Prof. Dr. Th. Letschert

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

NVP – Nebenläufige und Verteilte Programme

Aufgabenblatt 12

Aufgabe 1

Mit *NodeJs*[1] hat ein reaktives Framework in letzter Zeit beeindruckende Popularität gewonnen.

NodeJs basiert auf *V8*, dem JavaScrip-Interpreter / virtuelle Maschine von Google. Es stellt eine Platform dar, auf der einfache ereignisgetriebene Anwendungen – also zum Beispiel Web-Server / Web-Applikationen – realisiert werden können. Die Implementierung besteht aus C/C++–Code, der die relevanten Funktionalitäten (Socket-I/O, Selector, etc.) des Betriebssystems über eine JavaScript-API, *Socket.IO*[2] zur Verfügung stellt.

Da in NodeJs ein C/C++–Herz schlägt, wird natürlich keine Java-NIO verwendet. Die technische Lösung ist aber stärker vom Betriebssystem, als von der Programmiersprache abhängig. Informieren Sie sich grob(!) über den Kommunikations-Kern von NodeJs, die *uv*-Bibliothek[3] :

1. Was sagt Bert Belder in seinem Vortag über `libuv`[4] zum *select*–Systemaufruf?
2. *select* kommt aus den *Berkeley Sockets*[5], der ersten Socket-Implementierung in 4.2BSD Unix (1983). – Ja richtig harter Code auf den sich alle setzen und verlassen ist oft recht alt.

   select ist nicht mehr auf jedem System die beste Methode um Ereignisquellen auf Bereitschaft abzufragen. In Linux gilt beispielsweise *epoll* (seit 2002 im Kernel) als der neuste coolste Systemaufruf für diesen Zweck.

   Wie sieht es mit NIO aus. Kommt es irgendwelche Systemaufrufe in C aus?

   Wenn Systemaufrufe genutzt werden, basiert es dann immer noch auf dem alten *select* von Berkley-Unix?

   Finden Sie heraus welche *select*–Implementierung in NIO auf Ihrem System (z.B. einem OSX-System) verwendet wird. Werfen Sie dazu einen Blick in den Quellcode des *DefaultSelectorProviders* für beispielsweise OSX von OpenJDK.[6]
3. Entspricht `libuv` dem *Reactor-Pattern*? Wird *Staging* unterstützt?
4. Ist `libuv` mit Netty vergleichbar, oder mit purer NIO, oder ist es etwas völlig anderes als beide?

[1] https://nodejs.org/en/
[2] https://socket.io
[3] https://github.com/libuv; http://nikhilm.github.io/uvbook/
[4] https://www.youtube.com/watch?v=nGn60vDSxQ4
[5] https://en.wikipedia.org/wiki/Berkeley_sockets
[6] http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/macosx/classes/sun/nio/ch/DefaultSelectorProvider.java

Aufgabe 2

"Nicht blockierende" Anwendungen oder *Anwendungen mit "reaktiver Kommunikation"* sind solche, bei denen Threads nicht in I/O-Operationen blockieren und sich nicht gegenseitig bei synchronisierten Aktionen blockieren. Statt dessen werden sie informiert, wenn Daten verfügbar sind und Synchronisationsbedarf wird so weit wie möglich eliminiert.

In einem einfachen Szenario wird bei der Kommunikation ein Systemaufruf wie *select* ausgeführt, der gleichzeitig mehrere Kommunikations-Schnittstellen überwacht. Ein (einziger) Thread – der eine *"Event Loop"* ausführt – wird informiert und behandelt die Daten. *NodeJs* ist das Paradebeispiel für ein solches System.

Erläutern Sie: Warum und wann ist diese Systemarchitektur von Vorteil:

- Immer, oder
- vor allem bei, oder
- nur bei stark I/O-lastigen Anwendungen wie einem Webserver (Netzwerk-Kommunikation + DB-Kommunikation)

und in wie weit spielt die Unterstützung asynchroner I/O durch die Plattform eine Rolle?

Aufgabe 3

Ein oft unterschätzter Aspekt reaktiver Architekturen ist höhere Komplexität der Software. Threads mit blockierenden Aufrufen wurden erfunden, um mit einem einfachen Kontrollfluss arbeiten zu können, trotz nicht-deterministisch auftretender Ereignisse. Das System ordnet den realen Ablauf entsprechend dem Eintreffen der Ereignisse. Die Algorithmen der Ereignisverarbeitung sind weitgehend unabhängig davon entsprechend logischer Verarbeitungsreihenfolgen organisiert. In einer "reaktiven Anwendung" richtet sich die Systemstruktur dagegen nach den Ereignissen, der Zusammenhang von Ereignissen geht verloren und muss anderweitig rekonstruiert werden.

