High Performance Client Server Infrastructure

Tobias Höfler
Betreuer: Philipp Fehre
Seminar Future Internet WS2011
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: hoeflert@in.tum.de

KURZFASSUNG

Die Client-Server Architektur ist trotz ihres Alters noch immer aktuell. Seit den Anfängen des Internets und insbesondere heute steigt die Anzahl an Clients immens schnell, was hochperformante und skalierbare Architekturen notwendig macht. Mögliche Ansätze, Client-Server Architekturen effizient zu realisieren werden in diesem Paper, mit den entsprechenden Grundlagen, behandelt.

Schlüsselworte

Sockets, I/O Modelle, Event, Node, iterativer Server, paralleler Server

1. EINLEITUNG

Das Client-Server Modell ist das vorherrschende Modell in verteilten Anwendungen. Hierbei bietet der Server gewisse Dienste und Daten an, während Clients diese in Anspruch nehmen. Somit besteht zwischen diesen beiden eine Produzenten-Konsumenten-Beziehung. Die immens wachsende Zahl an Teilnehmern in verteilten Anwendungen, nicht nur durch Computer sondern auch durch mobile Endgeräte, lässt die Zahl an Clients kontinuierlich wachsen, was eine effiziente Implementierung der Server notwendig macht. Dieses Thema wird in der vorliegenden Arbeit behandelt. Schon im Jahr 2003 beschäftige sich Dan Kegel mit diesem Problem ([6]), wobei dieses auch acht Jahre später nicht an Bedeutung verloren hat.
Kapitel 2 bildet die notwendigen Grundkenntnisse für die weiteren Kapitel. Kapitel 3 diskutiert danach mögliche Entwurfskonzepte für Client-Server-Architekturen mit deren Vor- und Nachteilen. In Kapitel 4 wird eine Einführung in die ereignisorientierte Programmierung gegeben inklusive der Vorstellung eines aktuellen Vertreters der dieses Paradigma verfolgt. Im letzten Kapitel 5 werden mit Beispielimplementierungen Geschwindigkeitstests durchgeführt und ausgewertet.

2. SOCKET PROGRAMMIERUNG

Zur Netzwerkprogrammierung werden typischerweise Sockets verwendet. Ursprünglich, für die erste BSD-Implementierung entwickelt, stellten diese die Grundlage für Implementierungen anderer Betriebssysteme dar. Sockets ermöglichen es Anwendungen über Transportschichtprotokolle wie TCP oder UDP miteinander über ein Netzwerk zu kommunizieren. Der Zugriff auf die Sockets selbst wird über Funktionen einer Programmbibliothek realisiert. Sie implementiert Funktionen wie *socket()* um einen Socket zu öffnen, *bind()* um den Socket an eine Adresse und einen Port zu binden, *listen()* um auf dem Rechner auf ankommende Verbindungen zu lauschen, *connect()* um eine Verbindung zu einem anderen Socket aufzubauen, *write()* um auf den Socket zu schreiben und um somit Daten an den Kommunikationspartner zu senden, *read()* um Daten vom Socket zu lesen und *close()* um eine Verbindung abzubauen. Der Socket selbst wird hierbei vom Betriebssystem bereitgestellt. Über diese Funktionen hinaus, existieren noch einige andere Funktionen die die Möglichkeiten mit Sockets zu interagieren erweitern. Auch werden verschiedene Sockettypen unterschieden. Der prinzipielle Ablauf einer Socket-Kommunikation unterscheidet sich, abhängig von der jeweiligen Rolle des Prozesses, das heißt ob dieser die Client- oder Serverseite realisiert. Eine Kommunikation mittels TCP bedeutet zum Beispiel für den Server, den Aufruf der Funktionen socket(), bind() und listen() um einen Socket zu kreieren, diesen an eine Adresse und einen Port zu binden und um auf eingehende Verbindungen zu warten. Die Clientseite ruft lediglich ein socket() und ein connect() auf um einen eigenen Socket zu erzeugen und eine Verbindung zum Server herzustellen. Nach einem serverseitigen accept() kann die eigentliche Kommunikation mittels read() und write() erfolgen. Abschließend rufen beide Prozesse die close() Funktion auf, um die jeweiligen Sockets zu schließen[4].
Die bisher eingeführten Funktionen und der schematische Kommunikationsablauf bilden die Basis für die folgenden Kapitel.

2.1 I/O Modelle

Der Zugriff auf Sockets aus der Anwendung heraus erfolgt stets mit Hilfe von Deskriptoren. Diese identifizieren den vom Betriebssystem erstellen Socket eindeutig und bieten somit die Kommunikationsschnittstelle zwischen Anwendung und Socket. Beim Zugriff auf diese, kann ein Prozess blockieren. Im Hinblick auf diese Blockierung werden im Folgenden fünf unterschiedliche Methoden für den Zugriff auf Sockets vorgestellt.

2.1.1 Blockierende I/O

Standardmäßig blockieren Sockets. Man spricht hierbei vom *blockierenden I/O* Modell. Dies bedeutet, dass ein Prozess der beispielsweise die Funktion *read()* aufruft, um Daten von einem Socket zu lesen blockiert, das heißt mit seiner Ausführung stoppt und so lange wartet, bis der Empfang der Daten abgeschlossen ist. In der Zwischenzeit kann der Prozess keinerlei weitere Aktionen durchführen.

doi: 10.2313/NET-2012-04-1_3

2.1.2 Nichtblockierende I/O

Dem Gegenüber steht die *nichtblockierende I/O*. Hierbei muss ein Socket explizit als „nicht blockierend“ gesetzt werden. Ruft ein Prozess nun die *read()*-Funktion in diesem Modell auf, wird er nicht blockiert, sondern bekommt vom Betriebssystem-Kernel eine Fehlermeldung zurück. Diese Fehlermeldung impliziert, dass die angeforderten Daten noch nicht vollständig übertragen wurden, somit dem Prozess noch nicht zur Verfügung stehen. Der Prozess wiederholt nun die Anfrage mithilfe von *read()* so lange, bis keine Fehlermeldung zurückgegeben wird, sondern ein „OK“. Damit sind die Daten übertragen und für den Prozess nutzbar. Diese wiederholte Anfragen wird auch *Polling* genannt. Einer der größten Nachteile ist hierbei der (unnötige) Verbrauch von CPU Zeit, also Rechenleistung.

