestCase()
        throws IrrecoverableTestCaseException, TestCaseException {
        // Hier kommt normal der eigentliche Code des Testfalls
    }
}
```

Abbildung 24: Beispielimplementierung eines Testfalls

Die Methode `setParams` dient dem Setzen der Parameter für einen Testfall (globale und lokale Parameter aus der Konfiguration). Falls ein Parameter nicht akzeptiert werden soll, d.h. er ist ungültig und der Testfall kann nicht ausgeführt werden, kann durch Werfen einer `IrrecoverableTestCaseException` angezeigt werden, dass der Testfall mit dieser Parametrierung nicht ausgeführt werden kann. So die Parametrierung für den Testfall gültig ist, kann durch Propagierung des Aufrufs (durch die Zeile `super.setParam( params );`) diese übernommen werden.

Der eigentliche Testfall befindet sich in der Methode `executeTestCase`, falls ein Aufruf als fehlgeschlagen gewertet werden soll, d.h. ein Fehler, der auftreten kann (d.h. ein Auftreten ist vorhersehbar) ist vorgefallen, kann dies durch Werfen einer `TestCaseException` geschehen.

Bei unvorhergesehenen Fehlern, die ein Beenden des Testfalls und/oder der ganzen Test Suite zur Folge haben sollen, ist eine `IrrecoverableTestCaseException` zu werfen.

Wenn mehr als ein Thread zur Testausführung bestimmt ist, d.h. das Attribut `threads` eines `<testCase>`-Elements ist größer als eins, ist darauf zu achten, dass die Methode `executeTestCase` nicht als `synchronized` deklariert wurde. Da, falls sie als `synchronized` deklariert wurde, alle Aufrufe des Testfalls serialisiert werden, d.h. der Test läuft ab, als wäre nur ein aufrufender Thread vorhanden (keine Parallelisierung).

5.3 Implementierung des Tools

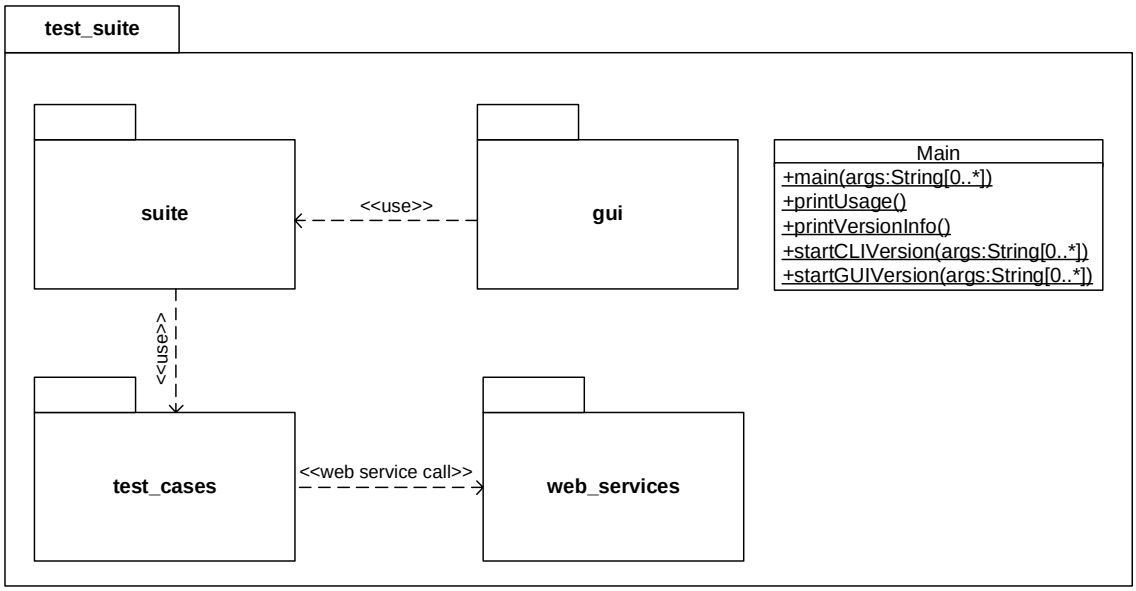


Abbildung 25: Paket Struktur der *Test Suite-Applikation*

Wie aus Abbildung 25 zu entnehmen ist, besteht die *Test Suite*-Applikation (welche sich im Paket `da_pattern_axis.test_suite` befindet) aus vier Paketen.

Das Paket `gui` beinhaltet Klassen und Ressourcen für die graphische Oberfläche. Das Paket `suite` enthält die Klassen, die die eigentliche Applikationslogik darstellen. In den Paketen `web_services` und `test_cases` befinden sich die Klassen, die den Web Service und die Testfälle realisieren und werden im Kapitel „*Implementierte Testfälle*“ genauer beschrieben.

Das Paket `da_pattern_axis.test_suite` enthält nur eine Klasse mit dem Namen `Main`, diese Klasse stellt den Einsprungspunkt (genauer die Methode `main`) der Applikation dar. Je nach Kommandozeilenparameter werden die Methoden `printUsage`, `printVersionInfo`, `startCLIVersion` (startet Kommandozeilenversion) oder `startGUIVersion` (startet graphische Benutzeroberfläche) von `main` aufgerufen.

In den folgenden zwei Unterkapitel werden die Pakete `gui` und `suite` genauer erläutert, wie schon erwähnt ist eine genauere Beschreibung der Klassen für den Web Service und die Testfälle in „*Implementierte Testfälle*“ zu finden.

5.3.1 Implementierung der graphischen Benutzeroberfläche

Da die im Paket *gui* realisierte graphische Benutzeroberfläche eine Verwendung eines Frameworks (in diesem Fall Swing, eine Einführung in Swing kann z.B. u.a. in [Kru02] gefunden werden) darstellt und dadurch kaum eine eigene technische Entwurfsleistung notwendig war, werden die Klassen nur aufgezählt und kurz beschrieben, welches Oberflächenelement der Benutzeroberfläche sie realisieren.

`ExecuteTestCaseDialog`	Diese von `javax.swing.JDialog` abgeleitete Klasse realisiert den Dialog *Execute Test Case*.
`InfoDialog`	Diese Klasse wurde von `javax.swing.JDialog` abgeleitet und realisiert den

	Dialog *Info*.
`MainWindow`	Ist eine Klasse, die von `javax.swing.-JFrame` abgeleitet wurde und das *Hauptfenster* der Applikation realisiert.
`MainWindowProgressInfo`	Diese Klasse implementiert die Schnittstelle `da_pattern_axis.-test_suite.suite.ProgressInfo` und besitzt die Aufgabe Nachrichten von `da_pattern_axis.test_suite.suite.-LoggingTestSuite` und `da_pattern_axis.test_suite.suite.-TestCaseRunner` zu verarbeiten und an die Benutzeroberflächenelemente weiterzupropagieren.
`MessageDialog`	Diese Klasse die von `javax.swing.-JDialog` abgeleitet wurde, realisiert einen Dialog für Meldungen an den Benutzer z.B. Fehlermeldungen.
`ParametrisationTableModel`	Diese von `javax.swing.table.-AbstractTableModel` abgeleitete Klasse stellt den Inhalt der Tabelle im Gruppenfeld *Parametrisation for Current Test Case* dar, sie wird von der Klasse `MainWindow` verwendet. Diese Klasse ist notwendig, da in Swing die Komponenten dem *Model View Controller* Pattern [vgl. GHJV96, S. 5 – 8 und POSA1] folgend entworfen wurden.
`ResultWindow`	Mittels dieser von `javax.swing.JFrame` abgeleiteten Klasse wird das *Anzeige-*

	fenster für Output Files realisiert.
`SelectTestCasesDialog`	Diese Klasse wurde von `javax.swing.-JDialog` abgeleitet und realisiert den Dialog Execute Test Case.
`TestCasesTableModel`	Diese von `javax.swing.table.-AbstractTableModel` abgeleitete Klasse stellt den Inhalt der Tabelle im Gruppenfeld *Test Cases* dar, sie wird von der Klasse `SelectTestCasesDialog` verwendet. Es gelten auch für diese Klasse die Anmerkungen zur der Klasse `ParametrisationTableModel`.

Tabelle 5: Klassen der graphischen Benutzeroberfläche

Bei der Implementierung ist zu beachten, dass Swing nicht thread-safe ist und daher zum Aktualisieren von Benutzeroberflächenelementen aus einem anderen Thread die Methoden `SwingUtilities.invokeAndWait` (blockierender Aufruf) bzw. `SwingUtilities.invokeLater` (nicht-blockierender Aufruf) verwendet wurden und zu verwenden sind [vgl. MuWa98].

5.3.2 Implementierung des Test Frameworks

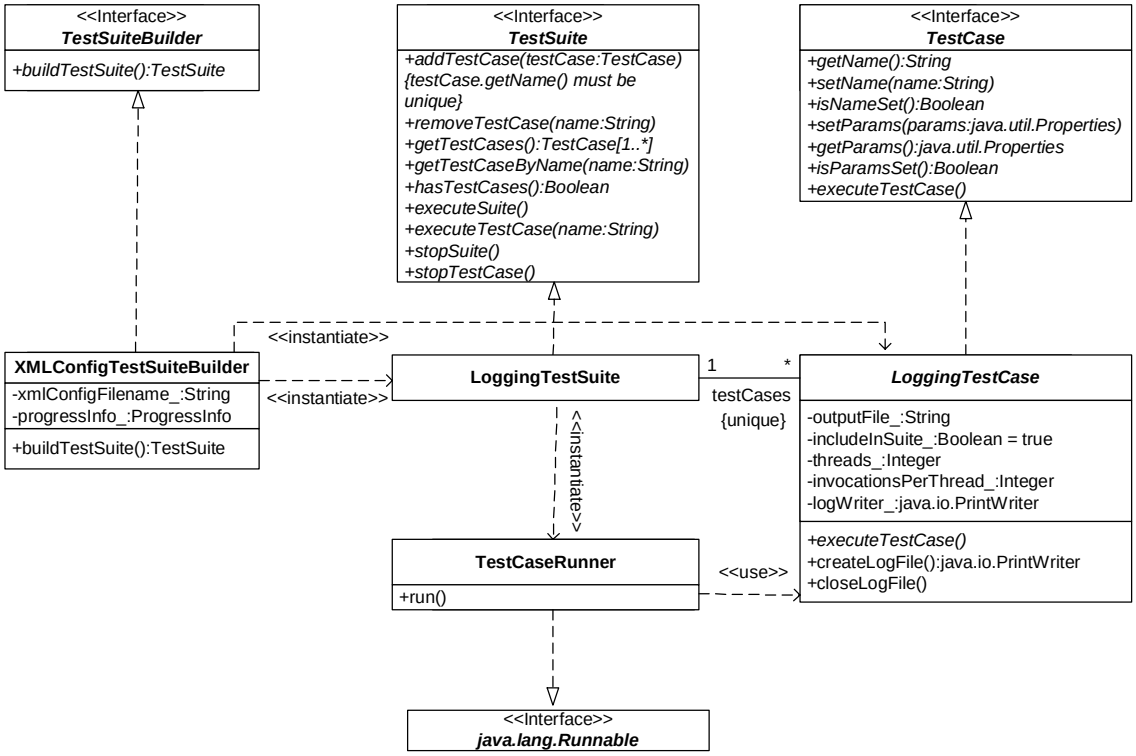


Abbildung 26: Klassendiagramm des Test Frameworks 1/2

Die im Klassendiagramm in Abbildung 26 zuoberst abgebildeten Interfaces definieren die drei wichtigsten Schnittstellen, die die Klassen des Test Frameworks zu implementieren haben.

`TestSuiteBuilder` stellt ein Interface für Klassen dar, die dem *Builder* Pattern [GHJV96] entsprechend, eine konkrete Klasse, die ein `TestSuite`-Interface implementiert, erstellt und entsprechende `TestCase`-Objekte hinzufügt (mittels der Methode `TestSuite.addTestCase`).

Diese Schnittstelle wird von `XMLConfigTestSuiteBuilder` implementiert, diese Klasse erbaut ein `LoggingTestSuite`-Objekt mit deren zugehörigen `LoggingTestCase`-Objekte aus einer XML-Konfiguration, wie in dem entsprechenden Kapitel dargestellt. Die Klassen mit dem Präfix `Logging` schreiben Daten wie Aufrufdauer, Aufrufnummer etc. in eine Datei, die in der Benutzeroberfläche mit *Output File* bezeichnet wird. Ein jedes `TestCase`-Objekt wird durch das Attribut `name_` (zugreifbar mittels den Methoden `getName` und

setName) identifiziert, d.h. in einem TestSuite-Objekt muss der Name des TestCase-Objekts eindeutig sein.

Das Interface TestSuite stellt die Schnittstelle dar, die alle Objekte implementieren müssen, die eine Sammlung von TestCase-Objekten darstellen.

Es besitzt Methoden mittels denen die Ausführung der Testfälle (executeSuite) oder eines einzelnen Testfalls (executeTestCase) angestoßen werden kann. Mit den entsprechenden stopp-Methoden kann die Ausführung beendet werden, es ist zu beachten, dass die Ausführung eines Testfalls (damit ist die Methode executeTestCase eines TestCase-Objekts gemeint) nicht unterbrochen wird (z.B. durch die Methode interrupt), sondern der Testfall zu Ende geführt wird.

Die Ausführung der Testfälle in einem LoggingTestSuite-Objekt geht wie folgt vonstatten: Es wird in executeSuite über alle LoggingTestCase-Objekte iteriert, falls beim aktuellen LoggingTestCase-Objekt das Attribut includeInSuite_ gesetzt wurde, wird die Methode LoggingTestSuite.executeTestCase mit dem Parameter LoggingTestCase.getName aufgerufen.

In der Methode executeTestCase werden dann die Threads und deren TestCaseRunner-Objekte instanziert. Dem TestCaseRunner-Konstruktor werden als Aktualparameter u.a. das LoggingTestCase Objekt, das als Test ausgeführt werden soll, eine Referenz auf das aufrufende LoggingTestSuite-Objekt, ein ProgressInfo-Objekt und ein Objekt der Klasse da_pattern_axis.threads.-Barrier übergeben.

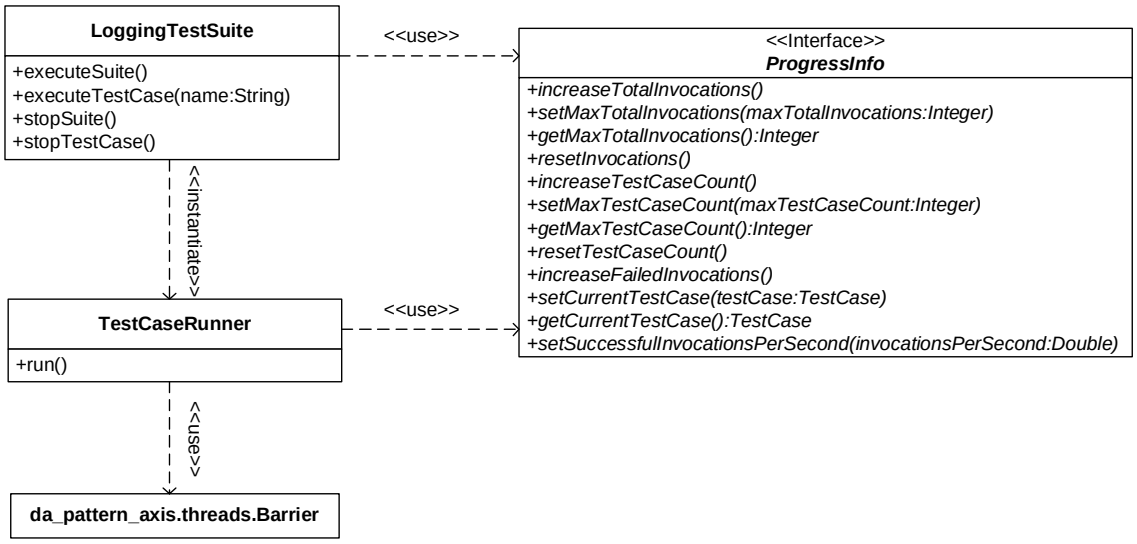


Abbildung 27: Klassendiagramm des Test Frameworks 2/2

Das `Barrier`-Objekt dient dazu, dass alle Threads zugleich ihre Ausführung fortsetzen, wenn alle `TestCaseRunner`-Objekte vollständig instanziert wurden (Eine genauere Beschreibung des Synchronisationsmechanismus Barrier findet sich in [Tan02, S. 140f]).

D.h. ein `TestCaseRunner`-Thread ruft in seiner `run`-Methode `Barrier.waitForNotification` auf und wird erst geweckt. wenn alle zu instanzierenden Objekte `waitForNotification` aufgerufen haben.

Wenn der erste `TestCaseRunner`-Thread geweckt wurde, wird die Startzeit des Testfalls genommen.

Nun wird eine Schleife durchlaufen, die solange ausgeführt wird, bis die erforderliche Anzahl an Testfall-Aufrufen erreicht wurde oder der Thread zu stoppen ist. In dieser Schleife wird die Startzeit des Aufrufs genommen, dann der eigentliche Testfall aufgerufen (die Methode `LoggingTestCase.executeTestCase`) und danach wird schließlich die Endzeit des Aufrufs genommen.

Auf Basis, ob eine Exception geworfen wurde, wird ein Aufruf als erfolgreich oder fehlgeschlagen gewertet. Tritt eine `IrrecoverableTestCaseException` auf, wird die Ausführung beendet.

Da für die Zeitmessung die Methode `System.currentTimeMillis` verwendet wird, ist zu beachten, dass die Granularität der Zeitmessung vom darunterliegenden Betriebssystem abhängt, welches die Ergebnisse verfälschen kann (z.B. ist die Granularität bei manchen Windows Systemen bei 10 ms) [vgl. Vol04]. Sollte eine genauere Zeitmessung benötigt werden, so muss auf andere Bibliotheken bzw. auf JNI (Java Native Interface) zurückgegriffen werden.

Der Kontrollfluss in `executeTestCase` in der Klasse `LoggingTestSuite` blockiert nach Instanzierung aller `TestCaseRunner`-Objekte so lange (über ein Monitor Objekt [vgl. Tan02, S.135 – 137 und POSA2]) bis alle `TestCaseRunner`-Threads terminiert sind. Dies wird durch einen Zähler, der als Klassenvariable der Klasse `LoggingTestSuite` definiert wurde, erreicht; bei Instanzierung der `TestCaseRunner`-Objekte wird er inkrementiert. Nach Verlassen der Schleife, die die Testfälle in `TestCaseRunner` aufruft, dekrementiert. Falls der Zähler den Wert null annimmt, wird der wartende Thread geweckt.

Nachdem der Thread geweckt wurde (wir befinden uns in `LoggingTestSuite.executeTestCase`), wird die Endzeit des Testfalls genommen.

Ereignisse, die `LoggingTestSuite`- und `TestCaseRunner`-Objekte zu anderen Objekten weiterpropagieren wollen, werden durch Aufrufe der entsprechenden Methoden, die durch das Interface `ProgressInfo` definiert wurden, realisiert.

Dieses Interface wird benötigt, um Ereignisse an die Benutzeroberfläche weiterpropagieren zu können.

5.4 Implementierte Testfälle

Die Testfälle orientieren sich an den vom W3C im Dokument „*SOAP Version 1.2 Specification Assertions and Test Collection*“ im Kapitel „*3.4 RPC Methods / Procedures Used by the Test Collection*“ festgelegten Methoden [vgl. W3C03b].