Nehmen wir als Beispiel eine sitzungsorientierte Anwendung, bei der Clients Faktorisierungs-Anfragen an einen Server senden. Der Server antwortet mit der Primfaktorzerlegung. Er verwendet einen Cache um die Anfragen zu bedienen. Am Ende einer Sitzung sendet er die Rechnung: Die Kosten aller Anfragen während einer Sitzung. Als Kosten $k(n)$ der Zerlegung einer Zahl n wird deren Logarithmus angesetzt $k(n) = \log n$.

Der Client profitiert dabei nicht vom Cache: ihm wird $k(n)$ Berechnung in Rechnung gestellt, auch wenn das Ergebnis im Cache zu finden war. Die Anwendung soll TCP verwenden. Eine Sitzung wird auf eine Verbindung abgebildet.

In dieser Anwendung haben wir ein Sitzungskonzept und damit einen Zustand der Kommunikation: die aufgelaufenen Faktorisierungskosten der Sitzung. Zudem gibt es einen Synchronisationsbedarf: auf den Cache wird von jeder Sitzung zugegriffen. Eine "reaktive" Lösung hat einen einzigen Thread und Synchronsiation ist darum kein Thema. Der Zustand einer Sitzung muss aber explizit verwaltet werden. Eine Thread-orientierte Lösung hat es leichter mit der Zustandsverwaltung, muss aber Synchronisation in Betracht ziehen.

Reaktive Server basieren *technisch* auf BS-Mechanismen wie dem *selector*-Systemaufruf und asynchroner I/O. Die Kontrolle geht damit stets von einer *"Event-Loop"* aus.

Um einigermaßen bequem anwendbar zu werden, setzt man Frameworks ein. Frameworks basieren auf Mustern.

Aktuelle Frameworks für reaktive Server, wie etwa Netty, unterstützen das *Reactor-Muster*[7] das *Acceptor-Connector* Muster[8] und das *Staged Event-Handling Muster*[9]

Reactor Das *Reactor*-Muster hat folgende Bestandteile:

- *Resources* Das sind die Kommunikationskanäle, die Quellen und senken von Ereignissen / Nachrichten.
- *Event Demultiplexer* Diese Komponente verteilt Ereignisse an die zuständigen *Request Handler*
- *Request Handler* verarbeiten dann das Ereignis / die Nachricht in anwendungsspezifischer Art.

[7] https://en.wikipedia.org/wiki/Reactor_pattern
[8] http://www.cs.wustl.edu/schmidt/PDF/Acc-Con.pdf
[9] https://en.wikipedia.org/wiki/Staged_event-driven_architecture; http://www.mdw.la/papers/mdw-phdthesis.pdf

Acceptor–Connector Das *Acceptor–Connector* Muster beschreibt die konzeptionelle Trennung zwischen der Initialisierung eines Dienstes und dem mit dem Dienst verbunden Service. Platt gesagt realisiert eine Komponente *accept* eine andere behandelt die Verbindung, die durch ein *accept* initiiert wird.

Staged Event Handling Beim *Staged Event Handling* geht es darum, die Ereignisbehandlung in Stufen oder Schritte aufzuteilen. Jede der Stufen kann dabei ihre eigene Threading-Strategie verfolgen.

Vereinfacht hat man folgende Architektur (Grafik nach Doug Lea: *Scalable IO in Java*[10]):