2.1.3 I/O Multiplexing

Das dritte Modell ist das *I/O Multiplexing*. Dieses bezieht sich auf die *poll*, *select* und *epoll* Systembefehle welche grundsätzliche die gleiche Funktionalität liefern. Diese Funktionen blockieren genauso wie es bei dem *blockierenden I/O* Modell der Fall war. Im Unterschied zu den dort vorgestellten Mechanismen, werden den oben genannten Funkionen allerdings nicht nur ein Socket-Deskriptor, sondern gleich mehrere übergeben. Der jeweilige Befehl blockiert dann solange bis einer der registrierten Sockets bereit zum lesen oder schreiben ist. Das I/O Multiplexing bietet somit keine Vorteile, wenn der Prozess nur einen Socket- bzw. I/O-Deskriptor, sondern nur wenn er mehrere gleichzeitig verwaltet, ohne dass einer den kompletten Ablauf blockiert.
poll und *select* wurden etwa zur gleichen Zeit unabhängig voneinander entwickelt. *select* in BSD und *poll* in System V. *epoll* wurde später im Linux Kernel implementiert und ist vor Allem auf Skalierbarkeit optimiert[2].

2.1.4 Signalgesteuerte I/O

Das *signalgesteuerte I/O Modell* bietet eine weitere nichtblockierende Möglichkeit für den Zugriff auf Deskriptoren. Grundsätzlich verursachen Signale eine Prozessunterbrechung. Hat dieser Prozess eine Routine für das entsprechende Signal registriert, wird diese ausgeführt und der Prozess fortgeführt. Falls dies nicht der Fall ist, führt der Betriebssystemkernel die Standardroutine des jeweiligen Signals aus[9]. Im Kontext der Socket-Programmierung kann das *SIGIO*-Signal verwendet werden. Der Betriebssystem-Kernel informiert den Prozess über dieses Signal, sobald der Socket lese- bzw. schreibbereit ist. Hierzu registriert der Prozess eine Routine, die das *SIGIO*-Signal abfängt und die Daten vom Socket liest oder diesen beschreibt. Dies bedeutet, dass keine Blockierung stattfindet, solange der Socket nicht zum lesen oder schreiben bereit ist.

2.1.5 Asynchrone I/O

Bisher wurde das Lesen eines Sockets als einfache Operation angesehen. Tatsächlich geschieht jedoch noch ein Schritt zwischen dem Warten auf das vollständige Ankommen der Daten und deren Verfügbarkeit für den Prozess. Nachdem die Daten angekommen sind, müssen diese zuerst vom Kernel in den Userspace kopiert werden, damit der Prozess darauf zugreifen kann. Die bisherigen Methoden haben stets den Prozess informiert, sobald die Daten angekommen und im Kernelspace verfügbar sind, das Kopieren der Daten in den Userspace musste jedoch der Prozess selbst durchführen, was wiederum eine blockierende Aktion darstellt. Das *asynchrone I/O Modell* optimiert diesen Sachverhalt, indem der Prozess erst informiert wird, sobald die Daten vollständig angekommen und vom Kernel- in den Userspace kopiert sind. Somit ist dies ein Modell, was die komplette Interaktion mit Sockets nicht blockierend realisiert. Leider wird dieses Modell lediglich von einigen POSIX.1 Systemen unterstützt[7].

2.2 Zusammenfassung & Bewertung

Die hier vorgestellten Möglichkeiten mit Sockets zu interagieren bilden die Grundlage einer jeden Netzwerkprogrammierung. Falls diese Aktionen selbst nicht durch die verwendete Programmiersprache gekapselt werden, können hierbei durch geschickte Wahl des Modells Performancesteigerungen erzielt werden. Grundsätzlich werden blockierende und nichtblokierende I/O Modelle unterschieden, wobei der Prozess beim erstgenannten vollständig anhält um auf Daten zu warten. Für effiziente Implementierungen mit mehreren Clients ist grundsätzlich ein nichtblockierender Ansatz sinnvoll. Weitere Möglichkeiten der Interaktion mit Sockets bietet das I/O Multiplexing und die signalgesteuerte I/O. Bei beiden Modellen wird der Prozess informiert, sobald der Socket-Deskriptor bereit ist. Das signalgesteuerte Modell enthält leider einige Hürden für effiziente Implementierungen, da das *SIGIO-Signal* bei einigen Sockets nicht nur gesendet wird, sobald die Daten vollständig angekommen sind, sondern auch wenn diverse andere Kommunikationsereignisse auftreten. Das asynchrone I/O Modell steigert die Performanz noch um ein Weiteres, indem hierbei der aufrufende Prozess erst informiert wird, sobald die Daten vollständig für ihn verfügbar sind.

3. CLIENT-SERVER DESIGN

Wie in Kapitel 1 erwähnt, bedient ein Server typischerweise mehrere Clients. Dabei lauscht er immer auf eingehende Verbindungen (vgl. 2), bearbeitet diese und schickt dem entsprechenden Client eine Antwort. Hierbei hat die Art der Bearbeitung einer Anfrage enorme Auswirkungen auf die Effizienz des Server. Mögliche Arten werden im Folgenden beschrieben.

Die einfachste Art einen Serverprozess zu implementieren ist der sequentielle Fall. Hierbei bearbeitet der Server die Anfrage eines Clients, schickt die Antwort an diesen zurück und lauscht anschließend wieder auf neue Verbindungen. Diese Art von Server wird auch *iterativer Server* genannt wobei die ankommenden Anfragen nacheinander beantwortet werden.
Dem gegenüber steht der *parallele Server*. Er kreiert für jede einzelne Client-Anfrage mittels *fork()* einen neuen Kind-Prozess der die Anfrage dann bearbeitet und die Antwort an den Client schickt. Der Hauptprozess kann dabei sofort nach dem Erzeugen des neuen Prozesses neue Anfragen annehmen. Die Anzahl an Kind-Prozessen ist dabei ausschließlich durch das Betriebssystem begrenzt bzw. durch den verfügbaren Speicher.
Abbildung 1 zeigt beide Servertypen im Pseudocode. Ganz links ist der iterative Server, welcher eine Clientanfrage erhält, diese bearbeitet und die entsprechende Antwort zurück sendet. In der Mitte ist die Implementierung des parallelen Servers skizziert. Er erhält eine Anfrage des Clients, erzeugt

doi: 10.2313/NET-2012-04-1_3

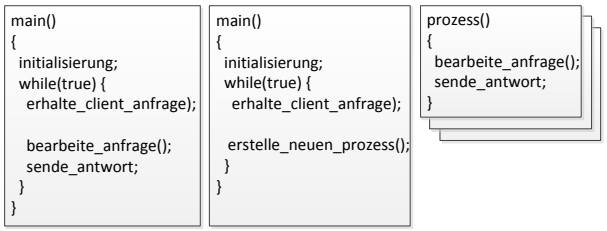

Abbildung 1: Iterativer vs. Paralleler Server (vgl. [1], S. 38)

einen neuen Prozess und kann sofort wieder neue Clientanfragen annehmen. Der erzeugte Prozess, rechts in der Abbildung, bearbeitet die Anfrage und sendet die Antwort an den Client.
Grundsätzlich ist bei mehreren Clients die Antwortzeit paralleler Server besser als die bei iterativen. Hinzu kommt hierbei allerdings die Zeit, die für das *fork()*, also die Erzeugung des neuen Kind-Prozesses entsteht bzw. allgemein die für die Prozesssteuerung notwendig ist. Diese entfällt natürlich beim *iterativen Server*[7].
Um einen *parallelen Server* effizient zu implementieren, muss die zusätzliche benötigte Zeit gegenüber dem *iterativen Server* möglichst klein sein. Aus diesem Grund werden häufig Threads anstatt Prozesse verwendet. Diese Thematik wird im Folgenden behandelt.