Diese Methoden wurden in einem Web Service implementiert, die dann von den Testfällen aufgerufen werden.

5.4.1 Der Web Service mit den Methoden für die Testfälle

Der Web Service wurde in der Klasse `SOAP12TestCollectionRPCService` im Paket `da_pattern_axis.test_suite.web_services` implementiert.

Die folgende Beschreibung der Methoden des Web Service ist an die Beschreibung in [W3C03b] angelehnt.

Zusätzlich zu den Typen, die im Namespace `http://www.w3.org/2001/-XMLSchema` definiert sind, gibt es die Typen `SOAPStruct`, `SOAPStructStruct` und `SOAPArrayStruct` im Namespace `http://soapinterop.org/xsd`.

`SOAPStruct` besitzt drei Komponenten, die wie folgt definiert sind:

- Element mit dem Typ `string`
- Element mit dem Typ `int`
- Element mit dem Typ `float`

`SOAPStructStruct` besitzt vier Komponenten, die wie folgt definiert sind:

- die drei Komponenten von `SOAPStruct`
- zusätzlich ein Element mit dem Typ `SOAPStruct`

`SOAPArrayStruct` besitzt vier Komponenten, die wie folgt definiert sind:

- die drei Komponenten von `SOAPStruct`
- zusätzlich ein Element mit dem Typ `string[]` (also ein `string`-Array)

Die Klassen `SOAPStruct`, `SOAPStructStruct` und `SOAPArrayStruct` liegen als Implementierung im Paket `da_pattern_axis.test_suite.test_cases` vor.

Es folgt nun eine Aufzählung und Beschreibung der einzelnen implementierten Methoden:

returnVoid
Diese Methode besitzt weder Eingabe- noch Ausgabeparameter. Sie besitzt auch keinen Rückgabewert.

echoStruct
Diese Methode besitzt einen Eingabeparameter und einen Rückgabewert mit dem Typ `SOAPStruct`. Diese Methode liefert den Eingabeparameter zurück.

echoStructArray
Diese Methode besitzt einen Eingabeparameter und einen Rückgabewert mit dem Typ `SOAPStruct[]` (also ein `SOAPStruct`-Array). Diese Methode liefert den Eingabeparameter zurück.

echoStructAsSimpleTypes
Diese Methode besitzt einen Eingabeparameter und drei Ausgabeparameter (für die Rückgabewerte). Der Eingabeparameter ist vom Typ `SOAPStruct`, die Ausgabeparameter entsprechen den Komponenten von `SOAPStruct` (also Parameter mit dem Typ `int`, `float` und `string`). Diese Methode liefert die Komponenten des Eingabeparameters zurück.

echoSimpleTypesAsStruct
Diese Methode besitzt drei Eingabeparameter (die Komponenten von `SOAP-Struct` (also Parameter mit dem Typ `int`, `float` und `string`)). Der

Rückgabewert ist vom Typ `SOAPStruct`. Diese Methode liefert die Eingabeparameter als Komponenten eines `SOAPStruct`-Objekts zurück.

echoNestedStruct

Diese Methode besitzt einen Eingabeparameter und einen Rückgabewert mit dem Typ `SOAPStructStruct`. Diese Methode liefert den Eingabeparameter zurück.

echoNestedArray

Diese Methode besitzt einen Eingabeparameter und einen Rückgabewert mit dem Typ `SOAPArrayStruct`. Diese Methode liefert den Eingabeparameter zurück.

countItems

Diese Methode besitzt einen Eingabeparameter mit dem Typ `string[]` (also ein `string`-Array) und einen Rückgabewert mit dem Typ `int`. Diese Methode liefert die Anzahl der Elemente des Eingabeparameters zurück.

isNil

Diese Methode besitzt einen Eingabeparameter mit dem Typ `string` und einen Rückgabewert mit dem Typ `boolean`. Diese Methode liefert `true` zurück, falls kein Eingabeparameter übergeben wurde oder ein Attribut `xsi:nil` mit dem Wert 1 besitzt, ansonsten wird `false` zurückgeliefert.

Die im folgenden aufgezählten Methoden besitzen einen Eingabeparameter vom Typ, der aus dem Teil des Methodennamens nach dem Präfix **echo** hervorgeht, und liefern den Eingabeparameter zurück: **echoFloatArray**, **echoStringArray**, **echoIntegerArray**, **echoBase64**, **echoBoolean**, **echoDate**, **echoDecimal**, **echoFloat**, **echoString** und **echoInteger**.

5.4.2 Die Testfälle für den Web Service

Die Klassen, die im Paket `da_pattern_axis.test_suite.test_cases` die Testfälle realisieren, rufen die entsprechenden Methoden des Web Service mittels des Axis Frameworks auf. Eine jede Klasse, die einen Testfall realisiert, besitzt den Suffix `Test`. Die Methode, die ein Testfall aufruft, ist anhand des

Präfix, der dem Methodennamen mit großem Anfangsbuchstaben entspricht, festgelegt. Beispielsweise ruft der Testfall in der Klasse `EchoStringTest` die Methode `echoString` des Web Service auf.

Alle Testfälle benötigen den Parameter `endpoint`, der die URL des Web Service angibt, und `webservice`, der den Namen des zu benutzenden Web Service angibt. Wenn von Parametern die Rede ist, sind Parameter in der XML-Konfiguration gemeint, siehe das Kapitel „*Konfiguration und Erstellung eigener Testfälle*“.

Der Name des Web Service in dieser Implementierung, der die im vorherigen Kapitel beschriebenen Methoden enthält, lautet `echoA`.

Die Testfälle können in fünf Klassen eingeteilt werden:

- **Testfälle, die keine Werte übergeben**
  Es existiert nur ein Testfall, der keine Parameter besitzt und keine Werte als Argumente an eine Methode des Web Service übergibt, es ist der Testfall, der in der Klasse `ReturnVoidTest` implementiert wurde.

- **Testfälle, die nicht-parametrierbare Werte übergeben**
  Diese Testfälle übergeben als Argumente Werte, die nicht parametrierbar sind, d.h. sie werden automatisch generiert. Dies wären die in den Klassen `EchoIntegerTest`, `EchoFloatTest`, `EchoStructTest`, `EchoStructAsSimpleTypesTest`, `EchoSimpleTypesAsStructTest`, `EchoNestedStructTest`, `EchoDateTest`, `EchoDecimalTest`, `EchoBooleanTest` und `IsNilTest` realisierten Testfälle.

- **Testfälle, die ein Array übergeben dessen Werte nicht parametrierbar sind**
  Diese Testfälle besitzen den Parameter `arraySize`, der die Größe des zu übergebende Arrays angibt. Klassen, die diese Art von Testfällen realisieren, sind `EchoIntegerArrayTest`, `EchoFloatArrayTest`, `EchoStructArrayTest` und `EchoNestedArrayTest`.

- **Testfälle, die größere Daten ohne Array übertragen**
  Mittels des Parameters `dataFile` kann eine Datei mit zu übertragenden Daten für diese Testfälle angegeben werden. Folgende Klassen realisieren diese Testfälle: `EchoBase64Test` und `EchoStringTest`.

- **Testfälle, die größere Daten mit einen Array übertragen**
  Diese Testfälle besitzen zusätzlich zum Parameter `dataFile` noch den Parameter `elementsPerRow`, der die Anzahl an Elementen pro Array Komponente angibt, in die die Daten aus der mittels `dataFile` bestimmten Datei aufgeteilt werden soll. Die Klassen `EchoBase64Array-Test`, `EchoStringArrayTest` und `CountItemsTest` realisieren diese Testfälle.

6 Benchmark Suite

Benchmark bedeutet nach [Web93]: „*…something that serves as a standard by which others may be measured*“, das heißt um zu vergleichen benötigt man einen Standard, anhand dessen Alternativen verglichen werden können. Benchmarks im Bereich der Informationstechnologie sind üblicherweise eine Sammlung von Tests mit denen Charakteristika z.B. Performance (Throughput, Responsezeiten) getestet und verglichen werden können.

Wie in [KeWi00] dargelegt, wird für gewöhnlich zwischen Mikro- und Makrobenchmarks unterschieden. Mikrobenchmarks konzentrieren sich auf einen spezifischen Aspekt eines Systems. z.B. wie lange dauert es, um 50.000 Dreiecke zu zeichnen, wie viele Integer-Operationen pro Sekunde etc., wohingegen Makrobenchmarks mehrere Aspekte abdecken, z.B. der SPECjAppServer2004 Benchmark, der eine Applikation aus dem Bereich des Business Computings mit Supply Chain Management, Inventarverwaltung und weiteren Bereichen zu Testzwecken verwendet [vgl. Spec05].

Für diesen Benchmark wurden die Testfälle, aus denen für das Test Suite Tool implementierten ausgewählt, und stellen daher ein Mikrobenchmark dar, das heißt, sie testen nur bestimmte Aspekte des Systems.

Die Benchmark Suite wurde auf einem Rechner, der der unten folgende Spezifikation (Hardware wie auch Softwaremäßig) entspricht, durchgeführt.

Server / Client:

Prozessor:	AMD Athon XP 3000+ (mit 2,16 GHz)
Hauptspeicher:	512 MB DDR RAM mit 333 MHz Takt
Betriebssystem:	Fedora Core 2 mit Kernel 2.6.8-1.521
JDK:	Sun JDK 1.4.2_05
Optionen Server:	-server -Xmx256m
Optionen Client:	-Xms128m -Xmx256m
Web Service Framework:	Apache Axis 1.1

Auf die Durchführung in einem Netzwerk wurde mangels Hardware verzichtet.

Folgende Testfälle wurden für den Benchmark ausgewählt:

- `ReturnVoidTest`
  Dieser Test besitzt keine Parameter und auch keinen Rückgabewert.

- `EchoIntTest`
  Es wird eine Pseudozufallszahl als Eingabeparameter übergeben und diese Zahl wird zurückgegeben.

- `EchoIntArrayTest`
  Bei diesem Test wird ein Array das aus 32 Elementen besteht und mit Pseudozufallszahlen gefüllt wurde übergeben und auch zurückgeliefert.

- `EchoStructTest`
  Dieser Testfall übergibt als Eingabeparameter einen selbstdefinierten Typ namens `SOAPStruct` der vom Web Service auch wieder zurückgeliefert wird. Die `int`-Komponente und die `float`-Komponente des `SOAPStruct`-Typs werden mit einer Pseudozufallszahl gefüllt und die `string`-Komponente mit dem String „TEST“.

- `EchoStructArrayTest`
  Für diesen Testfall gilt das gleiche wie für den `EchoStructTest`, außer dass ein Array, das aus 32 Elementen besteht, erzeugt wird und die `string`-Komponenten für ein jedes Array-Element aus dem String „TEST “ plus des Indexes des Elements besteht.

- `EchoStringTest`
  Es wurde mit Dateien, die alphanumerische Zeichen in UTF-8 beinhalten, mit einer Größe von 2, 4, 8, 16, 32, 64 und 128 KB als Daten für den Eingabeparameter verwendet.

Die ausgewählten Testfälle sind mit niedriger Last bis zu hoher Last (von `EchoVoidTest` bis zu `EchoStringTest` mit 128 KB als Daten) parametriert und decken alle zuvor definierten Klassen für Testfälle bis auf die Klasse *Testfälle, die größere Daten mit einen Array übertragen* ab. Es wurde kein Testfall aus

der letzten Klasse ausgewählt, da in diesem Benchmark das Verhalten unter Last von Interesse ist und Tests mit hoher Last schon mittels der `EchoStringTest`s abgedeckt sind.

Um den Zugriff von einer Vielzahl von Clients zu messen, gibt es wie in [Vol04] dargelegt grundsätzlich drei Möglichkeiten:

- von einem einzelnen Prozess (Thread) auf einem Rechner,
- von vielen Anfragen eines multi-threaded Clients oder
- von vielen (virtuellen) Clients auf einem oder mehreren Rechnern.

Für diesen Benchmark wurde die zweite Variante gewählt, d.h. es wird ein multi-threaded Client verwendet, der auf dem gleichen Rechner wie der Server läuft.

Es wurden zwei Testreihen je Server-Variante durchgeführt, zuerst eine mit einem Client der mittels 1, 25, 50, 75 und 100 Threads zumindest 2500 Requests (falls sich keine Ganzzahl bei der Division der Zielrequests durch die Threadanzahl ergibt, wird die nächste größere Ganzzahl für die Anzahl an Requests pro Thread verwendet) absetzt, die zweite mit 1, 2, 4, 8, 16 und 20 Threads.

Dieser Benchmark dient dazu, die Abhängigkeit der Antwortzeiten für einen Methodenabruf von der Anzahl der Threads bzw. simulierten Clients, die quasi-parallel Anfragen an den Server absetzen, bei steigender Last zu messen, da sich die zu testenden Server in der Architektur, wie bzw. ob sie Nebenläufigkeit implementieren, unterscheiden [vgl. Vol04, S. 14].

Der Server, der Thread Pooling verwendet, wurde so parametriert, dass er maximal 100 Threads im Pool aufnimmt und zu Beginn keine gepoolten Threads beinhaltet (*Lazy Acquisition* Pattern).

Die Server, die das *Leader / Followers* und *Half-Sync/Half-Async* Pattern implementieren, besitzen vom Applikationsstart an 100 *Worker* Threads.

Als Think Time werden 50 ms für jeden Client-Thread zwischen den Aufrufen verwendet, um allfällige Starvation-Effekte [vgl. Tan02, S. 143] zu verhindern.

7 Ergebnisse des Benchmarkings und deren Interpretation

In den folgenden Unterkapiteln werden die Ergebnisse der Benchmark Suite dargestellt. Es wird betrachtet, wie sich die mittlere Aufrufdauer unter zunehmender Last und Grad an Nebenläufigkeit entwickelt. Die Beschreibung der Ergebnisse beginnt mit der Darstellung der Ergebnisse für den Bereich 1-100 Threads, dann werden die Ergebnisse für den Bereich 1-20 Threads interpretiert. Die gesammelten Daten sind in aggregierter Form im *Anhang D* aufgelistet, es wurden die minimale, maximale, mittlere (Median) und durchschnittliche Aufrufdauer sowie die Aufrufe pro Sekunde und die Standardabweichung (von der durchschnittlichen Aufrufdauer) aufgezeichnet bzw. berechnet.

7.1 Ergebnisse von ReturnVoidTest

Dieser Test war der einzige Test, der keine Parameter und auch keinen Rückgabewert besaß.

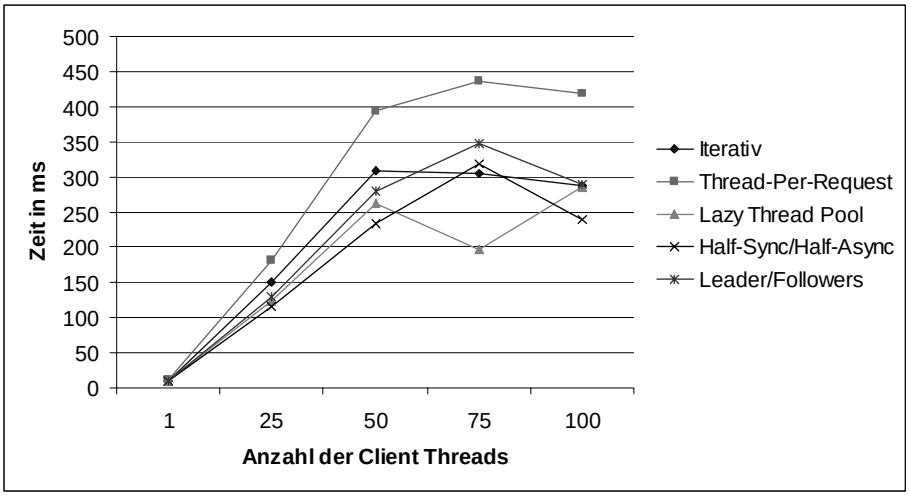


Abbildung 28: Median der Aufrufdauer von ReturnVoidTest im Bereich 1-100 Threads

Die in Abbildung 28 dargestellte Testreihe im Bereich von 1-100 Threads ergab, dass im Falle der Verwendung nur eines Client Threads alle Server-

Varianten bis auf die *Thread-Per-Request*-Variante eine mittlere Aufrufdauer von 10 ms aufwiesen (die *Thread-Per-Request*-Variante besaß mit 11 ms die längste Aufrufdauer). Die Variante mit der geringsten Standardabweichung von 1,53 war die mit dem *Lazy Thread Pool*, danach folgt die *Iterative*-Variante mit 1,63. Die größte Standardabweichung mit 2,22 wies die Variante nach dem *Thread-Per-Request* Pattern auf.

Im Bereich von 25 bis 50 Client Threads besitzt die dem *Half-Sync/Half-Async* Pattern folgende Variante die geringste mittlere Aufrufdauer mit 116 bzw. 234 ms. Mit der zweitgeringsten mittleren Aufrufdauer folgt die *Lazy Thread Pool*-Variante mit 124 bzw. 262 ms, mit der drittgeringsten Aufrufdauer folgt die dem *Leader/Followers* Pattern folgende Variante, danach die *Iterative*-Variante und abgeschlagen die *Thread-Per-Request*-Variante.

Dieses Bild setzt sich bis zu 75 Client Threads fort, wo die *Lazy Thread Pool*-Variante die geringste mittlere Aufrufdauer besitzt.

Bei 100 Client Threads besitzen die *Iterative*, die *Lazy Thread Pool*- und die *Leader/Followers*-Variante beinahe die gleiche mittlere Aufrufdauer von 287, 286 bzw. 290 ms, nur die *Half-Sync/Half-Async*-Variante unterbietet diese mit 239 ms.

Deutlich ist, dass die *Thread-Per-Request*-Variante abgeschlagen bei diesen Szenarien in der mittleren Aufrufdauer liegt.

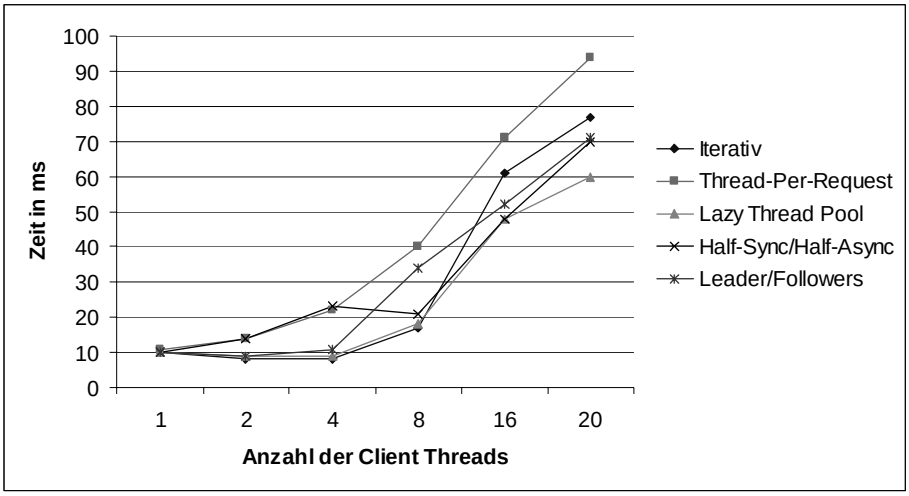


Abbildung 29: Median der Aufrufdauer von ReturnVoidTest im Bereich 1-20 Threads

In der in Abbildung 29 dargestellten Testreihe im Bereich 1-20 Client Threads besitzt die *Iterative*-Variante im Bereich von 2 bis 8 Client Thread die geringste mittlere Aufrufdauer (bei 2 Client Threads 8 ms, bei 4 Client Threads 8 ms und bei 8 Client Threads 17 ms) vor der *Lazy Thread Pool*-Variante, die die zweitgeringste mittlere Aufrufdauer in diesen Szenario aufweist. Im Bereich von 2 bis 4 Client Threads besitzt die *Leader/Followers*-Variante eine geringere Aufrufdauer (bei 2 Client Threads die gleiche wie die *Lazy Thread Pool*-Variante) als die *Half-Sync/Half-Async*-Variante, ab 8 Client Threads ist die mittlere Aufrufdauer der *Half-Sync/Half-Async*-Variante geringer als die der *Leader/Followers*-Variante.