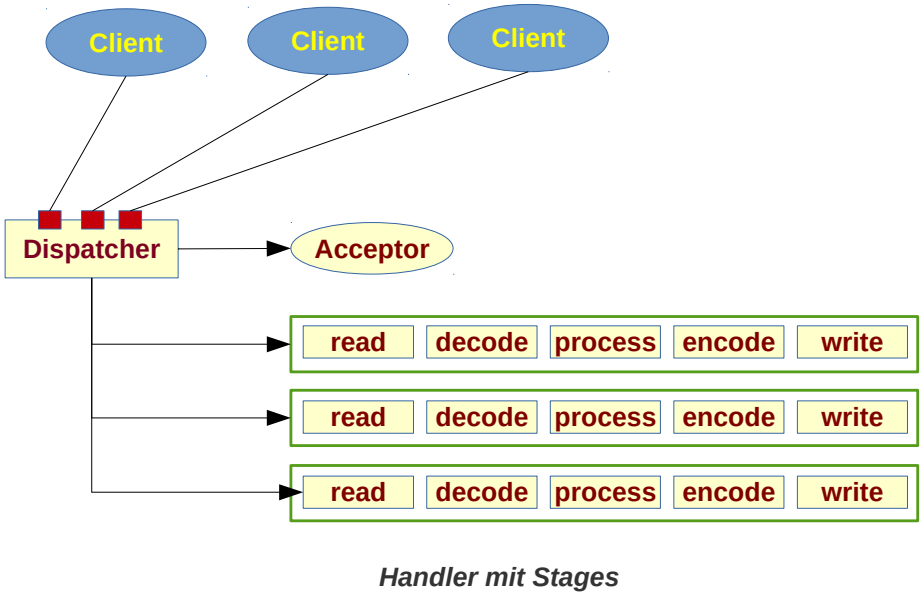

Abbildung 1: Reactor-, Acceptor-Connector-, Staging-Muster

Auf das *Staging* wollen wir nicht weiter eingehen.

Um einen vagen Einblick in die Mühen der Systemprogrammierung und um ein Gefühl für das reaktive Programmierparadigma zu bekommen, soll der Rest eines reaktiven Servers prototypisch "zu Fuß" mit Java-Bordmittlen implementiert werden.

Wir denken dazu "reaktiv", d.h. ausgehend von den Ereignissen: In der blockierenden Variante lesen Threads Daten von "ihrer" Socketschnittstelle:

```
val pw = new PrintWriter(sock.getOutputStream())
val stream = new InputStreamReader(sock.getInputStream())
val reader = new BufferedReader(stream)

while (true) {
  val msg = reader.readLine()
  ...
}
```

In der reaktiven Variante muss auf das Ereignis *"Irgendwo treffen irgendwelche Daten ein"* reagiert werden. Hier wird es schon mühsam, denn bedauerlicherweise kommen die Daten nicht unbedingt in den Häppchen, in den denen wir sie gerne verarbeiten würden. Es ist also nicht gesagt, dass da eine Zeile kommt. Es kann mehr kommen oder auch weniger.

Noch schlimmer: Es sind Bytes, nicht Zeichen, die da über die Schnittstelle eintreffen. Nach der Vertreibung aus dem ASCII-Paradis vor vielen vielen Jahren sind Bytes keine Zeichen. Ein Zeichen benötigt in der Regel mehrere Bytes.

[10] http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

Vielleicht sind gerade die ersten Bytes eines Zeichens eingetroffen, das letzte fehlt noch. In der blockierenden Variante beschäftigt sich der *BufferedReader* mit dieser Thematik. Er hat es aber auch leichter damit, er kann einfach warten, bis genug Daten da sind und wenn zu viele da sind, liefert er nur die aus, die gebraucht werden und puffert den Rest.

Bytes müssen gepuffert werden bis ein vollständiges Zeichen erkannt wurde und Zeichen müssen gepuffert werden, bis eine Zeile erkannt wurde.

1. Skizzieren Sie (informal) einen Algorithmus mit Threads, synchronisierten Puffern und blockierenden Aktionen mit dem eintreffende Bytes zu Zeilen zusammengefügt werden und diese dann einer Anwendung zur Verfügung gestellt werden.
2. Skizzieren Sie (informal) einen Algorithmus mit der gleichen Aufgabenstellung, der mit einer *Event-Loop* und ohne blockierenden Aktionen und Synchronisation arbeitet.
3. Informieren Sie sich über die Zeichencodierung von *UTF-8*. Wieviele Bytes machen ein Zeichen aus.
4. In der Java-API gibt es die Klasse *java.nio.charset.CharsetDecoder*, die bei der Umsetzung von Bytes zu Zeichen hilft. Nutzen Sie diese Klasse für einen NIO-Server als Demo-Anwendung, der eintreffende Bytes entsprechend einer vorgegebenen Zeichencodierung – z.B. UFT-8 – zu Zeilen zusammensetzt und auf der Konsole ausgibt.
5. Implementieren Sie einen reaktiven cachenden Faktorisierungs-Server entsprechend dem Reaktor-Muster auf Basis von NIO.