3.1 Prozesse und Threads

„Ein Prozess ist ein Programm in Ausführung“. Er ist Grundbaustein der Parallelität auf Rechnern. Hierzu wird ein Zeitmultiplexing der CPU durchgeführt, die, kontrolliert vom Betriebssystem, verschiedenen Prozessen Rechenzeit zuteilt. Um einen Prozess zu verwalten, nutzt das Betriebssystem den *Process Control Block (PCB)*, der alle notwendigen Informationen über den Prozess enthält. Hierzu zählen beispielsweise eine eindeutige Kennung, Prozesszustand, Prozesskontext und einige mehr. Wechselt das Betriebsystem nun den Prozess, dem aktuell die komplette Rechenleistung zugeteilt ist, müssen einige Schritte durchgeführt werden. Zuerst muss der Kontext des aktuell laufenden Prozesses gespeichert werden, das bedeutet dessen Informationen in einen PCB geschrieben werden. Dann wird anhand gewisser Algorithmen ein neuer Prozess ausgewählt (Scheduling). Der letzte Schritt besteht darin, die Informationen des PCBs des neuen Prozesses in die CPU zu kopieren[4]. Diese Prozessverwaltung kostet Ressourcen und Zeit, weshalb häufig Threads anstatt Prozesse für parallele Server verwendet werden. Ein Thread ist eine Aktivität innerhalb eines Prozesses. Ein Prozess beinhaltet immer mindestens einen Thread, wobei mehrere Threads in einem Prozess laufen können. Diese teilen sich eine gemeinsame Prozessumgebung, liegen also im selben Adressbereich und geöffnete Datei- oder Socket-Deskriptoren, Signale und einiges mehr. Sie besitzen jedoch einen eigenen Programmzähler, eigene Register und einen eigenen Stack. Da diese Informationen jedoch weitaus geringer sind, als die eines Prozesses, ist die Erzeugung eines Therads zehn bis hundert mal schneller als die eines Prozesses und die CPU kann deutlich schneller zwischen Threads umschalten, als zwischen Prozessen[12]. Deshalb werden Threads auch als leichtgewichtige Prozesse bezeichnet.
Zur Realisierung von Threads gibt es mehrere Möglichkeiten: Sie können auf Benutzer- oder Betriebssystem- bzw. Kernelebene oder zwischen diesen beiden Ebenen laufen. Bei der erstgenannten Methode übernimmt eine Bibliothek die Verwaltung der Threads. Somit ist das Betriebssystem außen vor und erkennt die Existenz mehrerer Threads nicht. Diese Variante ist einfach zu implementieren und besitzt einen Geschwindigkeitsvorteil gegenüber der zweiten Methode, da nie zwischen Benutzer- und Systemmodus umgeschaltet werden muss. Des weiteren hat eine große Anzahl an Threads kaum belastende Auswirkungen auf das Betriebssystem da dieses prinzipiell nichts von den Threads weiß, folglich auch keine Verwaltung übernehmen muss. Diese Methode besitzt allerdings auch Nachteile. So ist das Scheduling, also die Verteilung der CPU-Zeit auf die Threads nicht fair. Das Betriebsystem führt das Scheduling hierbei auf Prozessebene durch. Besitzt ein Prozess nun beispielsweise deutlich mehr Threads als ein anderer, bekommen trotzdem beide die gleiche Rechenzeit zugeteilt. Auch eine Priorisierung der Threads ist nicht möglich. Vorteile eines Mehrprozessorsystems können nicht ausgenutzt werden, da die Thread-Bibliothek diese nicht zur Kenntnis nehmen kann. Dies hat zur Folge, dass Threads in einer solchen Umgebung niemals echt parallel ablaufen können, sondern lediglich die Prozesse, in denen sie laufen.
Die zweite Variante ist die Nutzung von Threads auf Betriebssystemebene. Hierbei werden diese vom Betriebssystem selbst verwaltet, folglich fallen die eben genannten Nachteile der ersten Methode weg, was bedeutet, dass nun ein faires Scheduling stattfindet und auch eine Priorisierung vorgenommen werden kann. In einem Mehrprozessorsystem oder einer CPU die hardwareseitig Multithreading unterstützt kann nun auch eine echt paralleler Ablauf von Threads stattfinden.
Diesen Vorteilen stehen die Nachteile gegenüber, dass zwischen System- und Benutzermodus gewechselt werden muss, wenn ein neuer Thread angelegt wird. Weiterhin können sehr viele Threads das Betriebssystem belasten.
Die dritte Möglichkeit für Threads ist ein hybrider Ansatz, welcher im Solaris Betriebssystem implementiert ist. Hierbei werden Benutzer-Threads, Kernel-Therads zugeteilt, womit dieser Ansatz die Vorteile beider erstgenannten Methoden besitzt[1].

3.2 Threads - Vor- und Nachteile

In Kapitel 3.1 wurden hauptsächlich Vorteile von Threads gegenüber Prozessen dargestellt. Wie Kapitel 5 und auch [7] anhand verschiedenster Messungen belegt, sind Threadimplementierungen schneller als Prozessimplementierungen. Neben den in Kapitel 3.1 genannten Nachteilen existieren jedoch noch einige mehr.
Wie bereits erwähnt, laufen Threads innerhalb eines Prozesses im gleichen Adressraum. Wenn mehrere Threads parallel laufen, was bei einer Implementierung eines parallelen Servers der Fall ist, kann der Programmierer nicht vorhersagen, welcher Thread gerade läuft. Beim Zugriff auf gemeinsame Ressourcen können somit unvorhersehbare Ereignisse eintreten, weshalb Threads synchronisiert werden müssen. Diese Aufgabe kann unter Umständen schwierig werden und erfordert eine sorgfältige Programmentwicklung[9]. Die Parallelität des Servers macht das debuggen, also die Lokalisierung von Fehlhandlungen, auch schon bei einfachen Aktionen komplex. Auch das Testen der Implementierung ist sehr komplex, da eintretende Aktionen oder Fehler vom

doi: 10.2313/NET-2012-04-1_3

Scheduling abhängen, somit auch schwierig reproduzierbar sind. Durch die Nutzung von Threads auf Benutzerebene, ist die Portierung der Applikation auf andere Systeme sehr umständlich, da die jeweiligen Bibliotheken Betriebssystem-spezifisch sind.
Grundsätzlich macht die Verwendung von Threads die Programmentwicklung sehr komplex und ist häufig Quelle von Fehlern.[8].