Ab 16 Client Threads besitzen alle multi-threaded Varianten bis auf die dem *Thread-Per-Request* Pattern folgende eine geringere mittlere Aufrufdauer als die *Iterative*.

Auch hier ergibt sich das Bild, dass die *Thread-Per-Request*-Variante mit höherer Client Thread-Anzahl schlechter als alle anderen Varianten abschneidet.

7.2 Ergebnisse von EchoIntegerTest

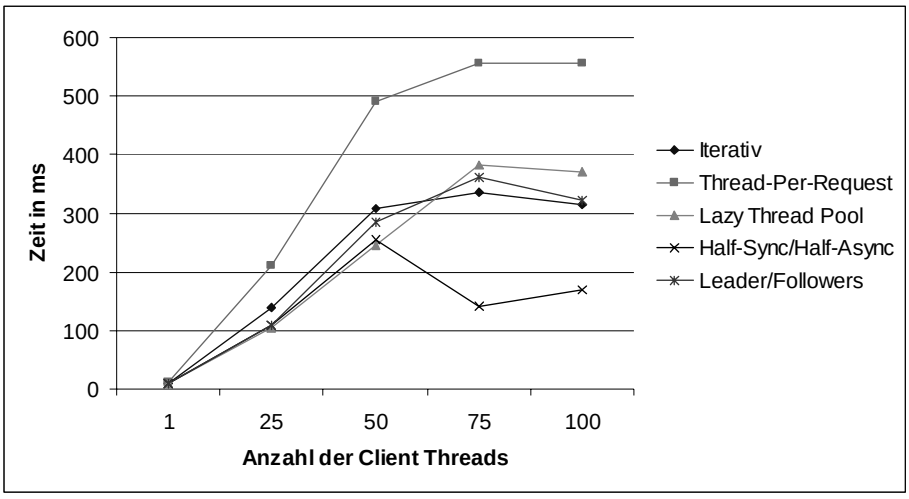


Abbildung 30: Median der Aufrufdauer von EchoIntegerTest im Bereich 1-100 Threads

Auch bei diesem Test zeigte sich, dass die *Thread-Per-Request*-Variante mit höherer Client Thread-Anzahl schlechter als die anderen Varianten abschnitt.

Wie aus Abbildung 30 ersichtlich, ist die Variante mit der geringsten mittleren Aufrufdauer bei bis zu 50 Client Threads die *Lazy Thread Pool*-Variante, ab 75 Client Threads besitzt die *Half-Sync/Half-Async*-Variante die geringste mittlere Aufrufdauer, gefolgt von der *Iterativen*.

Bei 75 und 100 Client Threads besitzen die *Lazy Thread Pool*- und *Leader/Followers*-Varianten eine höhere mittlere Aufrufdauer als die *Iterative*.

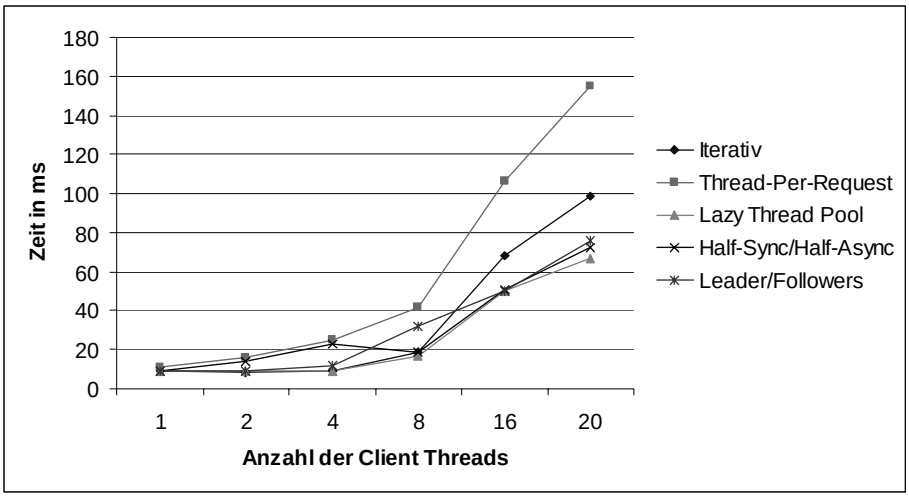


Abbildung 31: Median der Aufrufdauer von EchoIntegerTest im Bereich 1-20 Threads

In der in Abbilung 31 dargestellten Testreihe im Bereich von 1-20 Threads wiederholt sich das aus dem `EchoVoidTest` schon bekannte Bild, mit der Ausnahme, dass die *Iterative* nur im Bereich von 1 bis 4 Client Threads die Variante mit der niedrigsten mittleren Aufrufdauer ist und ab 8 Client Threads die *Lazy Thread Pool*-Variante die mit der niedrigsten ist, wobei die *Lazy Thread Pool*-, die *Half-Sync/Half-Async*- und die *Leader/Followers*-Varianten bei 16 Client Threads beinahe bzw. die gleiche mittlere Aufrufdauer besitzen (*Lazy Thread Pool* und *Leader/Followers* mit 50 ms und *Half-Sync/Half-Async* mit 51 ms).

7.3 Ergebnisse von EchoIntegerArrayTest

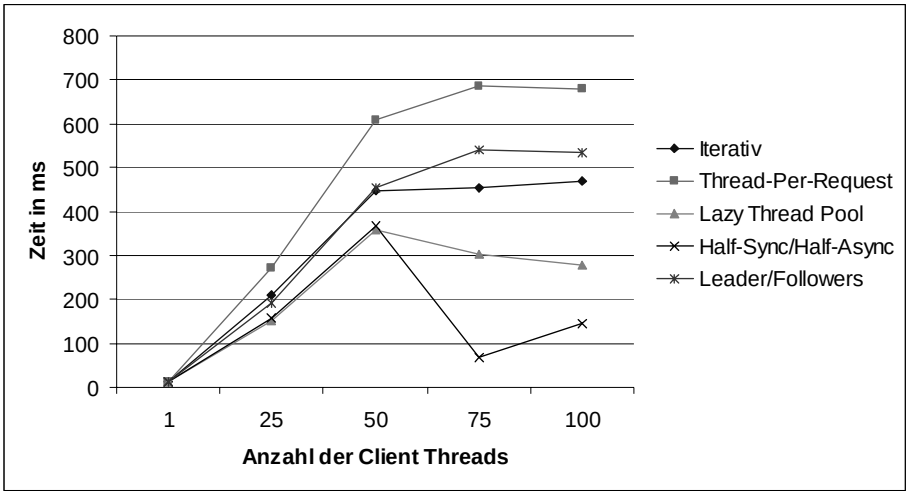


Abbildung 32: Median der Aufrufdauer von EchoIntegerArrayTest im Bereich 1-100 Threads

Bei dieser Testreihe im Bereich 1-100 Threads ist wieder die *Thread-Per-Request*-Variante die mit der höchsten mittleren Aufrufdauer.

Bei 25 und 50 Client Threads besitzt die *Lazy Thread Pool*-Variante die niedrigste mittlere Aufrufdauer knapp vor der *Half-Sync/Half-Async*-Variante (bei 25 Client Threads 151 ms vs. 158 ms, bei 50 357 ms vs. 368 ms), ab 75 Client Threads besitzt die *Half-Sync/Half-Async*-Variante eine geringere mittlere Aufrufdauer als die *Lazy Thread Pool*-Variante.

Die *Leader/Followers*-Variante besitzt ab 50 Client Threads eine höhere mittlere Aufrufdauer als die *Iterative*.

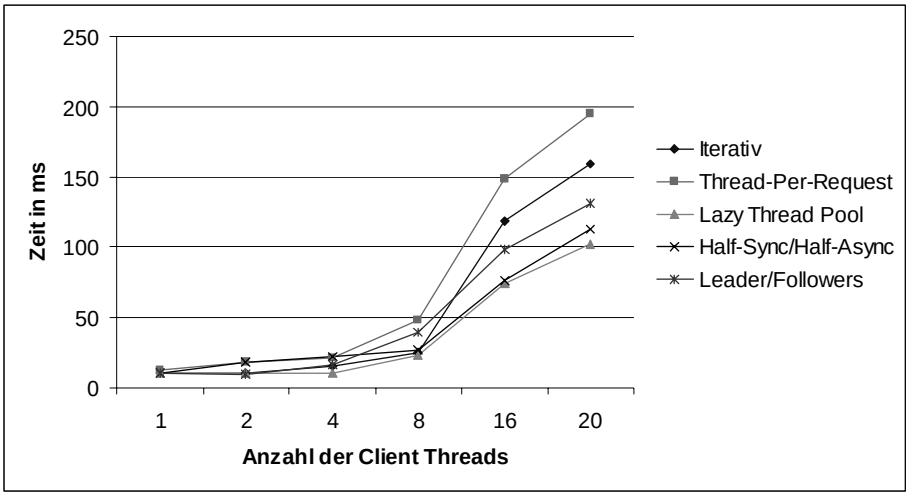


Abbildung 33: Median der Aufrufdauer von EchoIntegerArrayTest im Bereich 1-20 Threads

Die in Abbildung 33 abgebildete Testreihe im Bereich von 1-20 Threads beinhaltete keine Überraschungen. Die *Thread-Per-Request*-Variante ist wie in den anderen Testreihen wieder die mit der höchsten mittleren Aufrufdauer, die *Lazy Thread Pool*-Variante die mit der geringsten.

7.4 Ergebnisse von EchoStructTest

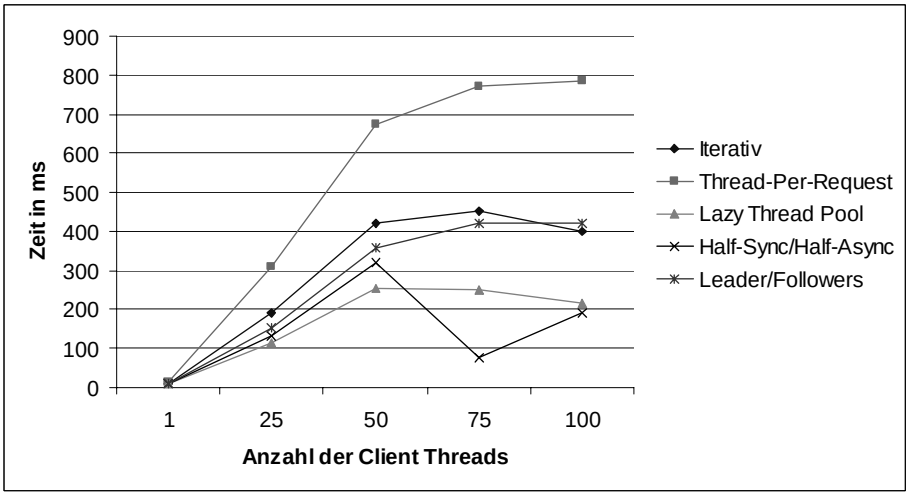


Abbildung 34: Median der Aufrufdauer von EchoStructTest im Bereich 1-100 Threads

In dieser Testreihe (siehe Abbildung 34) im Bereich 1-100 Threads besitzt der *Lazy Thread Pool* im Bereich 2 bis 50 Client Threads die geringste mittlere Aufrufdauer, er wird gefolgt von der *Half-Sync / Half Async*-, der

Leader/Followers-, der *Iterativen*- und schließlich der *Thread-Per-Request*-Variante.

Ab 75 Client Threads besitzt die *Half-Sync/Half-Async*-Variante die geringste mittlere Aufrufdauer. Bei 100 Client Threads ist die mittlere Aufrufdauer für die *Iterative*-Variante mit 399 ms geringfügig geringer als die der *Leader/Followers*-Variante mit 421 ms.

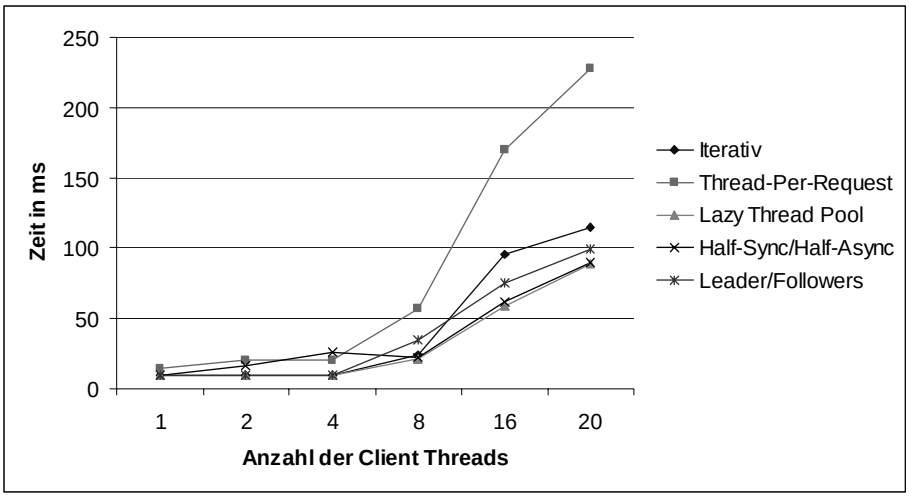


Abbildung 35: Median der Aufrufdauer von EchoStructTest im Bereich 1-20 Threads

Die in Abbildung 35 dargestellte Testreihe im Bereich von 1-20 Client Threads zeigt wieder, dass die *Lazy Thread Pool*-Variante mit der geringsten mittleren Aufrufdauer vor allen anderen Varianten liegt. Deutlich zeigt sich auch hier (insbesondere bei 20 Client Threads) die hohe mittlere Aufrufdauer der Variante nach dem *Thread-Per-Request* Pattern.

7.5 Ergebnisse von EchoStructArrayTest

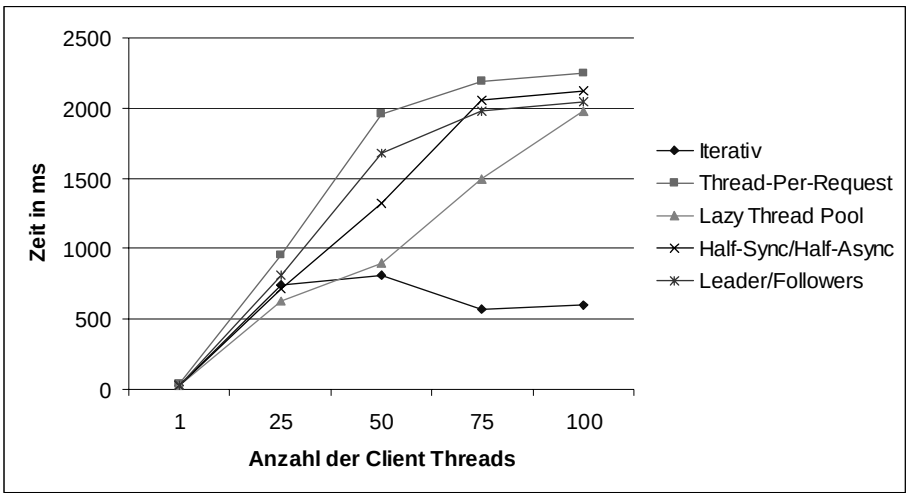


Abbildung 36: Median der Aufrufdauer von EchoStructArrayTest im Bereich 1-100 Threads

Die Testreihe, die in Abbildung 36 dargestellt ist, zeigt als erste Testreihe, dass bei hoher Last und hoher Anzahl an Client Threads die mittlere Aufrufdauer der *Iterativen*-Variante am niedrigsten ist.

Ansonsten zeigte sich nur das gewohnte Bild, dass die *Lazy Thread Pool*-Variante die Variante mit der geringsten mittleren Aufrufdauer der multi-threaded Server ist. Abweichend von den vorhergehenden Testreihen, besitzt die *Leader/Followers*-Variante ab 75 Client Threads eine geringere mittlere Aufrufdauer als die *Half-Sync/Half-Async*-Variante.

Im Vergleich zum `EchoStructTest` zeigte sich die *Thread-Per-Request*-Variante nicht im gleichen Ausmaß unvorteilhaft gegenüber den anderen multi-threaded Servern.

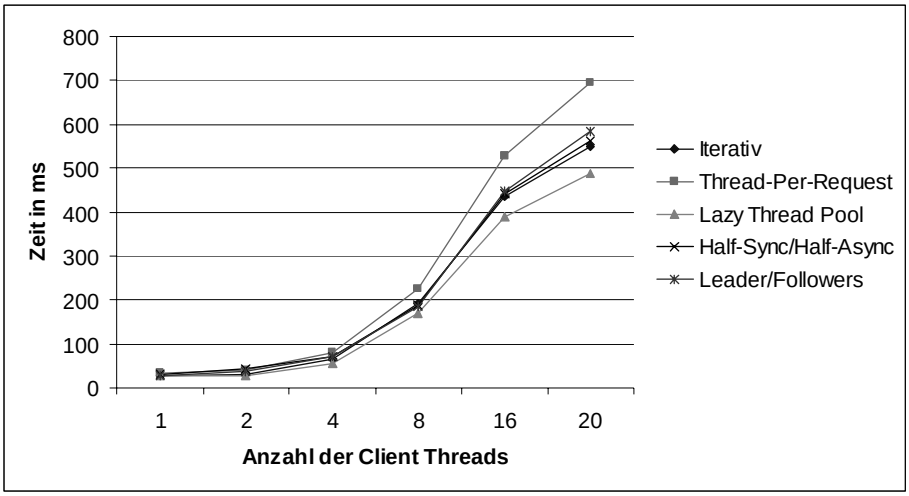


Abbildung 37: Median der Aufrufdauer von EchoStructArrayTest im Bereich 1-20 Threads

In der Testreihe mit 1-20 Client Threads (siehe Abbildung 37) zeigte sich die *Lazy Thread Pool*-Variante als die mit der geringsten mittleren Aufrufdauer und ab 16 Client Threads besitzt die *Iterative* Variante die zweitgeringste mittlere Aufrufdauer. Die *Half-Sync/Half-Async*- und die *Leader/Followers*-Variante folgen mit etwas höheren mittleren Aufrufdauern (Bei 20 Client Threads 549 ms für die *Iterative*, 563 ms für die *Half-Sync/Half-Async*-Variante und 583 ms für die *Leader/Followers*-Variante). Wie auch bei den anderen Tests, weist die *Thread-Per-Request*-Variante die höchste mittlere Aufrufdauer auf.

7.6 Ergebnisse von EchoStringTest

Im Rahmen dieses Tests wurden die Server-Varianten mit Datenfiles von 2 KB bis 128 KB (gefüllt mit alphanumerischen Zeichen, die in UTF-8 enkodiert sind) als Eingabeparameter für die Methode `echoString`, die von dieser wieder als Rückgabewert geliefert werden, verwendet.

