Election: Implementierung von Übung 3 in Node (JavaScript)

Felix Blechschmitt

Dieses Dokument enthält Informationen über den Aufbau und die Funktionen von election.

Getting Started

Das Projekt wurde in der Programmiersprache JavaScript nach dem Standard EcmaScript 6 (ES6) entwickelt. Dabei wurde der Interpreter node in der Version v4.4.4 verwendet.

Installation

Zunächst müssen alle notwendigen Abhängigkeiten installiert werden:

```
$ npm install
```

Usage

Zum Übersetzen der Anwendung sowie zum Starten der einzelnen Applikationen werden NPM-Skripte verwendet, die in der Datei package.json definiert werden.

Ein solches Skript wird über folgendes Kommando ausgeführt, dabei können optional Parameter übergeben werden:

```
$ npm run <SkriptName> [-- <Parameter>]
```

Ein Netzwerkknoten kann als Node.js-Skript als einzelner Prozess über ein NPM-Skript gestartet werden. Zum Starten einer Testsuite, die anhand verschiedener Parameter eine Netzwerktopologie erstellt und automatisiert alle notwendigen Netzwerkknoten startet, dient ein weiteres Node.js-Skript.

Build-Prozess

Node ist nicht in der Lage JavaScript-Dateien, die im ES6-Standard entwickelt wurden, direkt auszuführen. Diese müssen zunächst in den ES5-Standard übersetzt (transpiliert) werden. Hierzu wird babel verwendet.

Der Übersetzungsvorgang kann über das NPM-Skript "build" angestoßen werden. Dabei wird zunächst das möglicherweise bereits vorhandene Verzeichnis "build" geleert bzw. ein solches Verzeichnis erstellt. Anschließend werden alle JavaScript-Dateien von babel in den Standard ES5 übersetzt und inklusive Source Map im Verzeichnis "build" abgelegt.

```
$ npm run build
```

Starten eines Netzwerkknotens

Die Hauptanwendung des Projekts ist ein Netzwerkknoten, der einen Kandidaten bzw. einen Wähler in einem Netzwerk an Wähler- und Kandidatenknoten repräsentiert. Das Ziel der Kandidaten ist es, möglichst viele Wähler von sich zu überzeugen. Sie machen dies, indem sie über den Echo-Algorithmus Kampagnen verbreiten oder über sog. Flooding "Wähl-Mich"-Nachrichten verbreiten. Solche Nachrichten werden werden

wie ein Gerücht im Netzwerk verbreitet, wobei ein Knoten die Nachricht nur dann weiterleitet, wenn er dem Kandidaten anhand des Confidence-Levels zustimmt.

Ein solcher Netzwerkknoten wird über das NPM-Skript "start-node" gestartet. Die Konfiguration eines Knotens kann als Parameter übergeben werden. Werden keine Parameter angegeben, so werden die notwendigen Parameter abgefragt und für den Rest Default-Werte verwendet:

```
$ npm run start-noe
```

Alternativ können die notwendigen Parameter direkt beim Start übergeben werden, sodass die Anwendung ohne weitere Interkation mit dem Nutzer ausgeführt werden kann. Der Parameter "-h" listet alle möglichen Parameter auf:

```
$ npm run start-node -- -h
Usage: node run-node.js

      --endpointFilename=[ARG]  path to the endpoints file, leave blank to map ids to local ports
  -g, --graphFilename=[ARG]     path to the graph file defining the network node topology
      --id=[ARG]                ID of this endpoint
  -r, --receive=[ARG]           number of receives until a candidate starts a new call (e.g. campaign)
      --observer                start observer node
  -h, --help                    Display this help
```

So kann zum Beispiel ein Netzwerkknoten mit der ID 5 wie folgt gestartet werden:

```
npm run start-node -- -g ./config/graphElection.dot --id 5
```

Dieser Knoten ist dann auf localhost:4005 erreichbar. Der Knoten geht von einer Netzwerktopologie wie in der graphviz-Datei config/graph.dot definiert aus. Bei diesem Knoten handelt es sich um einen Wählerknoten, da die ID weder 1 noch 2 ist (was den beiden IDs für die Kandidatenknoten entspricht). Wird keine endpoint-Datei angegeben (wie in diesem Beispiel), so werden die Endpunkte als Lokal angenommen und der Netzwerkknoten mit der kleinsten ID erhält den Port 4001. Für jeden weiteren Knoten wird die Portnummer entsprechend hochgezählt. Die minimale Portnummer (default: 4000) kann über die Konstante MIN_PORT im EndpointManager angepasst werden. Die ID 0 ist für den Beobachter-Prozess reserviert und wird automatisch dem Port 4000 zugewiesen, was ebenso im EndpointManager definiert ist.