3.3 Preforking und Prethreading

Ein Ansatz, zur weiteren Optimierung der Serverarchitektur ist das Preforking bzw. Prethreading. Hierbei wird, wie zu Beginn dieses Kapitels, ein Child Prozess bzw. ein Thread pro Client erzeugt. Hierbei gingen die Kosten dieser Erzeugung zu Lasten der Antwortzeit, da die Erzeugung erst mit einer Client-Anfrage durchgeführt wird. Diese Zeit, kann jedoch auch auf den Serverstart verlagert werden. Hierzu wird eine gewisse Anzahl an Prozessen bzw. Threads beim Start des Serverprozesses erzeugt. Diese können dann sofort neue Client Anfragen bearbeiten, ohne weitere Kosten zu verursachen. Im Folgenden wird lediglich auf die Implementierung mittels Threads eingegangen, da diese in der Regel schnellere Serverimplementierungen ermöglichen. Abbildung 2 zeigt ein solches Prethreading Szenario. Hierbei wurden n

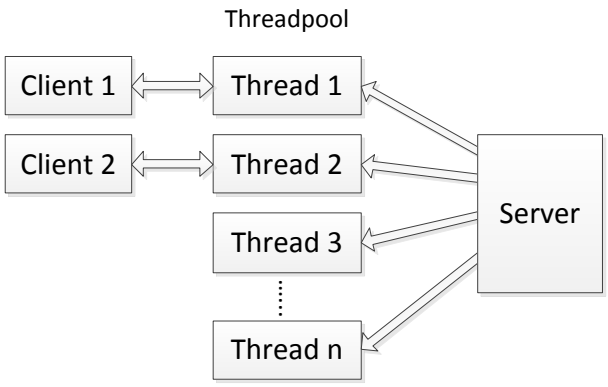

Abbildung 2: Prethreading (vgl. [7], S. 728)

Threads erzeugt wobei aktuell zwei Clients verbunden sind. Im Threadpool sind noch n-2 weitere Threads vorhanden, somit können problemlos n-2 weitere Clients bedient werden. Bei mehr als n Clients muss eine effiziente Server Implementierung weite Threads erzeugen bzw. die Notwendigkeit der neuen Erzeugung vorhersehen und schon durchführen, bevor alle Threads im Pool durch Clients belegt sind[7].

3.4 Zusammenfassung

Grundsätzlich werden iterative und parallele Serverimplementierungen unterschieden. Während der iterative Server Anfragen sequentiell bearbeitet, startet der parallele Server neue Prozesse oder Threads um die Aufgabe der Bearbeitung und Beantwortung der Anfrage zu übernehmen und kann selbst wieder auf neue Anfragen lauschen. Im Allgemeinen erlaubt die Verwendung von Threads eine performantere Implementierung als die Verwendung von Prozessen. Threads können hierbei in unterschiedlichen Ebenen, also im Benutzer- oder Systemmodus laufen, wobei beide Methoden Vor- und Nachteile besitzen, die für die jeweilige Implementierung genau abgewogen werden müssen. Als letztes wurde das Konzept des Preforking bzw. Prethreading eingeführt. Hierbei werden Prozesse bzw. Threads bereits im Vorfeld erzeugt um dies nicht erst während der eigentlichen Anfrage des Clients machen zu müssen. Dieses Konzept verbessert somit den parallelen Server.
Grundsätzlich macht die Verwendung von Threads das Programm sehr komplex und ist häufig die Quelle von Fehlern.

4. EREIGNISORIENTIERTE PROGRAMMIERUNG

Das folgende Kapitel führt das ereignisorientierte Programmierparadigma ein. Hierbei wird in Kapitel 4.1 eine Einführung in die Thematik gegeben. Darauf folgend wird Node vorgestellt, ein aktueller Vertreter dieser Sprachen.
Ein weiteres Beispiel für eine starke Orientierung an Ereignisse ist die Programmiersprache Erlang. Sie wurde speziell für die Entwicklung hochperformanter und paralleler Programme entwickelt. Anfangs konnte diese Sprache nur für Telekommunikationsanwendungen eingesetzt werden, da nur dort solche Anforderungen bestanden. Heute allerdings treten genau diese Anforderungen vermehrt auf, weshalb aktuelle Entwicklungen wie CouchDB oder auch ejabberd, worauf der Facebook-chat basiert, mit Erlang realisiert wurden.

4.1 Einführung

Im täglichen Leben treten Ereignisse sehr häufig auf, wie das Klingeln eines Telefons oder das Herunterfallen eines Schlüssels. Auf manche Ereignisse reagieren wir, während andere für uns uninteressant sind. Auch gibt es unerwartete Ereignisse wie einen Banküberfall oder auch ein Lottogewinn. Einige dieser Ereignisse sind sehr einfach zu beobachten, wie beispielsweise Dinge, die wir hören oder sehen, während andere nur durch vorhergehende Aktionen erkannt werden. So könnte das Ereignis „Aktien verkaufen“ auf das Lesen der Zeitung folgen.
In der Informationstechnik ist das Konzept der Ereignisse nicht neu. Sie treten beispielsweise in Form von Exceptions auf, die ein Programm unterbrechen um das Auftreten eines Ereignisses zu zeigen. Ein klassisches Beispiel wäre hier die Berechnung einer Division durch die Zahl Null. Ein weiteres Beispiel sind Benutzeroberflächen wie Java AWT, wo Mausklicks Ereignisse erzeugen und UI-Elemente wie Buttons darauf reagieren.
Obwohl Ereignisse etwas sehr natürliches sind und sie in der realen Welt sehr häufig anzutreffen sind, orientieren sich klassische Programmierparadigmen eher an synchronen Interaktionen, wie dem Anfrage-Antwort Muster. Hierbei führt ein Programm genau das auf Anfrage aus, wofür es geschrieben ist. Der ereignisorientierte Ansatz verfolgt hingegen das Ziel, dass Programme Ereignisse erkennen und selbständig entscheiden ob und wie sie darauf reagieren.[3].