7.6.1 Ergebnisse für 2 KB Daten

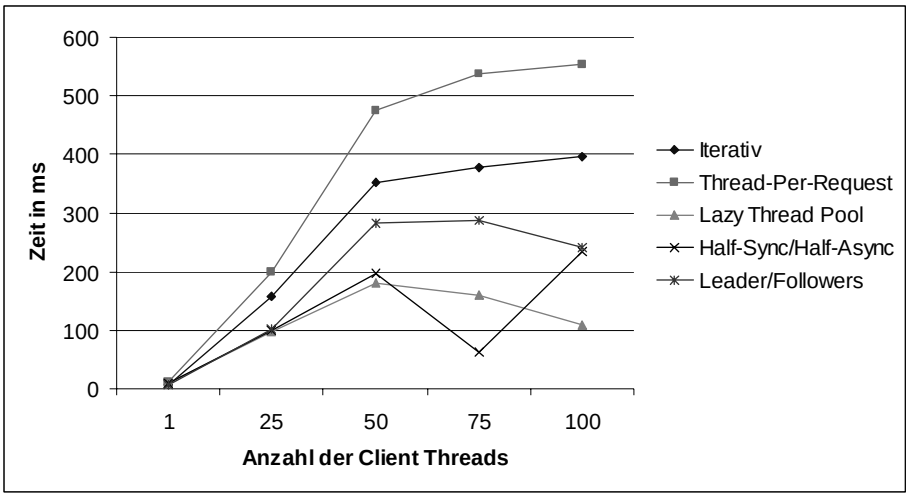


Abbildung 38: Median der Aufrufdauer von EchoStringTest mit 2 KB Text im Bereich 1-100 Threads

Für die in Abbildung 38 dargestellten Testreihe im Bereich von 1-100 Client Threads, zeigt sich wie bei den vorhergehenden Tests, dass die *Thread-Per-Request*-Variante die höchste mittlere Aufrufdauer besitzt. Die anderen multi-threaded Varianten besitzen eine niedrigere mittlere Aufrufdauer als die *Iterative*.

Bis auf den Test mit 75 Client Threads besitzt die *Lazy Thread Pool*-Variante die geringste mittlere Aufrufdauer gefolgt von der *Half-Sync/Half-Async*- und der *Leader/Followers*-Variante. Beim Test mit 75 Client Threads besitzt die *Half-Sync/Half-Async*-Variante die geringste mittlere Aufrufdauer, dies kann möglicherweise auf einen Ausreißerwert hinweisen.

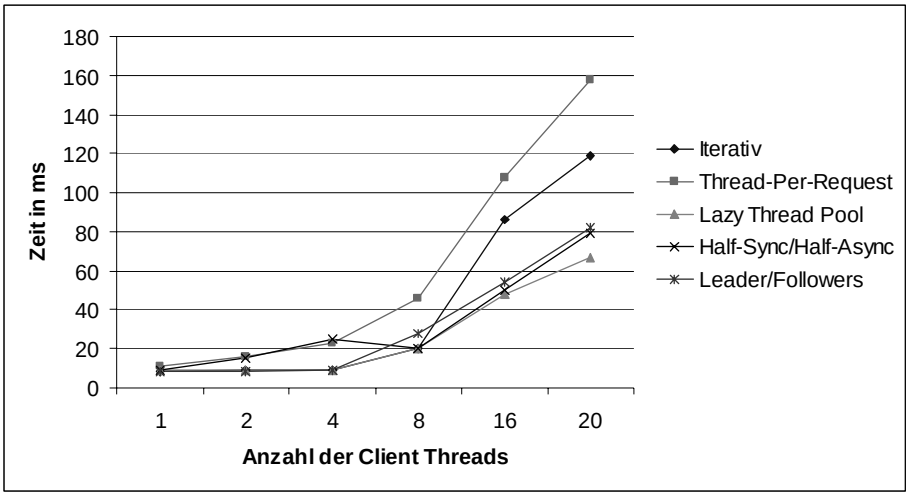


Abbildung 39: Median der Aufrufdauer von EchoStringTest mit 2 KB Text im Bereich 1-20 Threads

In der in Abbildung 39 dargestellten Testreihe im Bereich von 1-20 Client Threads, besitzen die *Iterative*-, *Lazy Thread Pool*- und *Leader/Followers*-Variante jeweils eine mittlere Aufrufdauer von 8-9 ms, nur die mittlere Aufrufdauer der *Thread-Per-Request*- und *Half-Sync/Half-Async*-Variante befinden sich auf deutlich höheren Niveau (bei 4 Client Threads 23 bzw. 25 ms).

Mit steigender Client Thread-Anzahl ergibt sich das gewohnte Bild, dass die *Lazy Thread Pool*-Variante vor der *Half-Sync/Half-Async*-Variante, gefolgt von der *Leader/Followers*- und der *Iterativen*-Variante liegt. Auch hier ist die Variante mit der höchsten mittleren Aufrufdauer die *Thread-Per-Request*-Variante.

7.6.2 Ergebnisse für 4 KB Daten

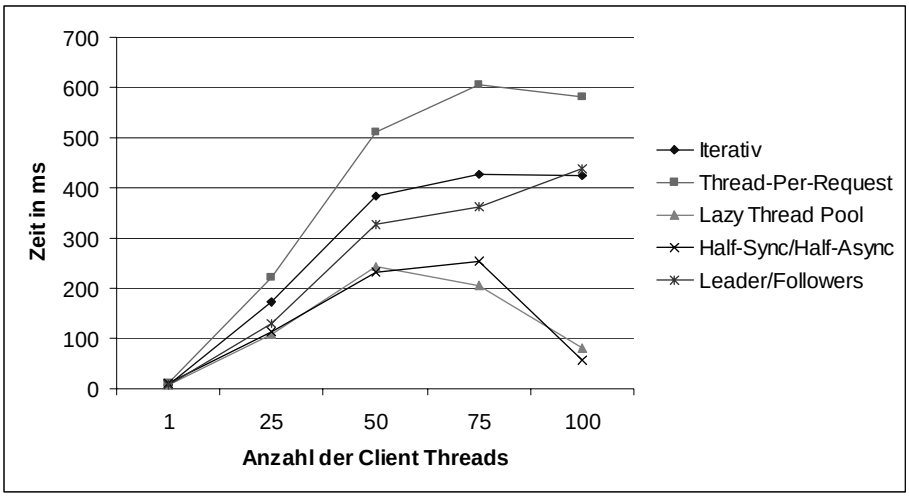


Abbildung 40: Median der Aufrufdauer von EchoStringTest mit 4 KB Text im Bereich 1-100 Threads

Auch in der in Abbildung 40 dargestellten Testreihe ist die *Thread-Per-Request*-Variante die mit der höchsten mittleren Aufrufdauer. Die *Lazy Thread Pool*- und die *Half-Sync/Half-Async*-Variante sind wieder die Varianten mit der geringsten mittleren Aufrufdauer. Diese werden von der *Leader/Followers*-Variante gefolgt, die bei 100 Client Threads eine höhere mittlere Aufrufdauer als die Iterative aufweist.

Überraschend sind die niedrigen mittleren Aufrufdauern der *Lazy Thread Pool*- und der *Half-Sync/Half-Async*-Variante von 82 bzw. 56 ms bei 100 Client Threads.

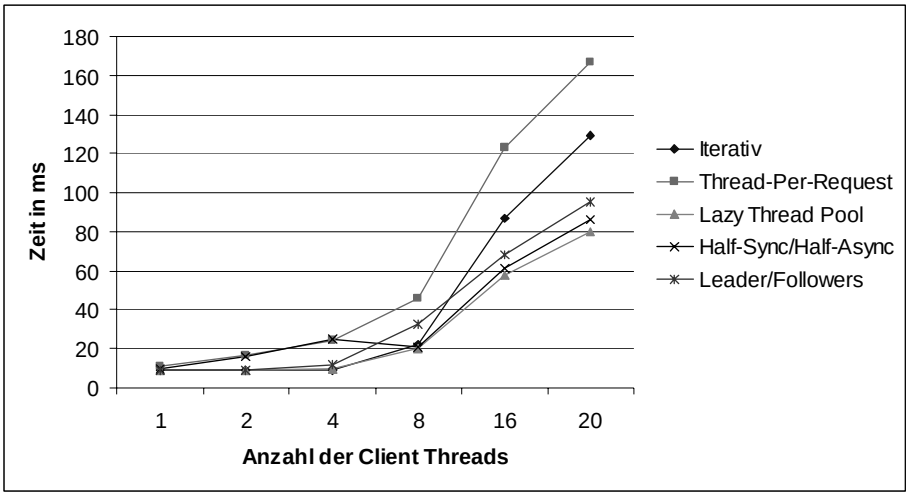


Abbildung 41: Median der Aufrufdauer von EchoStringTest mit 4 KB Text im Bereich 1-20 Threads

Die Testreihe im Bereich von 1-20 Client Threads (siehe Abbildung 41) zeigt sich frei von Überraschungen und gibt das bekannte Bild wieder, dass die niedrigste mittlere Aufrufdauer von der *Lazy Thread Pool*-Variante erreicht wurde.

7.6.3 Ergebnisse für 8 KB Daten

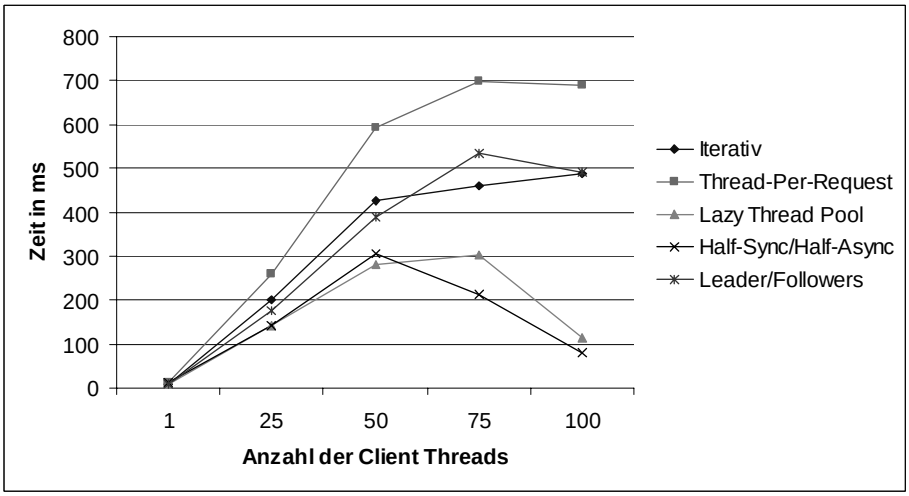


Abbildung 42: Median der Aufrufdauer von EchoStringTest mit 8 KB Text im Bereich 1-100 Threads

Wie schon bei den *Ergebnisse für 4 KB Daten* zeigt sich bei dieser in Abbildung 42 dargestellten Testreihe mit 1-100 Client Threads, dass mit dieser

Parametrierung die *Lazy Thread Pool*- und die *Half-Sync/Half-Async*-Varianten die geringste mittlere Aufrufdauer aufweisen.

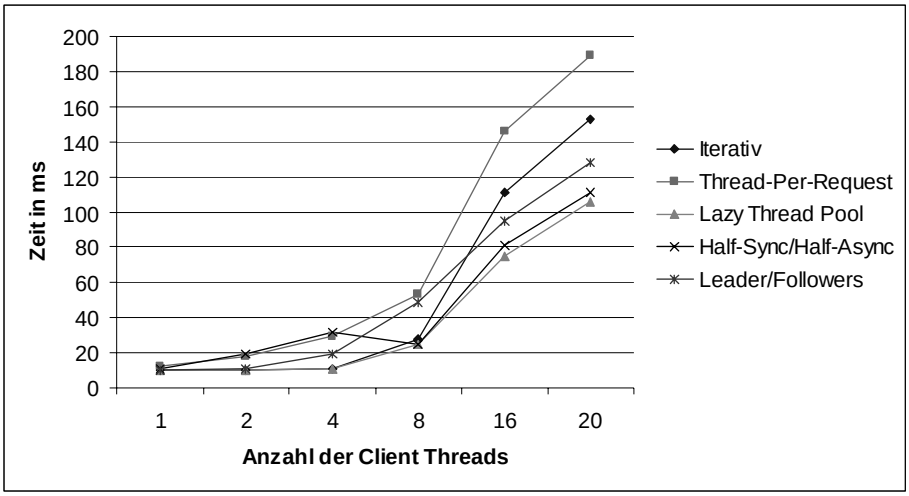


Abbildung 43: Median der Aufrufdauer von EchoStringTest mit 8 KB Text im Bereich 1-20 Threads

Auch im Bereich von 1-20 Client Threads (siehe Abbildung 43) zeigte sich mit steigender Client Threads-Anzahl, dass die *Lazy Thread Pool*- und *Half-Sync/Half-Async*-Variante die geringsten mittleren Aufrufdauern aufweisen.

7.6.4 Ergebnisse für 16 KB Daten

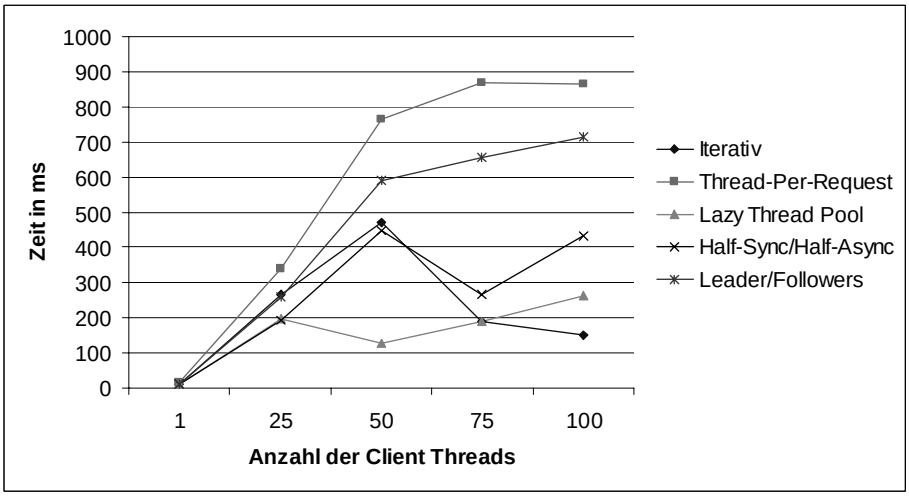


Abbildung 44: Median der Aufrufdauer von EchoStringTest mit 16 KB Text im Bereich 1-100 Threads

In dieser in Abbildung 44 dargestellten Testreihe mit 1-100 Client Threads zeigte sich das schon in *Ergebnisse von EchoStructArrayTest* vorzufindende Bild, dass nämlich bei hoher Last und hoher Client Thread-Anzahl die mittlere Aufrufdauer der *Iterativen* Variante geringer als die der multi-threaded Varianten ist, dieser Effekt tritt unter dieser Parametrierung ab 75 Client Threads auf (wobei bei 75 Client Threads die *Iterative*-Variante noch knapp hinter der *Lazy Thread Pool*-Variante liegt (191 ms vs. 188 ms)).

Die Ergebnisse für die multi-threaded Varianten spiegeln ansonsten nur das gewohnte Bild wider, nämlich dass die *Lazy Thread Pool*-Variante die mit geringsten mittleren Aufrufdauer vor der *Half-Sync/Half-Async*-, der *Leader/Followers*- und schließlich der *Thread-Per-Request*-Variante ist.

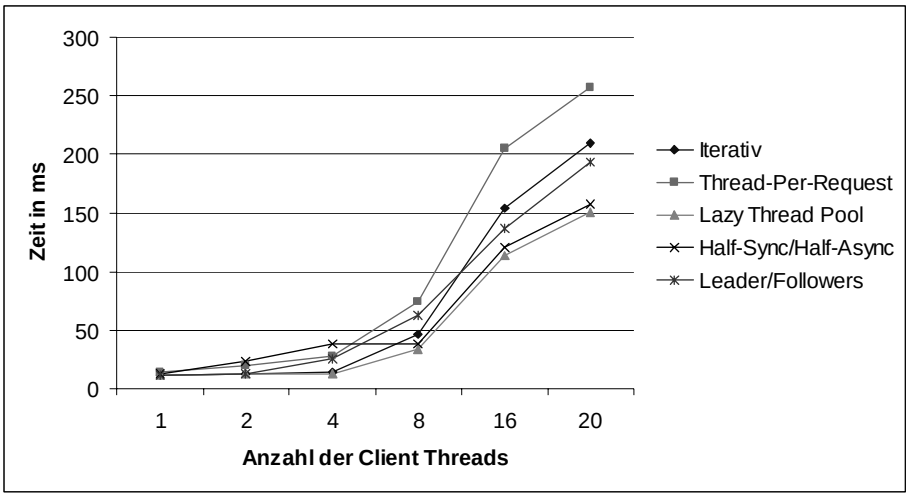


Abbildung 45: Median der Aufrufdauer von EchoStringTest mit 16 KB Text im Bereich 1-20 Threads

In der in Abbildung 45 illustrierten Testreihe im Bereich von 1-20 Client Threads ist wiederum kein außergewöhnliches Ergebnis zu vermerken. Es zeigt sich wieder, dass die *Lazy Thread Pool*-Variante, vor der *Half-Sync/Half-Asyn*-Variante, gefolgt von der *Leader/Followers*-Variante, liegt. Wiederum ist die *Thread-Per-Request*-Variante die mit der höchsten mittleren Aufrufdauer.

7.6.5 Ergebnisse für 32 KB Daten

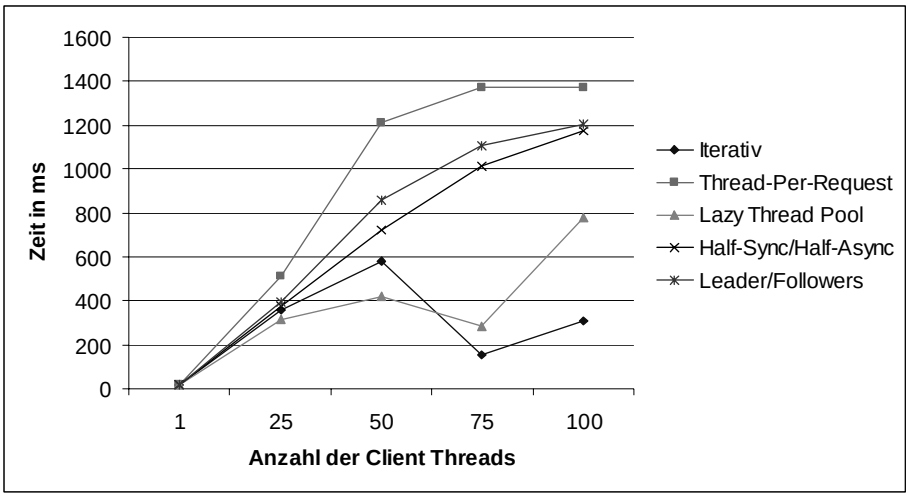


Abbildung 46: Median der Aufrufdauer von EchoStringTest mit 32 KB Text im Bereich 1-100 Threads

Die Ergebnisse der in Abbildung 46 illustrierten Testreihe mit 1-100 Client Threads zeigen ähnlich zu *Ergebnisse für 16 KB Daten* den Effekt, dass die *Iterative*-Variante bei hoher Last und hoher Client Threads-Anzahl die geringste mittlere Aufrufdauer aufweist (hier ab 75 Client Threads). Wieder ist die *Lazy Thread Pool*-Variante die multi-threaded Variante mit der geringsten mittleren Aufrufdauer. Interessant ist, dass die *Half-Sync/Half-Async*-Variante bei 100 Client Threads eine beinahe gleich hohe mittlere Aufrufdauer wie die *Leader/Followers*-Variante besitzt.