Starten des Init-Tools

Analog zur ersten Übung wird auch in diesem Projekt das Init-Tool verwendet, um Kontrollanchrichten an die Prozesse zu senden. Diesmal wird – abgesehen vom Kommando "stop all" zum Beenden aller Prozesse – hauptsächlich das Kommando "msg" verwendet, welches es erlaubt eine Nachricht des gewünschten Typs an einen oder mehrere Knoten zu senden.

Folgender Aufruf sendet beispielsweise eine INIT-Nachricht an beide Kandidaten, was den Wahlprozess startet und dafür sorgt, dass die Kandidaten damit beginnen, Kampagnen bzw. "Wähl-Mich"-Nachrichten zu verbreiten:

```
$ npm run init -- -c msg -t init --addresses "localhost:4001;localhost:4002" -m "empty"
```

Dabei ist es wichtig, dass ein Inhalt für die Nachricht gesetzt wird. Der Inhalt selbst ist bei einer INIT-Nachricht jeoch irrelevant. Andernfalls wechselt das INIT-Skript in den interaktiven Modus und fordert den Nutzer auf, einen Inhalt einzugeben.

Generieren eines Graphen

Zum Generieren der Netzwerktopologie wurde das Tool graphgen, welches über das Skript "graphgen" ausgeführt werden kann, erweitert. Dabei wird zunächst mithilfe des Havel-Hakimi-Algorithmus ein Graph generiert, welcher anschließend analog zu dem Verfahren in Übung 1 in eine graphviz-Datei gespeichert wird.

Auch hier können die benötigten Informationen direkt als Parameter übergeben. Eine Hilfe wird auch hier mit dem Parameter "-h" angefordert.

```
$ npm run graphgen -- -h
Usage: node graphgen.js

  -n                      Number of nodes
  -s, --supporter=[ARG]   Number of supporters
  -f, --friends=[ARG]     Number of friends
  -o, --out=[ARG]         Output filename
  -h, --help              Display this help
```

Über den Parameter "-o" bzw. "–out" (output) wird der Pfad angegeben, an dem der erzeugte Graph gespeichert werden soll.

Der Havel-Hakimi-Algorithmus dient ursprünglich dazu, anhand einer gegebenen Sequenz an Knotengraden zu überprüfen, ob es möglich ist, einen Graphen zu generieren, der genau die in der Sequenz angegebenen Knotengrade besitzt. Eine mögliche Sequenz wäre beispielsweise [4, 3, 3, 3, 1]. Sie gibt an, dass der Graph fünf Knoten besitzt, wobei ein Knoten vier kanten hat, drei Knoten jeweils drei Kanten und ein Knoten lediglich eine Kante besitzt.

In dieser Anwendung wird der Algorithmus dazu verwendet, um die Eingabeparameter zu verifizieren, um so sicher zu sein, dass ein Graph erzeugt werden kann, bei dem jeder Wähler f Nachbarknoten besitzt. Zusätzlich wurde der Algorithmus dahingehend erweitert, dass sich die im Ablauf verbundenen Kanten gemerkt werden, sodass nach einem Durchlauf des Algorithmuses der Graph erzeugt wurde.

Auf diese Weise wird das Wählernetz erzeugt, sodass jeder Wähler exakt f Freunde hat. Um sicherzugehen, dass der resultierende Graph zusammenhängend ist, wird der erzeugte Graph anschließend mit der Node-Bibliothek "connected-components" überprüft. Stellt sich heraus, dass es sich um keinen Zusammenhängenden Graphen handelt, wird die Eingabe verworfen und der Benutzer dazu aufgefordert andere Eingabeparameter zu verwenden.

Nachdem das Wählernetz erzeugt wurde, werden die beiden Kandidatenknoten hinzugefügt und diesen ihre Parteifreunde zugewiesen. Dazu wird ein zufälliger Knoten ausgewählt, dessen ID größer als 0 ist und kleiner als die maximale Anzahl an Knoten abzüglich der Anzahl an Parteifreunde *s*. Dieser Knoten sowie die *2* s* Folgeknoten werden alternierend den beiden Kandidaten als Parteifreunde zugewiesen. Die Parteifreunde haben somit in Summe einen Nachbarn mehr als normale Wählerknoten.

Ein Nachteil dieser Lösung ist jedoch, dass bei gleichen Eingabeparameter annähernd gleiche Graphen erzeugt werden.