4.2 Ereignisorientierte Programmierung

Verteilte Anwendungen, basieren häufig auf dem Anfrage-Antwort Interaktionsmuster. Dieses Muster ist sehr stark an die Client-Server-Architektur angelehnt, wobei der Server der Diensterbringer und der Client der Dienstnutzer ist, das heißt der Client stellt eine Anfrage und der Server beantwortet sie. Dieser Ablauf ist in Abbildung 3 in Form eines UML-Sequenzdiagramms dargestellt. Beispiele dieses Interaktionsmusters sind *REST*- und *SOAP-webservices* oder auch *remote procedure calls*. Eine Anfrage kann unterschied-

doi: 10.2313/NET-2012-04-1_3

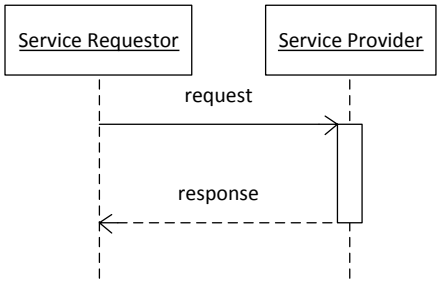

Abbildung 3: Anfrage Antwort Interaktionsmuster (vgl. [3])

licher Natur sein. So ist sie häufig eine Anfrage von Daten oder dem Ergebnis einer Berechnung wie *addiere drei und fünf und gibt das Ergebnis zurück* oder sie besitzt einen *update* Charakter wie *erhöhe die Menge an Bleistiften im Lager um 500*. Die dazugehörige Antwort gibt dann im ersten Beispiel die Zahl acht zurück während im zweiten Beispiel eine Antwort wie *Menge erfolgreich erhöht* sinnvoll ist. Vorteil von diesem Konzept ist, dass jeder Programmierer bereits damit vertraut ist. Außerdem passt dieses Interaktionsmuster häufig sehr gut in das Design von Applikationen. Trotzdem kommen Entwickler häufig an den Punkt, an dem eine Antwort nicht sofort geschickt werden kann, weil der Service Provider noch mit der Auswertung der Anfrage beschäftigt ist. Ein Beispiel hierfür wäre die Anfrage: *„Besitzt diese Instanz des Postschen Korrespondenzproblems eine Lösung? “*. Hierbei kann der Dienstanbieter eine Lösung liefern oder unendlich lange damit beschäftigt sein, folglich kann der Dienstnutzer im Zweifel unendlich lange auf die Antwort warten. Diese Form der Interaktion wird synchron genannt. Die meisten Anfrage-Antwort Interaktionen laufen synchron ab, das heißt der Sender ist so lange blockiert, bis er eine Antwort erhalten hat. Dieses Problem wird häufig mit *Timeouts* behandelt, das heißt es wird nur eine gewisse Zeit auf die Antwort gewartet. Danach wird die Anfrage abgebrochen. Die Verwendung von Ereignissen, kann hierbei aber sehr viel eleganter sein.

Ein Event bezeichnet etwas, was bereits passiert ist, während eine Anfrage eher die Aufforderung darstellt, dass etwas passieren soll. Wegen diesem Unterschied wird im Kontext von Ereignisse nicht von Dienstanbieter und Dienstnutzer, sondern von Ereignisproduzent und Ereigniskonsument gesprochen. Neben dem eben genannten, existieren noch weitere Unterschiede zwischen dem Anfrage-Antwort Muster und der eventgetriebenen Interaktion. Ein Ereignisproduzent, der ein Event an einen Konsumenten geschickt hat, erwartet nicht, dass dieser eine Aktion daraufhin durchführt. Sendet beispielsweise ein Sensor in einem Druckkessel laufend Ereignisse der Art „Mein gemessener Druck ist innerhalb der Grenzwerte“ so muss die Regelung nicht darauf reagieren. Des weiteren werden Ereignisse oftmals als einfache Nachricht gesendet, ohne dass eine Antwort des oder der Konsumenten erwartet wird. Dies bedeutet auch, dass der Ereignisproduzent nach dem Senden des Ereignisses sofort mit seinem Ablauf fortfahren kann, ohne auf eine Antwort zu warten. Dies wird *asynchrone Kommunikation* genannt. Abbildung 4 zeigt eine solche Interaktion. Hier ist zu sehen, dass der Ereignisproduzent die Kommunikation initi-

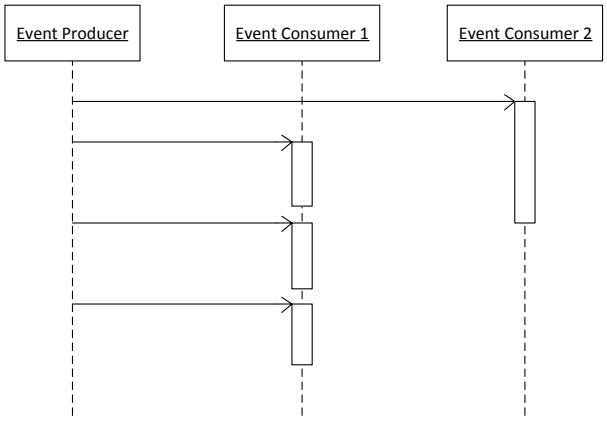

Abbildung 4: Eventtypische Push Interaktion (vgl. [3])

iert, sobald er ein Event zu verschicken hat. Man nennt dieses Vorgehen *Push*. Die Nachrichten die er versendet, sind Einwegnachrichten, es gibt also keine Antworten auf die er warten müsste. Diese Push-Benachrichtigung kann zu einer verbesserten Performance führen.
Um solche Push-Benachrichtigungen durchzuführen, muss der Ereignisproduzent über die Anzahl der Konsumenten und deren Adressierung informiert sein. Dies kann über die folgenden Mechanismen realisiert werden:

- Sämtliche, notwendige Informationen über die Konsumenten können fest im Ereignisproduzenten konfiguriert sein
- Der Ereignisproduzent findet die Konsumenten über einen Directory Service
- Die Ereigniskonsumenten müssen sich beim Produzenten registrieren, bevor sie Ereignisse erhalten

Jede dieser Methoden zieht natürlich eine höhere Belastung für den Ereignisproduzenten nach sich. Hierbei kann auch eine Middleware eingesetzt werden, die die Konsumenten verwaltet, einen sogenannten *Eventchannel*[3].
Die Konzepte der eventorientierten Programmierung gehen natürlich weit über das bisher eingeführte hinaus. Das bisherige reicht jedoch aus, um die Vorteile der Nutzung von Ereignisse bezüglich Client-Server Architekturen darzustellen.