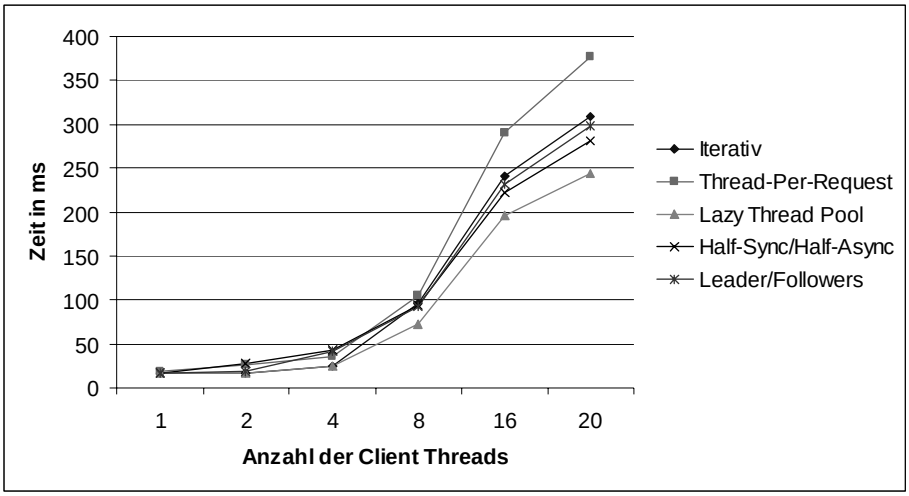


Abbildung 47: Median der Aufrufdauer von EchoStringTest mit 32 KB Text im Bereich 1-20 Threads

Die Testreihe mit 1-20 Client Threads (siehe Abbildung 47) weist wiederum keine außergewöhnlichen Ergebnisse auf, die Variante mit der niedrigsten mittleren Aufrufdauer ist wieder die *Lazy Thread Pool*-Variante.

7.6.6 Ergebnisse für 64 KB Daten

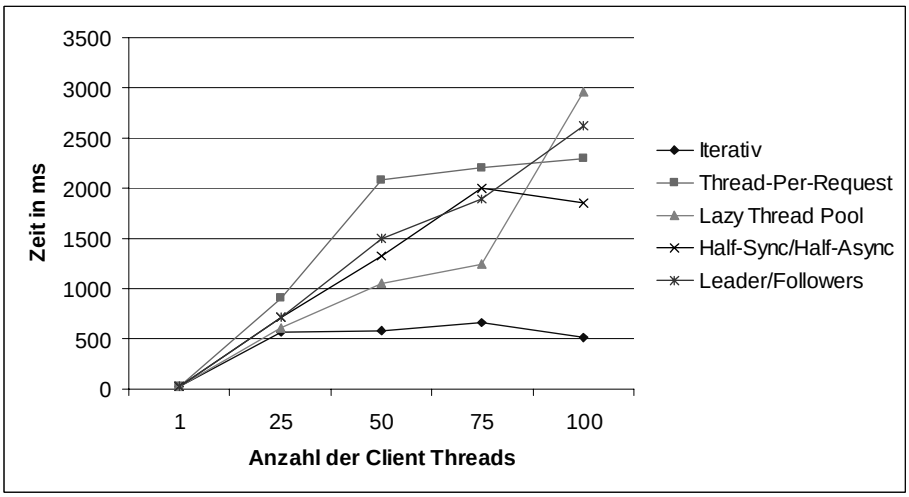


Abbildung 48: Median der Aufrufdauer von EchoStringTest mit 64 KB Text im Bereich 1-100 Threads

Interessant an der in Abbildung 48 dargestellten Testreihe im Vergleich zu den vorhergehenden (außer dass die *Iterative*-Variante die niedrigste mittlere Aufrufdauer aufweist und dies schon ab 25 Client Threads auftritt) ist, dass bei 100 Client Threads die *Lazy Thread Pool*- und die *Leader/Followers*-Variante

eine höhere mittlere Aufrufdauer als die *Thread-Per-Request*-Variante besitzen.

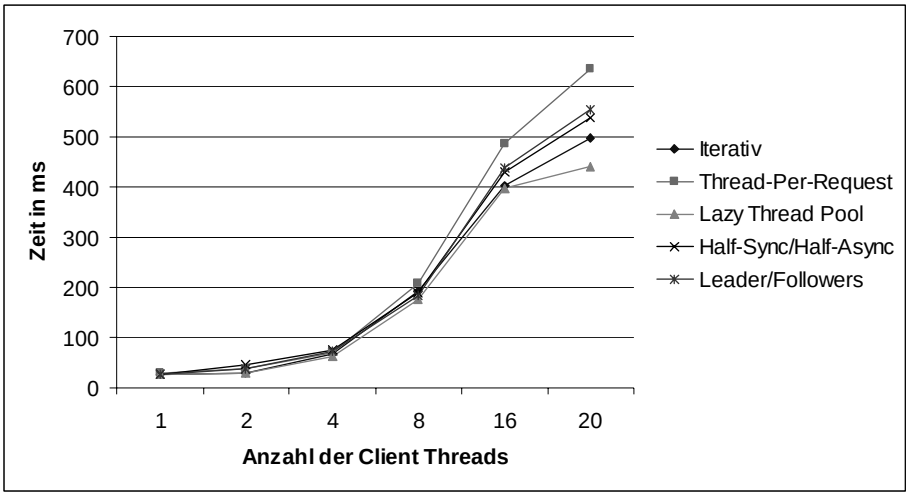


Abbildung 49: Median der Aufrufdauer von EchoStringTest mit 64 KB Text im Bereich 1-20 Threads

Auch in der in Abbildung 49 dargestellten Testreihe mit 1-20 Client Threads ist eine Abweichung zum gewohnten Bild festzustellen, nämlich, dass diesmal die *Half-Sync/Half-Async*- und *Leader/Followers*-Variante eine höhere mittlere Aufrufdauer bei 20 Client Threads als die *Iterative* aufweist.

7.6.7 Ergebnisse für 128 KB Daten

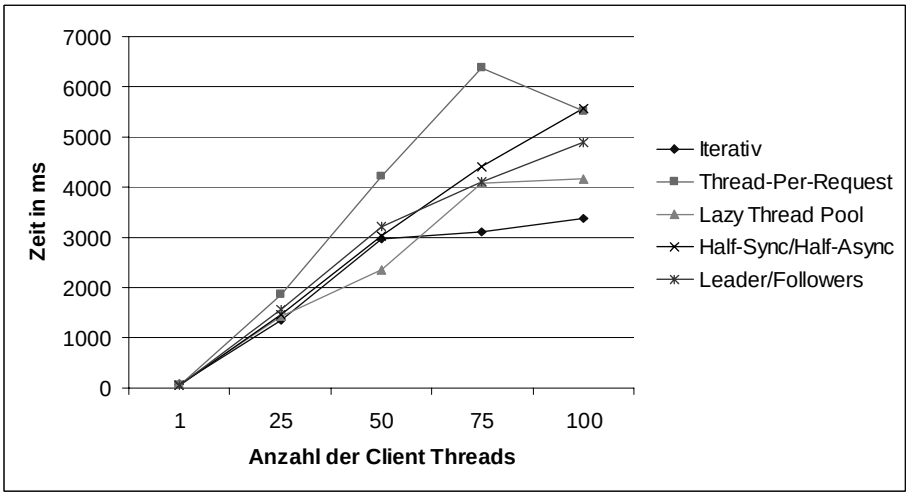


Abbildung 50: Median der Aufrufdauer von EchoStringTest mit 128 KB Text im Bereich 1-100 Threads

Die in der Abbildung 50 illustrierte Testreihe im Bereich von 1-100 Client Threads weist ein ähnliches Ergebnis wie in *Ergebnisse für 64 KB Daten* auf, außer dass diesmal nur die *Half-Sync/Half-Async*-Variante bei 100 Client Threads eine etwas höhere mittlere Aufrufdauer als die *Thread-Per-Request*-Variante aufweist (5557 ms vs. 5514 ms).

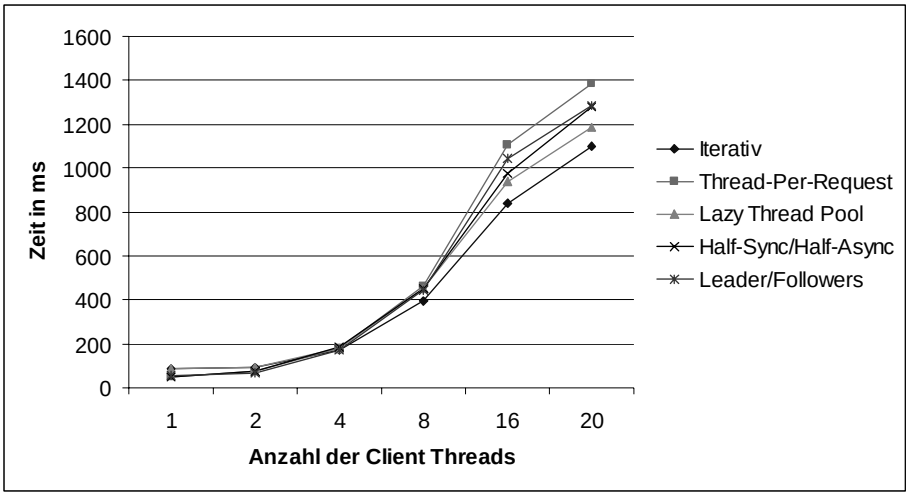


Abbildung 51: Median der Aufrufdauer von EchoStringTest mit 128 KB Text im Bereich 1-20 Threads

Die letzte durchgeführte Testreihe im Bereich von 1-20 Client Threads (siehe Abbildung 51) weist ein ähnliches Ergebnis wie in *Ergebnisse für 64 KB Daten* auf, nur dass die *Iterative*-Variante schon ab 4 Client Threads die mit der niedrigsten mittleren Aufrufdauer ist.

7.7 Interpretation der Ergebnisse

Bei der Testdurchführung zeigte sich, dass die Benutzung der *Thread-Per-Request*-Variante in diesen Testszenarien, außer bei `EchoStringTest` mit 64 KB und 128 KB beide im Bereich 1-100 Client Threads, immer die Variante mit der höchsten mittleren Aufrufdauer war.

Es zeigte sich schon beim `ReturnVoidTest` mit nur einem Client Thread, dass die *Thread-Per-Request*-Variante die höchste mittlere Aufrufdauer besaß, bei allen anderen Varianten war die mittlere Aufrufdauer gleich (die durchschnittliche aber höher) als die der *Iterativen*, was den Schluss nahe

legt, dass der Overhead der Verwaltung bzw. Erzeugung von Threads gegenüber der *Iterativen*-Variante sich in Grenzen hält.

In der Mehrzahl der Fälle zeigte sich unter zunehmender Last und Grad an Nebenläufigkeit, dass die *Lazy Thread Pool*-Variante niedrigere Aufrufdauern als die *Half-Sync/Half-Async*-Variante aufwies, die wiederum vor der *Leader/Followers*-Variante lag. Die drei vorher aufgezählten Varianten wiesen in der Mehrzahl der Fälle (bis auf `EchoStructArrayTest` und `EchoStringTest` mit 64 KB und 128 KB) geringere mittlere Aufrufdauern als die *Iterative* auf.

Interessant war, dass die *Leader/Followers*-Variante in der Mehrzahl der Fälle die zweitlangsamste multi-threaded Variante (vor der *Thread-Per-Request*-Variante) war, was vermutlich daran lag, dass das in [POSA2] beschriebene Pattern und dessen Implementierung direkt das API bzw. Systemaufrufe des zugrundeliegenden Betriebssystems benutzt, aber die getestete Variante in einer Umgebung lief, die eine Abstraktionsebene über dem Betriebssystem liegt (d.h. konkret eine Virtual Machine mit Garbage Collection).

Abschließend kann gesagt werden, dass sich im Benchmarking die *Lazy Thread Pool*-Variante gefolgt von der *Half-Sync/Half-Async*-Variante als die Varianten mit der niedrigsten mittleren Aufrufdauer in den meisten Szenarien darstellten.

Die genauen Gründe für das schlechte Abschneiden der *Leader/Followers*-Variante bedürfen einer genaueren Untersuchung (vor allem wie die Virtual Machine und die Laufzeitumgebung Multi-threading umsetzen), die den Rahmen dieser Arbeit sprengen würde.

8 Verwandte Arbeiten

In [Vol04] wird eine ähnliche Vorgehensweise wie in dieser Arbeit zum Benchmarking von Web Services vorgestellt, die aber zusätzlich Taktfrequenzen von CPUs und auch die Größe der übertragenen TCP-Pakete berücksichtigt.

Die TPC Benchmarks [TPC] im OLTP-Bereich messen Datenbank-Transaktionen; z.B. im TPC-C Benchmark wird die sich ergebende Maßzahl auf Geldeinheiten umgelegt. Derzeit ist ein Benchmark von der TPC für Application Server, die Web Services unterstützen, in Arbeit. Dieser wird den Name TPC-App tragen und wird ein Makrobenchmark sein.

Um die Interoperabilität zwischen verschiedenen Implementierungen sicherzustellen, hat die Web Services Interoperablity Organization unter anderem eine Supply Chain Management Applikation spezifiziert, die sich gut als Basis für Makrobenchmarks eignet [vgl. WS-I].

Für die Realisierung von Mikrobenchmarks werden häufig, wie auch in dieser Arbeit, die im W3C-Dokument „*SOAP Version 1.2 Specification Assertions and Test Collection*“ im Kapitel „*3.4 RPC Methods / Procedures Used by the Test Collection*“ festgelegten Methoden [vgl. W3C03b] verwendet.

In [GPRS99] wurde für TAO einem Real-Time CORBA ORB einige Optimierungsprinzipien entwickelt (u.a. das für den Common Case optimiert werden soll, den Berechnungszeitpunkt durch Preprocessing vorzuziehen und weitere ähnliche Prinzipien). Des Weiteren wurden in diesem Paper eine Implementierung dem *Leader/Followers* Pattern folgend und eine mit Queuing (ähnlich zum *Half-Sync/Half-Async* Pattern) im Bezug auf die Performance verglichen. Die Auswirkungen von einigen Algorithmen und Datenstrukturen (wie z.B. diverse Hashing-Algorithmen, lineare und binäre Suche), für das Demultiplexing sind auch Themen in diesem Paper.

Das Design und Performancemessungen für ein Framework für die Übertragung von medizinischen Bilddaten sind das Thema in [HPS96]. Die

Hauptkomponenten dieses Frameworks nutzen das *Strategy*, *Factory*, *Proxy* und *Bridge* Pattern aus [GHJV96]. Um dieses Framework einen Performance-Test zu unterziehen wurden Daten im Bereich von 1 MB bis zu 32 MB via TCP/IP übertragen.

Patterns (u.a. das *Thread-Per-Request* Pattern, *Thread Pooling* etc.) für konkurrente Server werden in [PeSo97] untersucht und einem Performance-Test unterzogen.

In [SmWi98] wird das Software Performance Engineering (am Beispiel von CORBA) behandelt, einem systematischen und quantitativen Ansatz um Software zu entwickeln, die Constraints im Bereich der Performance unterworfen ist.

9 Schlussfolgerungen

In dieser Arbeit wurde ein Konzept zum Testen von Web Service Frameworks erarbeitet, für den Axis Server verschiedene Architektur-Patterns implementiert und eine Test Suite inklusive eines generischen Performance-Test-Frameworks entwickelt.

Es konnte in dieser Arbeit gezeigt werden wie die Performance eines Servers durch die Implementierung unterschiedlicher Architektur-Patterns beeinflusst werden kann.

Bei dieser Arbeit stellte sich das Benchmarking schwieriger als erwartet dar, da das Zielsystem eine sogenannte „Managed Environment“ war, im konkreten Fall eine Java Virtual Machine mit deren Laufzeitumgebung, d.h. das Annahmen, insbesondere solche die in der POSA-Reihe gemacht wurden, die sehr systemnah waren, da in den Beispielimplementierungen direkt Systemaufrufe vorgenommen wurden, so nicht unbedingt gültig sein müssen, da die Virtual Machine wieder eine zusätzliche Abstraktionsebene darstellt.

Weitere aufgeworfene Fragen, die noch der Beantwortung harren, sind der Einfluss von Optimierungen und des Speichermanagements (z.B. Garbage Collection, Heap-Größe etc.) auf die Performance der implementierten Architektur-Patterns.

Des Weiteren ist es empfehlenswert die Testmethodik, die in dieser Arbeit angewandt wurde, zu erweitern bzw. zu verfeinern; indem Mehrprozessormaschinen als Server und auch ein Netzwerk mit mehreren Client-Maschinen genutzt wird, um ein realistischeres Setting zu ermöglichen.

10 Literaturverzeichnis

[Alex79] Alexander, Christopher: *The Timeless Way of Building*. Oxford University Press, New York 1979

[BCK98] Bass, Len; Clements, Paul; Kazman, Rick: *Software Architecture in Practice*. Addison Wesley Longman, Inc. 1998

[Bloc02] Bloch, Joshua: *Effektiv Java programmieren*. Addison-Wesley, 2002

[Cope92] Coplien, J.O.: *Advanced C++ - Programming Styles and Idioms*, Addison-Wesley, 1992

[EbFi03] Eberhart, Andreas; Fischer, Stefan: *Web Services: Grundlagen und praktische Umsetzung mit J2EE und .NET*. Carl Hanser Verlag München Wien, 2003

[GHJV96] Gamma, Erich et al: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley-Longman, Bonn 1996

[GPRS99] Gokhale, Aniruddha et al: *Applying Optimization Principle Patterns to Real-time ORBs*. 5th USENIX Conference on OO Technologies and Systems (COOTS'99) 1999

[GrKoZu04] Zuser, Wolfgang; Grechenig, Thomas; Köhle, Monika: *Software Engineering mit UML und dem Unified Process, 2. Auflage*. Pearson Studium, 2004

[GrTa03] Gröne, Bernhard; Tabeling, Peter: *A System for Concurrent Request Processing Servers*. VikingPLoP 2003

[Herz03] Herzner, Wolfgang: *Message Queues – Three Patterns for Asynchronous Information Exchange.* EuroPLoP 2003

[HPS96] Harrison, Timothy H.; Pyarali, Irfan; Schmidt, Douglas C.: *Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging*. In Computing Systems Journal, USENIX, Vol. 9, No. 3 (1996)

[JLS00] Gosling, James et al: *The Java Language Specification, Second Edition.* http://java.sun.com/docs/books/jls/, Abruf am 2004-09-16

[KeWi00] Kesselman, Jeff; Wilson, Steve: *Java Platform Performance: Strategies and Tactics.* Sun Microsystems Inc., 2000

[KoSt05] Koschel, Arne; Starke, Gernot: *Serviceorientierte Architekturen (SOA)*. In OBJEKTspektrum Ausgabe 3 (2005) S.18 - 19

[Kru02] Krüger, Michael: *Handbuch der Java-Programmierung, 3. Auflage*. Addison-Wesley 2002

[Mey92] Meyers, Scott: *Effective C++ - 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992

[MuWa98] Muller, Hans; Walrath, Kathy: *Threads and Swing.* http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html, Abruf am 2004-09-19

[OMG04] O.A.: *UML™ Resource Page.* http://www.omg.org/technology/uml/, Abruf am 2004-09-16

[PeSo97] Petriu, Dorina; Somadder, Gurudas: *A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers*. EuroPLoP 1997

[POSA1] Buschmann, Frank et al: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996

[POSA2] Schmidt, Douglas et al: *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000

[POSA3] Kirchner, Michael; Jain, Prashant: *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. John Wiley & Sons, 2004

[RFC2616] Fielding et al: *Hypertext Transfer Protocol -- HTTP/1.1*. http://www.w3.org/Protocols/rfc2616/rfc2616.html, Juni 1999, Abruf am 2004-11-06

[SmWi98] Smith, Connie U.; Williams, Lloyd G.: *Performance Engineering Models of CORBA-based Distributed-Object System*. http://www.perfeng.com/papers/corba.pdf, 1998, Abruf am 2005-05-25

[Spec05] O.A.: *SPECjAppServer2004 Design Document*. http://www.spec.org/jAppServer2004/docs/DesignDocument.html, Januar 2005, Abruf am 2005-03-06

[StTa02] van Steen, Maarten; Tanenbaum, Andrew S.: *Distributed Systems: Principles and Paradigms*. Prentice-Hall Inc., 2002

[Tan02] Tanenbaum, Andrew S.: *Moderne Betriebssysteme, 2. Auflage*. Pearson Studium, 2002

[TPC] Transaction Processing Performance Council: *TPC Benchmarks*. http://www.tpc.org, Abruf am 2005-04-17

[UML04] Jeckle, Mario et al: *UML 2 glasklar.* Carl Hanser Verlag München Wien, 2004

[Vol04] Volmer, Dominik: *Benchmarking von Web-Services.* In JavaSPEKTRUM Ausgabe 4 (2004) S.12 - 17

[W3C01] World Wide Web Consortium: *XML Schema Part 0: Primer*. http://www.w3.org/TR/xmlschema-0/, 2001-05-02, Abruf am 2004-09-09

[W3C03a] World Wide Web Consortium: *SOAP Version 1.2 Part 0: Primer*. http://www.w3.org/TR/soap12-part0/, 2003-06-24, Abruf am 2004-09-03

[W3C03b] World Wide Web Consortium: *SOAP Version 1.2 Specification Assertions and Test Collection*. http://www.w3.org/TR/soap12-testcollection/, 2003-06-24, Abruf am 2004-09-03

[W3C04a] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Third Edition)*. http://www.w3.org/TR/REC-xml/, 2004-02-04, Abruf am 2004-09-09

[W3C04b] World Wide Web Consortium: *Web Services Architecture*. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211, Abruf am 2005-02-06

[W3C04c] World Wide Web Consortium*: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. http://www.w3.org/TR/2004/WD-wsdl20-primer-20041221/, Abruf am 2005-02-14

[Web93] O.A.: *Webster's New Encyclopedic Dictionary.* Merriam-Webster Inc., 1993

[WS-I] Web Services Interoperability Organization. http://www.ws-i.org, Abruf am 2005-04-17

Anhang A: Die CD-ROM

Die beigelegte CD-ROM enthält die Diplomarbeit als PDF-Datei, die Sourcen und die ausführbaren Applikationen (Server-Varianten und *Test Suite*-Applikation).

Die Diplomarbeit ist im Root Verzeichnis unter dem Namen `da.pdf` zu finden, die Sourcen als ZIP gepackt im Archiv `source.zip` und die Applikationen im Verzeichnis `ready_to_run`.

Wobei im Verzeichnis `ready_to_run/axis_server` die Server-Varianten liegen und mittels entsprechenden Startskripten (`start_...sh` (für Unix) bzw. `start_...bat` (für Windows)) von dort aus gestartet werden können.

Die *Test Suite-Applikation* findet sich im Verzeichnis `ready_to_run/test_suite`, dieses Verzeichnis sollte auf einen Datenträger mit Schreib-/Lesezugriff kopiert werden, da im Verzeichnis `ready_to_run/test_suite/results` die Ergebnisse eines Testlaufs abgelegt werden. Gestartet kann die Applikation mittels des Skripts `start_test_suite.sh` (für Unix) bzw. `start_test_suite.bat` (für Windows) werden.

Anhang B: Start und Parametrierung der Server

Um den *SimpleAxisServer* zu starten, muss in das Verzeichnis gewechselt werden wo sich die JARs der modifizierten Versionen befinden (z.B. im *lib* Verzeichnis), danach kann der Server mit folgender Zeile gestartet werden:

```
java -cp ./MODIFICATION_JAR:$CLASSPATH \
org.apache.axis.transport.http.SimpleAxisServer -t
```

wobei `MODIFCATION_JAR` der Platzhalter für das zu verwendende JAR ist (z.B.: *axis_thread_pool.jar*). Soll die unmodifizierte Version des *SimpleAxisServer*s verwendet werden, so muss nur der Teil `./MODIFICATION_JAR` weggelassen werden.

Falls Windows verwendet wird, muss der Classpath-Teil wie folgt geändert werden:

```
... -cp .\MODIFICATION_JAR;%CLASSPATH% ...
```

Das gleiche gilt für das weitere Auftreten von Classpath-Definitionen (`-cp …`).

Der `./MODIFICATION_JAR` Teil garantiert das die Klassen dieses JARs zuerst verwendet werden, d.h. man kann mittels der Reihenfolge der JAR-Definitionen die Präzedenz beeinflussen.

Dadurch ist sicher gestellt, dass die modifizierte Version des *SimpleAxisServer* in `MODIFICATION_JAR` gestartet wird, anstatt der Version in *axis.jar*!

Ohne der Option `-t` wird nur die single-threaded Version des *SimpleAxisServer*s gestartet!

Um zusätzlich Logging Information von den neuen Komponenten (z.B. von Queues und Thread Pools) auszugeben, muss der *SimpleAxisServer* mit der Option `-d` gestartet werden.

Die Modifikation, die das *Pooling* Pattern für Threads benutzt (in *axis_thread_pool.jar)*, besitzt folgende zusätzliche Optionen:

- `-minThreads` für die Anzahl an Threads, die der Pool beim Start instanzieren soll (wenn größer Null findet *eager acquisition* statt, wenn gleich Null findet *lazy acquisition* statt; `-minThreads` muss kleiner gleich `-maxThreads` sein).

- `-maxThreads` die Maximalanzahl an Threads, die sich in diesem Pool befinden können (muss größer Null sein)

Der Server, der nach dem *Leader/Followers* Pattern (in *axis_leader_followers.jar*) implementiert wurde, unterstützt die zusätzliche Option –`threadCount`, um die Anzahl an Threads zu bestimmen (muss größer Null sein).

Die Modifikation, die das *Half-Sync/Half-Async* Pattern benutzt (in *axis_halfsync_halfasync.jar*), unterstützt die folgenden zusätzlichen Optionen:

- `-threadCount` um die Anzahl an Threads die Requests aus der Queue bearbeiten zu setzen (muss größer Null sein).

- `-maxQueueSize` um die maximale Größe der Queue zu setzen (muss auch größer Null sein).

Anhang C: Das Buildsystem

Um den Build-Prozess zu automatisieren, wird das Open Source Tool Ant[6] der Apache Software Foundation verwendet.

Mittels `ant -projecthelp` in der Kommandozeile im Verzeichnis *work* wird Information über die Build Targets ausgegeben.

Da es zu Problemen mit der Ant-Version, die mit der Linux-Distribution Fedora Core 2 installiert wurde gekommen ist, wird empfohlen eine eigene neue zu installieren, falls diese Distribution genutzt wird.

Build Targets:

- **ThreadPoolModification**
  Erstellt eine Version des *SimpleAxisServer*, welche den Thread Pool (nach dem *Pooling* Pattern aus [POSA3]) im Paket `da_pattern_axis.threads` nutzt. Die Modifikation wird in *axis_thread_pool.jar* gepackt.

- **HalfSync_HalfAsyncModification**
  Mittels dieses Targets wird eine Version des *SimpleAxisServer* erstellt, welche eine Queue und eine Anzahl von Threads, die die Requests, die in der Queue abgelegt wurden bearbeiten, verwendet. Der Server folgt dem *Half-Sync/Half-Async* Pattern [POSA2]. Die Modifikation wird in *axis_halfsync_halfasync.jar* gepackt.

- **LeaderFollowersModification**
  Erstellt eine Version des *SimpleAxisServer*, der einen Thread Pool nach dem *Leader/Followers* Pattern [POSA2] nutzt. Die Modifikation wird in *axis_leader_followers.jar* gepackt.

[6] Siehe http://ant.apache.org, Abruf am 2004-11-16.

- **InstallAxisJar**

  Kopiert das Original JAR (*axis.jar*) in das Verzeichnis für die JARs der Modifikationen, um sie mit der Konfiguration und den Web Services der Modifikationen zu testen.

- **Threads**

  Erstellt die Klassen für Multi-threading (z.B. Thread Pools, Barrier etc.), diese befinden sich im Paket `da_pattern_axis.threads`.

- **Container**

  Erstellt Klassen, die als Container dienen (z.B. eine FIFO-Queue etc.), diese befinden sich im Paket `da_pattern_axis.container`.

- **TestSuite**

  Dieses Target erstellt die *Test Suite*-Applikation, die Testfälle und auch den Web Service für die Tests. Die Applikation und die Testfälle werden in *test_suite.jar* gepackt.

- **DeployTestSuite**

  Installiert den Web Service für die Tests.

- **UndeployTestSuite**

  Deinstalliert den Web Service für die Tests.

- **SimpleTestCases**

  Erstellt Web Services und Testclients für einfache Tests. Diese Klassen werden in *simple_test_cases.jar* gepackt.

- **DeploySimpleTestCases**

  Installiert die Web Services für die einfachen Testclients.

- **UndeploySimpleTestCases**

  Deinstalliert die Web Services für die einfachen Testclients.

- **all**

  Erstellt alles. Es findet aber kein Deployment bzw. keine Installation von Web Services statt!

- **clean**

  Löscht alle nicht benötigten Dateien (Backup-Dateien, das Verzeichnis *classes* für die Class-Dateien, das Verzeichnis *lib*, das Verzeichnis *docs* etc.). Es werden aber nicht Web Services undeployt bzw. deinstalliert.

- **javadocs**

  Erstellt die Javadoc-Dokumentation für das Paket `da_pattern_axis`.

- **Utils**

  Erstellt die Utility-Klassen für dieses Projekt.

Anhang D: Messergebnisse

Die Ergebnisse in den Spalten Min, Max, Med, Avg und StdDev sind Zeitangaben in Millisekunden. Wobei Min der minimale gemessene Wert ist, Max der Maximale, Med der Median (d.h. 50% der Werte sind kleiner als dieser), Avg ist das arithmetische Mittel und StdDev die Standardabweichung vom arithmetischen Mittel. Min, Max, Med und Avg geben die Antwortzeiten an.

Messergebnisse für die Tests mit 1, 25, 50, 75 und 100 Client Threads:

ReturnVoidTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,13	7	83	10	10,5	1,63
Thread-Per-Request	15,82	9	194	11	11,89	2,22
Lazy Thread Pool	16,03	7	120	10	10,85	1,53
Half-Sync/Half-Async	15,92	8	100	10	11,23	1,77
Leader/Followers	16,13	7	53	10	10,56	1,7

ReturnVoidTest 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	7	433	150	150,08	37,15
Thread-Per-Request	100	7	854	182	195,23	58,47
Lazy Thread Pool	108,7	7	856	124	153,64	80,91
Half-Sync/Half-Async	113,64	7	1411	116	147,51	82,77
Leader/Followers	113,64	7	741	130	148,95	67,39

ReturnVoidTest 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	131,58	7	2312	309	301,95	95,91
Thread-Per-Request	108,7	7	820	394	399,88	59,97
Lazy Thread Pool	125	6	2261	262	317,29	183,96
Half-Sync/Half-Async	125	7	2819	234	297,83	193,65
Leader/Followers	125	6	1473	280	306,9	138,78

ReturnVoidTest 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	137,5	6	9782	305	430,25	265,56
Thread-Per-Request	107,61	8	9502	436	552,36	280,94
Lazy Thread Pool	130,26	6	3587	197	316,57	246,92
Half-Sync/Half-Async	130,26	6	6098	318	452,35	323,92
Leader/Followers	112,5	6	9661	347	465,9	323,54

ReturnVoidTest 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	138,89	6	10275	287	523,79	461,62
Thread-Per-Request	104,17	7	21556	418	649,48	525,36
Lazy Thread Pool	131,58	6	9148	286	563,99	528,9
Half-Sync/Half-Async	131,58	6	6937	239	584,38	572,55
Leader/Followers	131,58	6	11277	290	544,81	473,46

EchoIntegerTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,56	7	131	9	9,04	1,01
Thread-Per-Request	16,03	9	125	11	11,22	0,94
Lazy Thread Pool	16,45	7	118	9	9,35	0,86
Half-Sync/Half-Async	16,34	8	127	9	9,64	1,01
Leader/Followers	16,56	7	28	9	9,02	0,93

EchoIntegerTest 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	7	3217	140	139,92	40,91
Thread-Per-Request	92,59	9	617	210	222,68	53,39
Lazy Thread Pool	125	7	1943	105	130,55	67,64
Half-Sync/Half-Async	125	7	1976	110	127,41	56,5
Leader/Followers	119,05	8	1506	110	132,45	62,85

EchoIntegerTest 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	125	7	2463	308	312,09	90,72
Thread-Per-Request	89,29	9	1076	491	501,87	72,74
Lazy Thread Pool	125	7	3061	246	312,49	204,97
Half-Sync/Half-Async	119,05	7	2523	255	324,4	203,8
Leader/Followers	125	7	1702	286	315	156,05

EchoIntegerTest 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	123,75	7	9258	335	468,22	315,15
Thread-Per-Request	91,67	10	9853	556	682,82	296,51
Lazy Thread Pool	123,75	7	4967	382	471,56	310,14
Half-Sync/Half-Async	123,75	6	7726	141	454,7	488,09
Leader/Followers	117,86	7	4466	362	471,99	303,26

EchoIntegerTest 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	92,59	7	21409	316	560,9	503,68
Thread-Per-Request	92,59	10	22027	556	867,5	642,46
Lazy Thread Pool	100	8	22178	370	634,02	607,48
Half-Sync/Half-Async	125	7	9948	170	603,97	665,95
Leader/Followers	119,05	7	10211	321	590,77	531,19

EchoIntegerArrayTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	15,92	9	123	11	11,28	1,23
Thread-Per-Request	15,43	11	228	13	13,36	1,5
Lazy Thread Pool	16,03	9	37	11	11,27	0,99
Half-Sync/Half-Async	15,82	10	146	11	11,84	1,44
Leader/Followers	16,03	9	49	11	11,14	1,2

EchoIntegerArrayTest 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	948	210	199,69	35,76
Thread-Per-Request	73,53	11	752	272	285,18	62,81
Lazy Thread Pool	96,15	9	1373	151	178,28	91,67
Half-Sync/Half-Async	96,15	9	884	158	177,7	83,13
Leader/Followers	92,59	9	803	191	204,18	70,57

EchoIntegerArrayTest 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	3219	449	441,05	127,64
Thread-Per-Request	73,53	10	1381	608	629,32	81,59
Lazy Thread Pool	96,15	9	2584	357	403,2	237,21
Half-Sync/Half-Async	96,15	9	2050	368	415,64	231,69
Leader/Followers	92,59	9	1504	455	451,19	142,39

EchoIntegerArrayTest 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	95,19	9	9850	454	581,84	350,71
Thread-Per-Request	70,71	11	21384	687	881,67	421,22
Lazy Thread Pool	95,19	8	6183	302	600,03	546,67
Half-Sync/Half-Async	95,19	8	9085	68	555,28	753,69
Leader/Followers	91,67	9	9251	540	668,43	371,66

EchoIntegerArrayTest 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	10988	468	816,12	667,81
Thread-Per-Request	53,19	11	45026	678	1083,94	851,53
Lazy Thread Pool	96,15	8	7532	278	691,67	724,97
Half-Sync/Half-Async	92,59	8	7597	146	808,4	958,95
Leader/Followers	86,21	9	21435	535	863,12	699,31

EchoStructTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,23	8	29	10	10,08	0,8
Thread-Per-Request	15,06	13	251	14	14,79	1,31
Lazy Thread Pool	16,13	8	163	10	10,54	0,98
Half-Sync/Half-Async	16,03	9	30	10	10,7	0,98
Leader/Followers	16,23	8	22	10	10,21	0,97

EchoStructTest 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,17	8	519	190	181,97	27,34
Thread-Per-Request	65,79	13	870	310	328,7	65,52
Lazy Thread Pool	108,7	8	2692	116	149,16	84,22
Half-Sync/Half-Async	108,7	8	694	133	153,96	72,31
Leader/Followers	104,17	8	846	153	168,13	65,51

EchoStructTest 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,17	8	1326	420	403,09	72,65