4.3 Node

Node ist eine sehr neue, hochperformante Laufzeitumgebung für JavaScript. Im folgenden Kapitel wird diese genauer untersucht. Nach einer Einführung wird auf das Konzept des *Event-Loops* eingegangen und warum dieses so effizient ist. Hierbei wird eine beispielhafte TCP-Serverimplementierung mit Node genauer erläutert.

4.3.1 Einführung

Node ist eine JavaScript Laufzeitumgebung die es ermöglicht, JavaScript Programme außerhalb von Webbrowsern zu starten. Hierbei bietet es nicht nur die JavaScript-typische

doi: 10.2313/NET-2012-04-1_3

Funktionalität, sonder weitaus mehr, wie das verarbeiten von binären Datenströmen oder das Implementieren verschiedener Server. Node basiert auf der Google V8 Laufzeitumgebung die sehr performant, aber eben auf die JavaScript Funktionalität beschränkt ist. Deshalb wird diese mittels neuer APIs um POSIX Funktionen erweitert. Node nutzt eine Architektur Namens *Event-loop*, die in Kapitel 4.3.2 vorgestellt wird.
JavaScript selbst wurde im Jahr 1995 als eine einfache Scriptsprache zur Nutzung innerhalb von Webseiten für den Netscape Browser entwickelt. Als Webseiten immer dynamischer und schneller wurden und der Bedarf an Funktionalität innerhalb der Webseiten stieg wurde der Begriff *AJAX* populär, wobei das „J"in AJAX für JavaScript steht. Hierbei gab es auch keine wirkliche Auswahl, JavaScript war die einzige auf allen gängigen Browser verfügbare Programmiersprache. In diesem Zuge wandelte sich das Ansehen von Javascript von einer Sprache, die gerade gut ist um Webformulare zu überprüfen hin zu einer vollwertigen Programmiersprache. In letzter Zeit wurde darauf aufbauend etliche professionelle Frameworks entwickelt, wie jQuery, Dojo oder Prototype, die die Programmierung mit JavaScript erleichtern. Nahezu jeder Web-Programmierer hat schon einmal mit JavaScript gearbeitet, das heißt die Verbreitung und der Bekanntheitsgrad ist sehr hoch. Im Browser Wettstreit wurden darüber hinaus immer effizientere JavaScript Laufzeitumgebungen wie die V8 entwickelt, was nicht zuletzt ein Grund war, JavaScript auch auf die Serverseite zu portieren. Zu guter letzt ist JavaScript selbst bereits ereignisorientiert, ein weiterer Grund für die Entwicklung von Node[5].

4.3.2 Event-Loop

Ein grundlegender Teil der Node Architektur ist die *Event-loop*. Im Gegensatz zu vielen anderen Sprachen, bei denen die Nutzung von Ereignisse im Nachhinein dazuentwickelt wurde, ist sie in JavaScript bereits von Anfang an ein Kernkonzept. Dies liegt an der eigentlichen Aufgabe von JavaScript mit Benutzern zu interagieren. Jeder der bereits HMTL-Formulare entwickelt hat, kennt Ereignisse wie *onmouseover* oder *onklick*. Die *Event-loop* ist hierbei ein Konzept, um recht einfach mit solchen Ereignisse zu arbeiten. Dieses Konzept wurde nun von Node aufgegriffen und erweitert. Läuft JavaScript nämlich innerhalb einer Browsers, so gibt es nur eine beschränkte Anzahl an möglichen Ereignisse, wird es allerdings auf den Server portiert, wie es bei Node der Fall ist, kommt eine große Anzahl an Ereignisse hinzu, wie das Ankommen einer Anfrage an einen TCP-server oder das Ergebnis einer Datenbankabfrage.
Die Grundidee bei Node ist, dass alle I/O Aktivitäten nichtblockierend sind. Was bedeutet, dass beim Warten eines TCP-Servers auf eingehende Verbindungen, oder beim Zugriff auf eine Datenbank oder eine einfache Textdatei, das Programm nicht hält und wartet bis ein Ergebnis vorliegt, sondern sofort weiterläuft. Wenn die Daten bereit sind, sprich eine TCP-Anfrage bearbeitet ist oder die Datenbank die Ergebnisse der Abfrage bereit hält wird ein Event erzeugt[5]. Intern wird dies mit den in Kapitel 2.1 vorgestellten select, poll und epoll Funktionen realisiert. Dies bedeutet, dass der Betriebssystemkernel die Anwendung informiert, sobald die Daten bereit sind[10]. Programmiertechnisch ist dies eine Form der asynchronen Kommunikation (vgl. Kapitel 4.2) und wird mit sogenannten callback-Funktionen realisiert. So wird beispielsweise beim Erzeugen eines TCP-Servers, gleich eine Funktion mitübergeben, die aufgerufen wird, sobald ein neue Anfrage ankommt. Diese Funktion heißt callback Funktion. Sie reagiert somit auf die Ereignisse des Betriebssystems. Ein Beispiel hierzu ist in Kapitel 4.3.3 erläutert. Der prinzipielle Ablauf eines Node-Programms zeigt hierbei die Funktionsweise der *Event-loop* recht einfach. Zuerst läuft das Programm komplett von Anfang bis Ende ab. Dies kann als eine Art von Initialisierung aufgefasst werden. Hierbei werden sämtliche *Event-Listener*, sprich die callback-Funktionen registriert. Beinhaltet das Programm keine *Event-Listener* beendet sich das Programm, sonst allerdings bleibt es aktiv bzw. wird vom Betriebssystem schlafen gelegt, wartet auf eingehende Ereignisse und ruft anschließend die jeweiligen Funktionen auf.
Ein Kritikpunkt an Node ist, dass es *single-threaded* ist. Das bedeutet, dass das komplette Programm in einem einzige Thread abläuft. Somit werden die Vorzüge von Mehrprozessor- oder Multithreadingsystemen nicht genutzt. Prinzipiell wäre eine Erweiterung möglich, die das Erstellen mehrerer Threads erlauben, jedoch steigt hiermit die Komplexität des kompletten Systems und somit auch die für den Programmierer. Da Node jedoch niemals auf I/O wartet, ist die Zeit die zwischen dem Ankommen eines Ereignisse und dessen Bearbeitung stets sehr gering[5].