Thread-Per-Request	67,57	15	1212	673	684,28	73,05
Lazy Thread Pool	108,7	8	2651	254	344,97	228,64
Half-Sync/Half-Async	108,7	8	1730	318	348,93	186,02
Leader/Followers	108,7	8	1274	357	366,73	150,84

EchoStructTest 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	103,12	8	9523	453	574,21	340,27
Thread-Per-Request	57,56	13	21867	772	977,5	450,06
Lazy Thread Pool	107,61	7	6577	251	525,97	486,36
Half-Sync/Half-Async	107,61	8	9273	77	438,31	551,64
Leader/Followers	103,12	8	9580	419	570,09	355,63

EchoStructTest 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	100	8	10580	399	713,06	650,68
Thread-Per-Request	64,1	12	22514	785	1234,56	897,8
Lazy Thread Pool	108,7	7	9779	214	667,97	698,78
Half-Sync/Half-Async	104,17	8	9476	190	739,42	835,27
Leader/Followers	104,17	8	11058	421	694,17	573,49

EchoStructArrayTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	11,96	26	226	30	32,01	4,74
Thread-Per-Request	11,42	31	224	34	36,2	4,98
Lazy Thread Pool	12,32	25	207	29	29,83	2,19
Half-Sync/Half-Async	12,25	27	240	30	30,12	2,15
Leader/Followers	12,2	27	229	29	30,4	2,34

EchoStructArrayTest 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	28,41	26	1860	747	752,25	215,16
Thread-Per-Request	24,27	34	1986	951	962,61	162,65
Lazy Thread Pool	27,47	26	2518	623	695,09	334,86
Half-Sync/Half-Async	28,09	26	2393	718	749,29	247,07
Leader/Followers	26,88	27	2522	807	827,36	211,5

EchoStructArrayTest 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	29,76	25	9178	809	1432,11	1101,81
Thread-Per-Request	24,51	33	3842	1955	1947,5	185,61
Lazy Thread Pool	29,07	25	6904	893	1183,53	862,89
Half-Sync/Half-Async	27,78	25	5482	1318	1442,59	743,84
Leader/Followers	27,47	26	6149	1683	1663,42	339,02

EchoStructArrayTest 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,33	27	16499	574	1521,31	1521,63
Thread-Per-Request	24,5	31	47005	2191	2762,41	1172,21
Lazy Thread Pool	29,12	25	13211	1492	1891,87	1333,52
Half-Sync/Half-Async	27,81	25	10622	2054	2305	1097,26
Leader/Followers	28,12	28	12992	1980	2338,01	994,46

EchoStructArrayTest 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	30,86	29	22662	600	2754,35	2878,17
Thread-Per-Request	20,83	31	93016	2246	3427,01	2260,93
Lazy Thread Pool	28,74	25	13581	1978	2482,68	1664,07
Half-Sync/Half-Async	28,74	25	12147	2122	2574,86	1550
Leader/Followers	27,78	26	25657	2048	3076,97	2022,41

EchoStringTest 2kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,67	7	21	8	8,62	0,82
Thread-Per-Request	16,03	9	19	11	10,87	0,63
Lazy Thread Pool	16,56	7	20	9	9,1	0,54
Half-Sync/Half-Async	16,45	7	27	9	9,32	0,66
Leader/Followers	16,67	7	26	8	8,7	0,79

EchoStringTest 2kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	7	1949	157	146,34	34,77
Thread-Per-Request	96,15	10	466	200	203,75	38,82
Lazy Thread Pool	125	7	928	98	123,43	67,63
Half-Sync/Half-Async	125	7	1819	99	120,19	63,52
Leader/Followers	125	7	853	103	126,17	65,05

EchoStringTest 2kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	7	2634	351	337,16	116,47
Thread-Per-Request	92,59	12	925	474	477,2	49,34
Lazy Thread Pool	125	7	2555	181	299,89	230,14
Half-Sync/Half-Async	125	7	1894	198	295,35	214,92
Leader/Followers	119,05	7	1511	283	320,84	169,85

EchoStringTest 2kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	112,5	7	9298	377	506,49	321,2
Thread-Per-Request	91,67	10	9874	538	660,12	313,77
Lazy Thread Pool	123,75	6	5853	159	465,5	471,2
Half-Sync/Half-Async	117,86	6	5825	63	462,98	576,86
Leader/Followers	123,75	7	11027	288	448,79	357,2

EchoStringTest 2kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	8	9542	396	644,15	508,25
Thread-Per-Request	86,21	10	22256	554	846,88	631
Lazy Thread Pool	119,05	7	9211	110	630,07	751,98
Half-Sync/Half-Async	125	7	6064	233	605,74	614,23
Leader/Followers	119,05	7	5085	241	426,09	373,64

EchoStringTest 4kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,45	8	15	9	9,2	0,82
Thread-Per-Request	15,92	9	145	11	11,52	0,97
Lazy Thread Pool	16,34	7	19	9	9,56	0,8
Half-Sync/Half-Async	16,34	7	18	10	9,77	0,77
Leader/Followers	16,45	7	17	9	9,2	0,8

EchoStringTest 4kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	108,7	7	1903	173	163,59	33,52
Thread-Per-Request	89,29	10	618	221	230,52	46,67
Lazy Thread Pool	119,05	7	1882	109	135,1	72,4
Half-Sync/Half-Async	113,64	8	1661	114	137,2	73,6
Leader/Followers	113,64	7	1176	130	148,61	64,69

EchoStringTest 4kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	113,64	7	2204	384	371,55	112,86
Thread-Per-Request	86,21	11	1225	510	525,47	64,01
Lazy Thread Pool	113,64	8	1874	242	336,69	238,62
Half-Sync/Half-Async	113,64	8	2317	233	328,13	228,38
Leader/Followers	113,64	8	1778	326	347,28	163,13

EchoStringTest 4kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	107,61	7	9864	427	547,75	315,09
Thread-Per-Request	85,34	10	9743	606	759,08	339,38
Lazy Thread Pool	107,61	7	10727	205	541,17	551
Half-Sync/Half-Async	107,61	7	6715	254	541,9	503,5
Leader/Followers	112,5	7	5420	362	526,92	377,63

EchoStringTest 4kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	108,7	7	11947	425	703,52	589,13
Thread-Per-Request	86,21	11	22441	582	913,46	689,27
Lazy Thread Pool	113,64	7	10822	82	662,01	844,19
Half-Sync/Half-Async	108,7	7	9221	56	660,83	908,98
Leader/Followers	113,64	8	9650	438	696,12	535,94

EchoStringTest 8kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,13	9	16	10	10,57	0,96
Thread-Per-Request	15,62	10	18	12	12,58	0,79
Lazy Thread Pool	16,13	9	17	10	10,75	0,84
Half-Sync/Half-Async	16,03	9	18	11	11,02	0,75
Leader/Followers	16,13	9	26	10	10,73	1,01

EchoStringTest 8kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	1605	200	194,32	44,47
Thread-Per-Request	78,12	12	834	259	268,51	53,64
Lazy Thread Pool	100	9	1589	142	169,26	83,95
Half-Sync/Half-Async	100	8	1379	142	167,56	81,66
Leader/Followers	92,59	10	846	176	196,3	78,35

EchoStringTest 8kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	1964	426	423,44	156,99
Thread-Per-Request	75,76	12	1400	593	614,74	76,66
Lazy Thread Pool	96,15	9	3544	282	403	277,32
Half-Sync/Half-Async	96,15	8	2437	306	383,77	238,97
Leader/Followers	92,59	9	1608	390	422,43	203,61

EchoStringTest 8kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	95,19	9	10353	461	622,73	383,45
Thread-Per-Request	72,79	12	10145	697	874,09	383,98
Lazy Thread Pool	91,67	8	6935	302	636,42	605,58
Half-Sync/Half-Async	95,19	8	6689	213	498,35	489,56
Leader/Followers	91,67	9	4789	535	631,26	353

EchoStringTest 8kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	92,59	9	10422	487	792,22	644,18
Thread-Per-Request	73,53	12	22648	688	1113,2	820,24
Lazy Thread Pool	92,59	8	8311	114	639,47	766,18
Half-Sync/Half-Async	92,59	8	9417	80	718,07	968,45
Leader/Followers	89,29	9	10999	492	831	672,62

EchoStringTest 16kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	15,43	10	249	13	13,26	1,51
Thread-Per-Request	15,06	13	22	14	14,83	1,01
Lazy Thread Pool	15,53	11	25	12	13,01	1,06
Half-Sync/Half-Async	15,53	11	21	13	13,21	0,96
Leader/Followers	15,43	11	42	13	13,22	1,33

EchoStringTest 16kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	73,53	11	764	265	258,49	67,29
Thread-Per-Request	58,14	14	1031	338	374,77	87,61
Lazy Thread Pool	73,53	11	1620	196	239,51	129,32
Half-Sync/Half-Async	75,76	11	1288	194	223,54	111,28
Leader/Followers	69,44	12	1035	260	275,63	93,77

EchoStringTest 16kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	75,76	11	1915	471	504,87	207,23
Thread-Per-Request	58,14	13	1428	763	804,46	121,53
Lazy Thread Pool	71,43	10	5493	128	541,05	604,06
Half-Sync/Half-Async	71,43	10	3252	447	546,73	343,41
Leader/Followers	67,57	11	1901	589	598,2	214,36

EchoStringTest 16kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	75	10	8064	191	817,77	882,79
Thread-Per-Request	57,56	14	10489	868	1121,36	517,32
Lazy Thread Pool	70,71	10	7957	188	695,57	760,07
Half-Sync/Half-Async	72,79	10	8875	268	823,43	852,88
Leader/Followers	68,75	12	6161	656	895,35	537,82

EchoStringTest 16kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	73,53	10	11677	149	1086,62	1303,65
Thread-Per-Request	52,08	13	45034	866	1388,52	1069,52
Lazy Thread Pool	71,43	10	12054	262	1137,66	1188,1
Half-Sync/Half-Async	71,43	10	11945	431	1099,46	1128,48
Leader/Followers	67,57	11	21292	713	1143,93	835,83

EchoStringTest 32kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	14,53	15	24	17	17,42	1,3
Thread-Per-Request	14,12	16	44	19	19,47	1,46
Lazy Thread Pool	14,53	15	44	17	17,64	1,59
Half-Sync/Half-Async	14,53	15	217	17	17,89	1,6
Leader/Followers	14,53	15	24	17	17,68	1,5

EchoStringTest 32kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	52,08	16	1021	359	364,88	111,97
Thread-Per-Request	39,68	20	1232	513	566,95	139,65
Lazy Thread Pool	48,08	15	2198	317	378,79	206,76
Half-Sync/Half-Async	46,3	14	1857	378	415,51	173,81
Leader/Followers	47,17	15	1293	398	421,52	150,94

EchoStringTest 32kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	52,08	14	7675	580	781,78	603,53
Thread-Per-Request	40,32	20	2059	1208	1167,4	159,09
Lazy Thread Pool	48,08	14	8331	422	854,87	792,18
Half-Sync/Half-Async	46,3	16	3228	720	810,05	411,32
Leader/Followers	47,17	15	3357	857	873,44	336,59

EchoStringTest 32kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	51,56	15	6872	154	407,93	409,26
Thread-Per-Request	39,92	18	22588	1372	1668,04	721,2
Lazy Thread Pool	48,53	14	12910	286	1082,57	1116,3
Half-Sync/Half-Async	45,83	15	23283	1012	1224,39	759,94
Leader/Followers	45,83	15	12274	1104	1337,96	682,39

EchoStringTest 32kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	51,02	14	15507	308	1648,89	1732,52
Thread-Per-Request	26,32	17	93179	1373	2053,16	1443,22
Lazy Thread Pool	46,3	14	18530	777	1529,28	1455,39
Half-Sync/Half-Async	45,45	15	10774	1174	1561,22	1098,27
Leader/Followers	46,3	15	22543	1202	1689,82	1065,24

EchoStringTest 64kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	12,63	24	364	27	28,54	2,93
Thread-Per-Request	12,38	26	312	29	30,15	2,72
Lazy Thread Pool	12,69	23	290	26	27,97	2,65
Half-Sync/Half-Async	12,63	24	181	29	28,31	2,7
Leader/Followers	12,76	24	37	26	27,9	2,32

EchoStringTest 64kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,65	24	1802	563	589,38	184,03
Thread-Per-Request	25,51	26	2094	912	904,85	194,5
Lazy Thread Pool	27,78	24	2440	610	689,71	346,02
Half-Sync/Half-Async	26,6	24	2674	717	762,21	291,48
Leader/Followers	27,47	25	2299	716	751,68	265,99

EchoStringTest 64kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	30,86	23	9424	587	1386,83	1213,82
Thread-Per-Request	22,94	27	4143	2075	2074,33	295,89
Lazy Thread Pool	28,41	23	7557	1050	1371,23	1042,82
Half-Sync/Half-Async	26,04	23	5224	1329	1406,11	678,75
Leader/Followers	27,17	24	4690	1505	1520,26	588,53

EchoStringTest 64kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	29,82	25	16416	656	2204,03	2055,63
Thread-Per-Request	23,13	26	45045	2200	2696,6	1107
Lazy Thread Pool	27,81	23	12485	1238	1811,01	1521,7
Half-Sync/Half-Async	26,9	24	10082	2002	2195,63	1115,72
Leader/Followers	26,9	23	14671	1889	2154,09	1049,14

EchoStringTest 64kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	30,49	25	28832	508	2837,66	2859,66
Thread-Per-Request	24,75	26	93541	2293	3370,25	2218,01
Lazy Thread Pool	27,47	23	13708	2960	3023,92	1932,11
Half-Sync/Half-Async	27,47	24	12334	1851	2369,02	1528,92
Leader/Followers	26,32	25	15626	2616	3153,81	1815,53

EchoStringTest 128kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	7,91	47	356	86	75,47	17,73
Thread-Per-Request	9,43	47	385	52	54,33	4,5
Lazy Thread Pool	7,96	45	311	85	74,16	18,57
Half-Sync/Half-Async	9,73	45	370	51	51,46	3,83
Leader/Followers	8,5	45	243	56	66,25	17,98

EchoStringTest 128kb 25 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,67	48	2623	1359	1366,86	216,67
Thread-Per-Request	12,89	59	7252	1872	1829,25	572,38
Lazy Thread Pool	13,81	86	6671	1427	1525,22	686,28
Half-Sync/Half-Async	14,2	56	7547	1453	1531,33	575,5
Leader/Followers	13,81	55	4524	1569	1582,48	513,84

EchoStringTest 128kb 50 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,45	70	12480	2966	2772,07	1934,87
Thread-Per-Request	12,63	48	15802	4210	3723,93	1544,9
Lazy Thread Pool	14,04	51	12305	2363	2624,27	1447,26
Half-Sync/Half-Async	13,97	47	10587	3024	3070,35	1171,56
Leader/Followers	13,89	58	11444	3228	3252,26	1200,19

EchoStringTest 128kb 75 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,61	93	27684	3112	4053,05	2815,91
Thread-Per-Request	12,38	49	27767	6381	5684,63	2508,08
Lazy Thread Pool	14,31	69	16475	4092	4394,16	2209,03
Half-Sync/Half-Async	14,39	86	19482	4392	4612,97	2065,38
Leader/Followers	14,31	86	26878	4107	4771,63	1867,29

EchoStringTest 128kb 100 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	14,29	50	94825	3375	4934,96	3440,88
Thread-Per-Request	13,37	48	93044	5514	6373,87	2730,35
Lazy Thread Pool	13,74	71	32473	4152	5116,79	3365,63
Half-Sync/Half-Async	14,2	48	49660	5557	6150,65	3077,59
Leader/Followers	13,66	46	51394	4898	5922,95	2870,44

Messergebnisse für die Tests mit 1, 2, 4, 8, 16 und 20 Client Threads:

ReturnVoidTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,13	7	46	10	10,32	1,5
Thread-Per-Request	15,72	9	583	11	12,13	2,69
Lazy Thread Pool	16,03	8	83	10	10,78	1,54
Half-Sync/Half-Async	15,82	8	104	10	11,4	1,97
Leader/Followers	16,13	7	44	10	10,57	1,77

ReturnVoidTest 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	33,33	7	101	8	8,88	1,16
Thread-Per-Request	30,49	8	199	14	14,17	2,52
Lazy Thread Pool	32,47	7	99	9	9,84	1,56
Half-Sync/Half-Async	30,49	7	132	14	13,79	2,85
Leader/Followers	32,89	7	103	9	9,35	1,51

ReturnVoidTest 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	65,79	6	39	8	9,48	2,1
Thread-Per-Request	54,35	7	142	22	22,04	3,14
Lazy Thread Pool	65,79	6	36	9	10,02	2,19
Half-Sync/Half-Async	52,08	7	135	23	23,62	4,31
Leader/Followers	62,5	7	213	11	12,94	4,85

ReturnVoidTest 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	108,87	6	75	17	19,28	7,52
Thread-Per-Request	86,34	7	201	40	39,92	5,69
Lazy Thread Pool	108,87	6	163	18	20,89	8,94
Half-Sync/Half-Async	100,16	6	173	21	23,63	9,88
Leader/Followers	92,74	6	194	34	33,63	11,37

ReturnVoidTest 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	125,6	6	2169	61	61,73	22,23
Thread-Per-Request	119,62	7	389	71	82,6	25,75
Lazy Thread Pool	125,6	6	575	48	63,01	36,6
Half-Sync/Half-Async	125,6	6	1157	48	59,68	30,2
Leader/Followers	125,6	6	580	52	63,52	32,66

ReturnVoidTest 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	138,89	6	549	77	82,82	26,95
Thread-Per-Request	119,05	7	364	94	115,68	43,55
Lazy Thread Pool	131,58	6	859	60	82,22	50,94
Half-Sync/Half-Async	131,58	6	715	70	85,89	44,18
Leader/Followers	131,58	6	1312	71	83,27	41,05

EchoIntegerTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,56	7	25	9	8,9	0,92
Thread-Per-Request	16,03	10	30	11	11,15	0,79
Lazy Thread Pool	16,56	7	26	9	9,18	0,7
Half-Sync/Half-Async	16,34	8	131	9	9,89	1,34
Leader/Followers	16,56	7	123	9	8,89	1,03

EchoIntegerTest 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	33,33	7	22	8	8,78	1,04
Thread-Per-Request	30,12	9	113	16	15,11	2,38
Lazy Thread Pool	32,89	7	132	9	9,66	1,6
Half-Sync/Half-Async	30,12	8	36	14	14,21	2,2
Leader/Followers	32,89	7	22	9	9,65	1,85

EchoIntegerTest 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	65,79	7	32	9	9,59	1,95
Thread-Per-Request	53,19	9	161	25	23,5	5,02
Lazy Thread Pool	65,79	7	32	9	9,87	1,93
Half-Sync/Half-Async	53,19	7	189	23	23,02	4,67
Leader/Followers	62,5	7	48	12	12,61	3,84

EchoIntegerTest 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,33	7	137	19	21,2	8,51
Thread-Per-Request	86,34	10	178	42	41,09	13,57
Lazy Thread Pool	108,87	7	105	17	20,49	8,41
Half-Sync/Half-Async	104,33	6	173	19	21,5	8,28
Leader/Followers	96,31	7	192	32	31,49	11,22

EchoIntegerTest 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,62	7	2283	68	68,9	23,83
Thread-Per-Request	93,04	10	507	106	117,9	41,17
Lazy Thread Pool	125,6	7	606	50	63,71	34,4
Half-Sync/Half-Async	119,62	7	2253	51	62,96	32,86
Leader/Followers	125,6	7	782	50	61,95	31,95

EchoIntegerTest 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	119,05	7	2701	99	99,88	34,22
Thread-Per-Request	96,15	9	475	155	156,62	42,05
Lazy Thread Pool	125	7	900	67	86,02	47,21
Half-Sync/Half-Async	131,58	7	970	72	84,12	40,91
Leader/Followers	131,58	7	972	76	88,08	42,13

EchoIntegerArrayTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,03	9	55	11	11,1	1,07
Thread-Per-Request	15,43	12	337	13	13,4	1,45
Lazy Thread Pool	16,03	9	54	11	11,27	1,04
Half-Sync/Half-Async	15,82	10	165	11	11,81	1,4
Leader/Followers	16,03	9	41	11	11,07	1,22

EchoIntegerArrayTest 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,47	9	19	11	10,74	0,95
Thread-Per-Request	28,41	11	157	18	18,73	2,26