4.3.3 Beispiel: TCP-Server

Um die Funktionsweise von Node und der *Event-loop* zu verdeutlichen, ist in Abbildung 5 eine beispielhafte Implementierung eines TCP-Echo-Servers dargestellt. Er antwor-

```
var net = require('net');

var server = net.createServer(function (socket) {
  socket.write("Echo server\r\n");
  socket.pipe(socket);
});

server.listen(1337, "127.0.0.1");
```

Abbildung 5: Beispielhafte TCP-Server Implementierung mit Node.js

tet auf jede einkommende Anfrage schlichtweg mit dem vom Client eingegebenen Text. In Zeile eins wird hierzu das Modul *net* geladen, welches die Implementierungen eines TCP-Servers enthält. Danach wird mit *createServer()* ein neuer Server erzeugt. Hierbei wird als Funktionsparameter die callback Funktion übergeben. In diesem Beispiel handelt es sich um eine anonyme Funktion. Beim Aufruf der callback Funktion wird wiederum ein Parameter übergeben. Dies ist der Socket, mit dem der Client verbunden ist. Die Funktion selbst schreibt beim Aufruf nun einfach den String „Echo server" und den vom Client eingegebenen Text auf den Socket. Die letzte Zeile des Beispielcodes veranlasst den Server mit der lokalen IP Adresse auf dem Port 1337 zu lauschen. Nachdem das Programm nun einmal durchgelaufen ist, sozusagen die Initialisierung beendet hat, wird der Thread vom Betriebssystem schlafen gelegt. Er verbraucht somit keine Rechenleistung. Meldet das Betriebssystem nun das Ereignis einer neue Anfrage, reagiert das Programm mit dem Aufruf der callback Funktion, antwortet also mit „Echo server" und

doi: 10.2313/NET-2012-04-1_3

dem jeweiligen Text und wird anschließend vom Betriebssystem wieder schlafen gelegt.

4.4 Zusammenfassung

Ereignisse begegnen einem im Alltag sehr häufig, weshalb sie sehr natürlich sind. Im Gegensatz zum Anfrage-Antwort Interaktionsmuster, werden Ereignisse als Einwegnachrichten versendet. Der Ereignisproduzent erwartet also keine Antwort. Dies wird auch als *Push* bezeichnet. Der Ereigniskonsument entscheidet selbst, ob und mit welcher Aktion er auf ein Event reagiert, da nicht alle Ereignisse eine Reaktion erfordern.
Node ist eine neue Laufzeitumgebung für JavaScript, die auf Ereignissen basiert. Sie nutzt hierbei das Konzept der Event-loop, die nach einer ersten Registrierung sämtlicher Ereignisse nur noch auf eingehende Ereignisse wartet, wobei das Programm selbst schlafen gelegt wird. Kommt ein solches Ereignis an, wird eine speziell dafür registrierte callback Funktion aufgerufen. Diese reagiert somit auf eintreffende Ereignisse. Durch dieses Konzept, ist sämtliche I/O Operation in Node nicht blockierend, was enorme Performancesteigerungen nach sich zieht. Hierbei wird ein Node Programm nur in einem Thread ausgeführt, was ein häufiger Kritikpunkt an diesem Konzept ist.

5. PERFORMANCEVERGLEICH

In diesem Kapitel werden einige der vorgestellten Konzepte verglichen. Hierzu wurden verschiedene TCP Server in Python und Node umgesetzt deren Quellcodes in Kapitel A dargestellt sind. Die Server liefen hierbei auf einem Multicore Atom Rechner mit dem Betriebssystem FreeBSD 8.2. Die Anfragen wurden von einem Linux Rechner (Ubuntu 11.10) über ein Gigbit Ethernet gesendet. Zur Messung der Zeiten wurde das Apache Benchmark Tool in der Version 2.3 Revision 655654 eingesetzt wobei jeder Test mehrmals durchgeführt wurde. Die im Folgenden präsentierten Werte entsprechen den Mittelwerten dieser Durchläufe.

5.1 Einfacher HelloWorld-Server

Der erste Test wurde mit einfachen HelloWorld-Servern durchgeführt, welche schlicht den String *HelloWorld* zurück geben. Hierbei wurden eintausend Anfragen an den Server gesendet, wobei zehn Verbindungen gleichzeitig genutzt wurden. Tabelle 1 zeigt die entsprechenden Messergebnisse. In der ersten Spalte die Gesamtzeit des Tests, in der zweiten die Anzahl an Antworten des Servers pro Sekunde.

	t [s]	req/t [1/s]
Iterativer Server	1.1148	897.928
Paralleler Server - Prozesse	4.4712	220.562
Paralleler Server - Threads	1.8856	530.664
Node Server	1.3624	743.41

Tabelle 1: Performancevergleich, „HelloWorld“ Server

Auffällig ist, dass der iterative Server am schnellsten ist, da hier der gesamte Test in der kürzesten Zeit ablief und dieser Server am meisten Antworten pro Sekunde schicken konnte. Dieser Test ist dahingehend realitätsfern, da er schlicht einen kurzen String zuzrückliefert, was eher selten anzutreffen ist. Es werden serverseitig also keinerlei Berechnungen oder Zugriffe auf andere Ressourcen durchgeführt. Deshalb ist es auch nicht verwunderlich, dass der iterative Server hier am schnellsten abschneidet. Er muss weder neue Threads oder Prozesse kreieren, noch auf Events reagieren oder callback Funktionen aufrufen. Bei den parallelen Servern ist der mit Theads realisierte Server deutlich schneller als der mit Prozessen. Hierbei ist der Performancevorteil von Threads gegenüber Prozessen deutlich zu sehen. Der Server, der mit Node realisiert wurde, ist zwar langsamer als der iterative, jedoch den parallelen Servern deutlich voraus.

5.2 Berechnungen auf der Serverseite

Ein realitätsnahes Szenario wurde in diesem Test untersucht. Hierbei wird auf der Serverseite bei jeder Anfrage hundert Millisekunden gewartet bis die Antwort geschickt wird. Dies soll eine Aktion simulieren, wie beispielsweise eine Berechnung, einen Datenbankzugriff oder den Zugriff auf externe Ressourcen. Dieser Test wurde mit insgesamt hundert Anfragen durchgeführt, mit jeweils fünf parallelen Verbindungen. Tabelle 2 zeigt die entsprechenden Ergebnisse. Wie beim vorherigen Test die Gesamtzeit, und die Antworten pro Sekunde. Darüber hinaus ist hier Zeit pro Anfrage dargestellt.

	t [s]	req/t [1/s]	t/req [ms]
It. S.	10.461	9.56	104.606
P. S. - Prozesse	2.443	40.94	122.146
P. S. - Threads	2.306	43.373	115.288
Node S.	2.251	44.443	112.522

Tabelle 2: Performancevergleich, Server mit Berechnung

Die Gesamtzeit des Tests des iterativen Servers liegt bei knapp über zehn Sekunden, was nicht verwunderlich ist, da alle hundert Anfragen nacheinander abgearbeitet werden, wobei jede einhundert Millisekunden dauert. Auch für die parallelen und den Node Server ist die Gesamtzeit interessant. Der Optimalwert wäre hier zwei Sekunden, da je fünf Anfragen parallel geschickt wurden. Wie an den Ergebnissen zu sehen ist, ist der Node Server diesem Wert am nächsten. Auch hier ist der parallele Server mit Threads schneller als der mit Prozessen. Der Unterschied der drei letztgenannten Server ist in diesem Beispiel gering. Er steigt jedoch mit der Anzahl an parallelen Verbindungen, da genau hier die Stärken der parallelen Server dem iterativen bzw. des Node-Servers den parallelen gegenüber liegt.