Lazy Thread Pool	31,65	8	27	11	11,9	1,87
Half-Sync/Half-Async	28,74	9	189	18	17,72	2,75
Leader/Followers	32,05	9	167	10	10,95	1,28

EchoIntegerArrayTest 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	58,14	9	48	15	16,16	5,26
Thread-Per-Request	54,35	10	203	21	22,43	6,78
Lazy Thread Pool	60,98	8	49	11	13,8	3,82
Half-Sync/Half-Async	53,19	9	55	22	22,54	6,51
Leader/Followers	58,14	8	50	16	17,2	5,93

EchoIntegerArrayTest 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	92,74	8	321	25	28,82	11,45
Thread-Per-Request	75,88	11	364	48	53,19	22,93
Lazy Thread Pool	92,74	8	407	23	29,96	15,82
Half-Sync/Half-Async	89,43	9	233	27	31,75	13,99
Leader/Followers	86,34	9	239	40	40,36	13,7

EchoIntegerArrayTest 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,62	8	1385	119	111,42	26,22
Thread-Per-Request	76,12	12	610	149	156,88	46,92
Lazy Thread Pool	100,48	9	1169	74	92,73	47,56
Half-Sync/Half-Async	96,62	9	2141	76	94,1	48,19
Leader/Followers	96,62	9	481	98	108,59	40,96

EchoIntegerArrayTest 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	92,59	9	5087	159	150	28,82
Thread-Per-Request	75,76	13	687	195	208,89	51,21
Lazy Thread Pool	96,15	8	3236	102	124,67	64,63
Half-Sync/Half-Async	100	9	703	113	126,3	50,77
Leader/Followers	96,15	9	782	131	144,35	54,25

EchoStructTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,23	8	21	10	10	0,73
Thread-Per-Request	15,15	13	59	14	14,65	0,99
Lazy Thread Pool	16,23	8	29	10	10,26	0,69
Half-Sync/Half-Async	16,03	9	176	10	10,76	1,03
Leader/Followers	16,23	8	22	10	10,21	0,99

EchoStructTest 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,89	8	21	10	9,79	0,93
Thread-Per-Request	28,09	12	55	20	20,14	2,24
Lazy Thread Pool	32,47	8	27	10	10,35	1,03
Half-Sync/Half-Async	29,41	8	40	16	16,44	2,22
Leader/Followers	32,47	8	21	10	10,06	1,16

EchoStructTest 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	65,79	8	42	10	10,25	1,53
Thread-Per-Request	54,35	12	59	20	21,69	5,54
Lazy Thread Pool	64,1	8	40	10	10,92	1,83
Half-Sync/Half-Async	51,02	8	73	26	25,65	5,29
Leader/Followers	62,5	8	42	10	12,16	3,45

EchoStructTest 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	100,16	8	141	24	25,78	10,32
Thread-Per-Request	67,68	12	357	57	64,97	27,92
Lazy Thread Pool	100,16	8	216	21	25,48	11,49
Half-Sync/Half-Async	100,16	8	299	22	25,93	11,42
Leader/Followers	89,43	8	330	35	35,84	13,15

EchoStructTest 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,67	8	1147	96	96,14	19,02
Thread-Per-Request	67,89	13	634	170	180	45,74
Lazy Thread Pool	109,22	8	1501	59	78,68	45,41
Half-Sync/Half-Async	104,67	8	2054	62	79,29	42,76
Leader/Followers	109,22	8	492	75	85,67	36,72

EchoStructTest 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,17	8	6567	115	119,95	55,61
Thread-Per-Request	67,57	13	764	228	240,97	55,92
Lazy Thread Pool	113,64	8	1312	89	106,49	52,19
Half-Sync/Half-Async	113,64	8	2299	90	105,85	48,29
Leader/Followers	108,7	8	1538	99	116,17	52,31

EchoStructArrayTest 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	12,32	27	244	29	29,81	1,89
Thread-Per-Request	11,74	31	239	33	33,89	2,12
Lazy Thread Pool	12,38	26	240	29	29,56	1,92
Half-Sync/Half-Async	12,14	26	233	30	30,86	2,34
Leader/Followers	12,25	27	236	29	29,94	2,06

EchoStructArrayTest 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	23,81	26	222	30	32,06	4,9
Thread-Per-Request	21,19	30	256	41	42,72	6,04
Lazy Thread Pool	24,04	25	232	29	31,4	4,5
Half-Sync/Half-Async	20,16	26	297	43	47,25	12,64
Leader/Followers	22,32	25	282	36	37,45	7,42

EchoStructArrayTest 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,47	25	307	66	68,31	17,17
Thread-Per-Request	29,07	31	541	81	85,64	13,44
Lazy Thread Pool	30,86	25	488	57	73,08	36,5
Half-Sync/Half-Async	29,41	27	638	70	82,63	31,51
Leader/Followers	30,12	26	375	70	79,08	26,88

EchoStructArrayTest 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,95	26	1737	190	185,92	25,11
Thread-Per-Request	26,92	30	1059	225	243,38	80,3
Lazy Thread Pool	30,17	25	746	170	197,19	93,06
Half-Sync/Half-Async	28,78	27	770	188	215,22	80,63
Leader/Followers	29,12	25	809	186	212,25	79,26

EchoStructArrayTest 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,4	26	4559	435	420,92	111,33
Thread-Per-Request	25,12	49	1773	528	573,94	143,64
Lazy Thread Pool	28,87	25	1771	389	436,98	197
Half-Sync/Half-Async	28,22	27	1482	442	478,37	154,81
Leader/Followers	28,22	26	1479	449	485,79	148,2

EchoStructArrayTest 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,25	27	4211	549	514,71	140,45
Thread-Per-Request	24,75	32	1830	696	745,3	167,37
Lazy Thread Pool	28,74	26	2291	489	545,46	247,1
Half-Sync/Half-Async	28,41	27	2040	563	594,87	191,75
Leader/Followers	28,09	26	2062	583	621,84	174,41

EchoStringTest 2kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,67	7	23	8	8,67	0,82
Thread-Per-Request	16,03	9	22	11	10,95	0,62
Lazy Thread Pool	16,56	7	20	9	9,03	0,5
Half-Sync/Half-Async	16,45	7	21	9	9,36	0,67
Leader/Followers	16,67	7	19	8	8,61	0,83

EchoStringTest 2kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	33,33	7	22	9	8,88	0,93
Thread-Per-Request	29,41	10	28	16	16,49	1,16
Lazy Thread Pool	32,47	7	22	9	9,71	1,47
Half-Sync/Half-Async	29,76	7	26	15	15,1	1,75
Leader/Followers	33,33	7	21	8	8,78	0,97

EchoStringTest 2kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	64,1	7	37	9	10,2	2,37
Thread-Per-Request	54,35	9	47	23	22,29	5,47
Lazy Thread Pool	64,1	7	39	9	10,9	2,75
Half-Sync/Half-Async	52,08	7	81	25	24,63	3,78
Leader/Followers	62,5	7	40	9	11,97	4,2

EchoStringTest 2kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,33	7	171	20	22,03	8,97
Thread-Per-Request	83,47	9	174	46	43,71	11,28
Lazy Thread Pool	104,33	7	111	20	22,58	9,36
Half-Sync/Half-Async	104,33	7	128	20	22,03	8,48
Leader/Followers	96,31	7	147	28	29,3	11,17

EchoStringTest 2kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	109,22	7	1072	86	89,39	22,99
Thread-Per-Request	93,04	10	472	108	118,64	42,27
Lazy Thread Pool	125,6	7	923	48	63,43	34,83
Half-Sync/Half-Async	119,62	7	1351	50	64,01	34,14
Leader/Followers	119,62	7	818	54	66,68	34,63

EchoStringTest 2kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	108,7	7	3997	119	119,81	32,87
Thread-Per-Request	92,59	10	458	158	161,59	42,13
Lazy Thread Pool	125	7	2259	67	92,15	55,41
Half-Sync/Half-Async	125	7	1504	79	91,22	42,97
Leader/Followers	125	7	609	82	91,83	41,13

EchoStringTest 4kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,45	8	17	9	9,23	0,82
Thread-Per-Request	15,82	9	80	11	11,82	1,23
Lazy Thread Pool	16,45	8	15	9	9,5	0,71
Half-Sync/Half-Async	16,34	8	20	10	9,87	0,76
Leader/Followers	16,45	8	18	9	9,27	0,89

EchoStringTest 4kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,89	7	22	9	9,42	1,09
Thread-Per-Request	29,07	10	29	17	17,2	1,44
Lazy Thread Pool	32,89	7	21	9	9,63	0,85
Half-Sync/Half-Async	29,07	8	29	16	16,63	1,78
Leader/Followers	32,89	7	26	9	9,36	1,04

EchoStringTest 4kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	65,79	7	37	9	10,06	1,75
Thread-Per-Request	53,19	10	49	24	23,44	5,68
Lazy Thread Pool	64,1	7	40	10	11,57	2,91
Half-Sync/Half-Async	51,02	7	45	25	25,27	4,24
Leader/Followers	60,98	7	42	12	13,45	4,41

EchoStringTest 4kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	100,16	7	232	22	24,51	9,92
Thread-Per-Request	83,47	10	198	46	44,36	15,09
Lazy Thread Pool	104,33	7	207	20	24,14	10,29
Half-Sync/Half-Async	100,16	7	146	21	24,21	9,48
Leader/Followers	92,74	7	402	33	34,22	13,03

EchoStringTest 4kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	104,67	8	3353	87	90,05	21,83
Thread-Per-Request	89,71	10	444	123	126,58	41,02
Lazy Thread Pool	114,18	7	781	58	72,98	38,24
Half-Sync/Half-Async	114,18	7	2145	61	72,74	35,07
Leader/Followers	114,18	7	749	68	77,27	35,46

EchoStringTest 4kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	108,7	8	3048	129	123,65	27,89
Thread-Per-Request	86,21	9	755	167	172,59	47,02
Lazy Thread Pool	119,05	7	1929	80	98,78	52,5
Half-Sync/Half-Async	113,64	8	1278	86	101,26	46,51
Leader/Followers	113,64	8	910	95	106,56	45,33

EchoStringTest 8kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,13	9	17	10	10,57	0,93
Thread-Per-Request	15,62	11	21	12	12,65	0,78
Lazy Thread Pool	16,13	9	16	10	10,71	0,82
Half-Sync/Half-Async	16,03	9	18	11	11,15	0,87
Leader/Followers	16,13	9	17	10	10,62	1,05

EchoStringTest 8kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,05	9	25	10	10,76	1,16
Thread-Per-Request	28,41	12	283	18	18,99	1,72
Lazy Thread Pool	32,05	9	28	10	11,02	1,08
Half-Sync/Half-Async	28,09	9	30	19	18,91	2,18
Leader/Followers	31,65	9	27	11	11,91	2,11

EchoStringTest 8kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	62,5	8	47	11	12,67	3,28
Thread-Per-Request	51,02	10	54	29	27,4	5,69
Lazy Thread Pool	62,5	8	303	11	13,17	3,79
Half-Sync/Half-Async	47,17	9	51	32	30,97	5,68
Leader/Followers	55,56	9	349	19	20,4	8,07

EchoStringTest 8kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	92,74	8	173	28	29,84	10,38
Thread-Per-Request	75,88	10	295	53	52,33	17,85
Lazy Thread Pool	92,74	8	381	25	31,02	15,3
Half-Sync/Half-Async	92,74	8	405	25	31,88	16,3
Leader/Followers	78,25	8	388	49	47,37	14,64

EchoStringTest 8kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,62	9	1178	111	110,95	24,07
Thread-Per-Request	78,5	11	533	146	150,53	45,05
Lazy Thread Pool	100,48	8	789	75	92,21	46,81
Half-Sync/Half-Async	96,62	8	1237	81	94,46	44,81
Leader/Followers	93,04	9	819	95	108,56	45,67

EchoStringTest 8kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	96,15	9	1604	153	150,48	33
Thread-Per-Request	78,12	12	695	189	202,28	47,17
Lazy Thread Pool	96,15	9	969	106	127,3	64,42
Half-Sync/Half-Async	96,15	9	640	111	125,05	55,94
Leader/Followers	96,15	8	881	128	141,29	54,38

EchoStringTest 16kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	15,53	11	20	12	12,89	1,12
Thread-Per-Request	15,06	13	24	14	14,87	0,95
Lazy Thread Pool	15,53	10	19	12	12,89	0,97
Half-Sync/Half-Async	15,43	11	99	13	13,38	1,2
Leader/Followers	15,53	10	19	12	12,9	1,18

EchoStringTest 16kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	30,86	10	294	13	13,31	1,61
Thread-Per-Request	28,09	12	266	20	19,62	3,84
Lazy Thread Pool	30,86	10	55	13	13,46	1,58
Half-Sync/Half-Async	26,32	11	320	23	23,56	2,97
Leader/Followers	30,49	11	32	13	14,34	2,49

EchoStringTest 16kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	58,14	10	56	14	16,27	4,41
Thread-Per-Request	50	13	326	28	28,67	7,98
Lazy Thread Pool	58,14	10	356	13	16,95	5,41
Half-Sync/Half-Async	43,86	10	392	38	37,73	8,43
Leader/Followers	50	10	352	26	26,88	9,45

EchoStringTest 16kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	75,88	10	422	46	47,61	15,1
Thread-Per-Request	55,64	13	497	74	92,36	45,45
Lazy Thread Pool	75,88	10	393	34	46,95	26,9
Half-Sync/Half-Async	73,65	10	411	38	50,56	27,36
Leader/Followers	67,68	10	392	62	62,16	18,34

EchoStringTest 16kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	73,88	10	2294	154	151,56	31,08
Thread-Per-Request	57,09	14	750	205	224,16	69,29
Lazy Thread Pool	73,88	11	924	113	131,07	58,82
Half-Sync/Half-Async	73,88	11	656	120	140,41	63,54
Leader/Followers	71,77	10	1136	137	155,47	58,13

EchoStringTest 16kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	73,53	11	3883	210	205,56	47,31
Thread-Per-Request	59,52	14	1003	257	276,21	70,92
Lazy Thread Pool	73,53	10	1129	151	183	91,76
Half-Sync/Half-Async	73,53	10	1100	157	186,33	89,17
Leader/Followers	73,53	11	739	193	206,25	71,72

EchoStringTest 32kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	14,62	15	27	17	17,63	1,42
Thread-Per-Request	14,04	17	50	19	19,51	1,4
Lazy Thread Pool	14,53	15	25	17	17,6	1,45
Half-Sync/Half-Async	14,45	16	301	17	18,09	1,72
Leader/Followers	14,62	15	29	17	17,73	1,64

EchoStringTest 32kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	29,07	15	39	17	17,99	1,69
Thread-Per-Request	25,51	17	51	26	26,71	5,62
Lazy Thread Pool	29,07	14	42	17	17,77	1,6
Half-Sync/Half-Async	25	15	335	28	28,18	6,03
Leader/Followers	27,78	14	342	18	20,42	4,09

EchoStringTest 32kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	50	14	241	25	27,35	8,01
Thread-Per-Request	44,64	17	401	35	37,58	11,19
Lazy Thread Pool	48,08	14	365	25	29,88	11,54
Half-Sync/Half-Async	40,98	15	389	44	45,85	14,86
Leader/Followers	42,37	15	383	41	42,03	14,34

EchoStringTest 32kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	53,28	16	2031	96	95,11	17,15
Thread-Per-Request	44,71	17	810	105	123,29	53,69
Lazy Thread Pool	50,08	14	617	73	96,62	54,81
Half-Sync/Half-Async	46,37	15	631	94	114,71	50,27
Leader/Followers	49,1	15	564	93	106,96	41,28

EchoStringTest 32kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	52,33	15	1047	241	233,47	58,51
Thread-Per-Request	41,18	19	1001	290	330,59	104,31
Lazy Thread Pool	48,31	15	1206	196	233,27	117,84
Half-Sync/Half-Async	46,52	15	1152	222	259,56	109,72
Leader/Followers	47,4	15	969	232	261,99	99,07

EchoStringTest 32kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	52,08	17	976	309	299,48	74,37
Thread-Per-Request	41,67	18	1293	377	423,46	113,57
Lazy Thread Pool	49,02	16	1414	244	287,27	141,01
Half-Sync/Half-Async	47,17	14	1258	281	319,89	133,94
Leader/Followers	47,17	15	1142	298	331,3	124,48

EchoStringTest 64kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	12,63	24	386	28	28,62	2,97
Thread-Per-Request	12,32	26	328	29	29,82	2,48
Lazy Thread Pool	12,69	24	320	27	27,81	2,49
Half-Sync/Half-Async	12,56	24	405	28	28,61	2,88
Leader/Followers	12,69	23	344	28	28,4	2,83

EchoStringTest 64kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	23,15	23	344	31	34,5	7,81
Thread-Per-Request	21,74	25	378	39	40,99	8,16
Lazy Thread Pool	23,81	24	340	29	32,4	6,78
Half-Sync/Half-Async	20	24	440	46	48,43	11,26
Leader/Followers	21,93	24	374	38	40,21	9,45

EchoStringTest 64kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	32,47	24	428	68	70,16	11,46
Thread-Per-Request	31,25	25	810	71	75,25	18,81
Lazy Thread Pool	30,49	23	590	61	74,45	33,95
Half-Sync/Half-Async	28,74	24	500	75	85,46	29,56
Leader/Followers	29,76	24	514	72	80,94	27,34

EchoStringTest 64kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,3	25	1500	191	191,82	41,64
Thread-Per-Request	27,52	26	1010	208	232,32	89,26
Lazy Thread Pool	27,82	23	922	175	214,94	109,09
Half-Sync/Half-Async	27,22	25	1026	190	226,84	95,54
Leader/Followers	28,13	24	925	184	219,4	92,69

EchoStringTest 64kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,01	25	1051	404	413,05	107,29
Thread-Per-Request	25,9	27	1652	486	553,12	166,89
Lazy Thread Pool	26,72	24	2010	398	455,73	218,18
Half-Sync/Half-Async	26,72	23	1913	431	492,56	191,42
Leader/Followers	26,72	24	1582	438	490,52	185,32

EchoStringTest 64kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	31,25	25	1727	496	522,2	151,68
Thread-Per-Request	26,04	25	2036	634	697,85	188,77
Lazy Thread Pool	26,88	24	2372	440	524,29	275,7
Half-Sync/Half-Async	26,6	26	2173	539	607,18	246,96
Leader/Followers	26,88	26	1971	555	614,85	238,3

EchoStringTest 128kb 1 Thread:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	7,58	45	415	88	80,93	18,21
Thread-Per-Request	9,51	47	376	52	53,89	4,17
Lazy Thread Pool	7,84	45	450	86	76,47	17,9
Half-Sync/Half-Async	9,62	45	388	52	52,78	4,24
Leader/Followers	8,77	45	381	54	62,89	15,73

EchoStringTest 128kb 2 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	13,09	46	484	95	101	22,68
Thread-Per-Request	14,2	47	625	74	88,82	26,96
Lazy Thread Pool	12,56	46	493	95	105,91	30,41
Half-Sync/Half-Async	14,37	45	600	75	86,95	25,21
Leader/Followers	15,15	45	563	70	79,35	20,49

EchoStringTest 128kb 4 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,45	87	546	171	186,04	38,52
Thread-Per-Request	14,97	47	776	178	210,24	82,57
Lazy Thread Pool	14,2	48	827	182	221,3	85,83
Half-Sync/Half-Async	14,45	52	1177	185	219,61	86,58
Leader/Followers	15,06	46	887	172	207	77,37

EchoStringTest 128kb 8 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,37	50	1270	396	427,52	79,45
Thread-Per-Request	13,61	49	1413	465	523,78	187,33
Lazy Thread Pool	13,54	60	1596	450	507,1	207,42
Half-Sync/Half-Async	13,46	52	1390	453	515,14	183,1
Leader/Followers	13,76	55	1474	444	508,05	188,12

EchoStringTest 128kb 16 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,31	61	1794	843	907,42	143,37
Thread-Per-Request	13,22	47	3266	1105	1121,82	331,14
Lazy Thread Pool	13,65	60	3053	939	1011,53	412,02
Half-Sync/Half-Async	13,65	47	3356	974	1029,22	348,41
Leader/Followers	13,29	85	3885	1041	1066,58	352,48

EchoStringTest 128kb 20 Threads:

Server-Variante	InvocationsPerSecond	Min	Max	Med	Avg	StdDev
Iterativ	16,23	100	2138	1099	1132,98	172,46
Thread-Per-Request	13,3	47	4479	1381	1392,72	357,37
Lazy Thread Pool	13,74	90	4560	1186	1264,28	527,71
Half-Sync/Half-Async	13,37	46	4290	1280	1321,36	452,18
Leader/Followers	13,37	46	5881	1288	1321,49	442,52