6. FAZIT

Zum Entwurf hochperformanter Client-Server Architekturen bedarf es vor allem an Kenntnissen in den Programmiergrundlagen und den I/O Modellen. Diese wirken sich stark auf die Effizienz des Servers aus. So können je nach gewähltem Modell Prozesse vollständig blockieren, während sie auf I/O warten, oder auch benachrichtigt werden sobald die jeweiligen Ressourcen verfügbar sind.
Die Art der Serverimplementierungen ist eine weitere wichtige Eigenschaft effizienter Realisierungen. Hierbei erwies sich der parallele Server als eine Möglichkeit, Interaktion mit vielen Clients schneller zu ermöglichen, als die Anfragen sequentiell zu bearbeiten. Die Nutzung von Threads, die leichtgewichtiger als Prozesse sind erwies sich als eine Möglichkeit, den parallelen Server performant zu implementieren. In Kapitel 5 wurde deutlich, dass bei mehreren Clients

doi: 10.2313/NET-2012-04-1_3

der parallele Server deutlich schneller ist als der iterative. Eine weitere Verbesserung bietet dabei die Ausnutzung von Prethreading bzw. Preforking, um die Erzeugung der Prozesse bzw. Threads nicht auf Kosten der Antwortzeit durchzuführen.
Die Nutzung von Threads birgt jedoch auch Nachteile. So werden Programme, die diese Nutzen, sehr komplex und das Debuggen und Testen gestaltet sich als äußerst schwierig. Des weiteren ist eine sorgfältige Wahl des Threadmodells notwendig. Die Kosten der Erzeugung von Threads ist ein weiterer Nachteil.
Ein aktueller Ansatz, der die komplizierte und aufwendige Nutzung von Threads vermeidet, ist der eventgetriebene. Als ein Vertreter dieses Paradigma wurde Node vorgestellt. Es folgt strikt dem Ansatz, sämtliche I/O als nichtblockierend zu realisieren. Dabei werden I/O Operationen mit callback Funktionen ausgestattet, die aufgerufen werden, sobald die Operationen selbst beendet sind. Dadurch muss das eigentliche Programm, niemals auf Daten warten. Dies bedeutet auch, dass weder Prozesse noch Threads erzeugt werden müssen. Dieser Vorteil wurde auch in Kapitel 5 deutlich, indem eine Serverimplementierung in Node in den meisten Fällen den anderen Konzepten überlegen war.

7. LITERATUR

[1] G. Bengel: *Grundkurs verteilte Systeme*, S. 37-44, Vieweg+Teubner Verlag, 2004

[2] J. Corbet, A. Rubini, G. Kroah-Hartman: *Linux device drivers*, S. 163-165, O'Reilly Media, Inc., 2005

[3] O. Etzion, P. Niblett: *Event Processing in Action*, S. 3-58, Manning Publications Co., 2011

[4] E. Glatz: *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*, S. 91-98, 293-301, dpunkt.verlag., Heidelberg, 2006

[5] *Preview of Node: Up and Running: Scalable Server-Side Code with JavaScript* , `http://ofps.oreilly.com/titles/9781449398583/index.html`

[6] *The C10K problem* , `http://www.kegel.com/c10k.html`

[7] R. W. Stevens: *Programmieren von UNIX-Netzwerken*, S. 139-146, 719-737Carl Hanser Verlag, München, Wien, 2000

[8] *Advantages and Disadvantages of a Multithreaded/Multicontexted Application*, `http://download.oracle.com/docs/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm`

[9] J. Wolf: *Linux-UNIX-Programmierung*, S. 255-257, 367-371, Galileo Press, 2009

[10] *The Node.js Website*, `http://nodejs.org/l`

[11] J. Armstrong: *A History of Erlang*, Ericcson AB, S. 1, 16-20, ACM New York, NY, USA, 2007

[12] A. Tanenbaum: *Moderne Betriebssysteme, 3. Auflage*, S. 137, Pearson Studium, München, 2009

ANHANG

A. PROGRAMM-QUELLCODES

A.1 Iterativer TCP Server in Python

```
import SocketServer
import time

class MyTCPHandler(SocketServer.BaseRequestHandler):
  def handle(self):
    time.sleep(0.1)
    self.request.send("HelloWorld\n")

server = SocketServer.TCPServer(("192.168.50.5", 9999),
   MyTCPHandler)
server.serve_forever()
```

A.2 Paralleler TCP Server in Python mit Threads

```
import socket
import threading
import SocketServer
import time

class ThreadedTCPRequestHandler(
   SocketServer.BaseRequestHandler):

  def handle(self):
    time.sleep(0.1)
    self.request.send("HelloWorld\n")

class ThreadedTCPServer(SocketServer.ThreadingMixIn,
   SocketServer.TCPServer):
  pass

server = ThreadedTCPServer(("192.168.50.5", 9997),
   ThreadedTCPRequestHandler)

server_thread = threading.Thread(
   target=server.serve_forever)
server_thread.daemon = False
server_thread.start()
```

A.3 Paralleler TCP Server in Python mit Prozessen

```
import os
import SocketServer
import time

class ForkingTCPRequestHandler(
   SocketServer.BaseRequestHandler):

  def handle(self):
    time.sleep(0.1)
    self.request.send("HelloWorld\n")

class ForkingTCPServer(SocketServer.ForkingMixIn,
   SocketServer.TCPServer):
  pass

import socket
import threading

server = ForkingTCPServer(("192.168.50.5", 9998),
   ForkingTCPRequestHandler)

server_thread = threading.Thread(
   target=server.serve_forever)
server_thread.daemon = False
server_thread.start()
```

A.4 Node TCP Server

```
var net = require('net');

var server = net.createServer(function (socket) {
  setTimeout(function () {
    socket.end("HelloWorld\n");
  }, 100)
});

server.listen(9996, "192.168.50.5");
```


doi: 10.2313/NET-2012-04-1_3