Starten der Testsuite

Um eine Wahl durchführen zu können, ist es notwendig, dass alle Wählerknoten sowie die Kandidatenknoten und ein Observerprozess gestartet werden. Hierfür kann die Node-Anwendung "run-election.js" verwendet werden, welche über das NPM-Skript "start-election" ausgeführt wird. Als Parameter können u.A. die Anzahl an Knoten, die Anzahl an Freunde und Parteifreunde der Wählerknoten übergeben werden. Der Parameter "-h" zeigt eine Übersicht der möglichen Argument:

```
$  npm run start-election -- -h
Usage: node run-election.js

  -n, --nodes=[ARG]          number of nodes (without the observer node)
  -s, --supporters=[ARG]     number of supporters per candidate
  -f, --friends=[ARG]        number of friends for each voter
  -r, --receives=[ARG]       number of receives until a candidate starts a new call (e.g. campaign)
  -g, --graphFilename=[ARG]  path to the graph file defining the network node topology
```

```
-d, --delay=[ARG]          delay between initiating the election process and taking the snapshot (in s
-h, --help                 Display this help
```

Hinweis: werden keine Parameter angegeben, so werden die in "run-election.js" definierten Default-Werte verwendet.

Die Testsuite startet zunächst das graphgen-Tool, um ein Wählernetz passend zur gegeben Konfiguration zu erzeugen. Anschließend werden die einzelnen Knoten als eigenständige Kindprozesse gestartet. Dabei wird die Ausgabe jedes einzelnen Prozesses überwacht und auf die Standardausgabe des Hauptprozesses umgelegt. Sobald durch das Überwachen der Ausgabe erkannt wird, dass alle Knoten bereit sind, wird der Beobachter-Prozess gestartet und danach mithilfe des INIT-Skripts eine INIT-Nachricht an die beiden Kandidaten gesendet. Nach einer konfigurierbaren Verzögerung wird der Beobachter-Prozess angewiesen einen konsistenten Schnappschuss durchzuführen. Sobald ein Wahlergebnis verfügbar ist, werden alle Prozesse beendet und der Vorgang ist abgeschlossen.

Bei einem Durchlauf werden sehr viele Nachrichten zwischen den einzelnen Knoten ausgetauscht. Da jeder Knoten alle ein- sowie ausgehenden Nachrichten auf der Ausgabe protokolliert, wird die Ausgabe der Testsuite schnell sehr unübersichtlich. Aus diesem Grund wird empfohlen, die Ausgabe in eine Datei umzulenken, bzw. zum Beispiel mithilfe des *tee*-Kommandos die Ausgabe sowohl anzuzeigen und gleichzeitig in eine Datei zu schreiben. Bei der Ausgabe einer Log-Nachricht schreibt jeder Knoten seine eigene ID in runde Klammern. Dies kann man ausnutzen, um nur die Nachrichten eines bestimmten Knotens auszugeben. Folgendes Listing zeigt, wie die Testsuite gestartet wird und anschließend lediglich die Ausgabe des Beobachter-Prozess angezeigt wird, wobei sich die gesamte Ausgabe in der Datei log/out.log befindet:

```
$ npm run start-election -- -n 8 -s 2 -f 3 -r 3 -g ./config/graphElection.dot > ./log/out.log 2> ./log/
$ tail -f ./log/out.log | grep "(0)"
```

Aufbau

Das Projekt wurde nach einem stark modularisierten Konzept entwickelt. Im Folgenden werden die Verzeichnisstruktur, die einzelnen Komponenten sowie das verwendete Nachrichtenprotokoll beschrieben.

Verzeichnisstruktur

Die Source-Dateien befinden sich alle im Verzeichnis "src" im Hauptordner des Projekts. Prinzipiell kann eine gesamte Node-Anwendung in einer einzelnen JavaScript-Datei entwickelt werden. Dies ist jedoch aufgrund fehlender Übersicht nicht empfehlenswert. Daher wurde das Projekt in verschiedene Komponenten unterteilt, welche in einzelnen Dateien entwickelt wurden. Ähnliche Module wurden dabei in Unterverzeichnissen gruppiert:

- /lib: enthält wiederverwendbare Module, die anwendungsübergreifend verwendet werden können
- /parser: enthält Komponenten zum Einlesen von Dateien bestimmter Formate
- /tools: enthält zusätzliche Skripte, die als eigenständige Node-Anwendungen gestartet werden können

Die Datei "run-node.js" ist der Einstiegspunkt der Anwendung. Darin werden die einzelnen Komponenten miteinander verknüpft und der Netzwerkknoten gestartet.

Komponenten

Das folgende UML-Klassendiagramm zeigt einen Überblick über die einzelnen Komponenten der Anwendung.

Die Klasse Server ist eine einfache Implementierung eines TCP-Servers, der bei eingehenden Nachrichten eine vorher definierte Funktion aufruft. Ein Aufruf der Methode "listen(callback)" startet den Server und registriert die übergebene Funktion als Callback, welches bei eingehenden Nachrichten aufgerufen wird. Die Methode "stop()" beendet den Server.

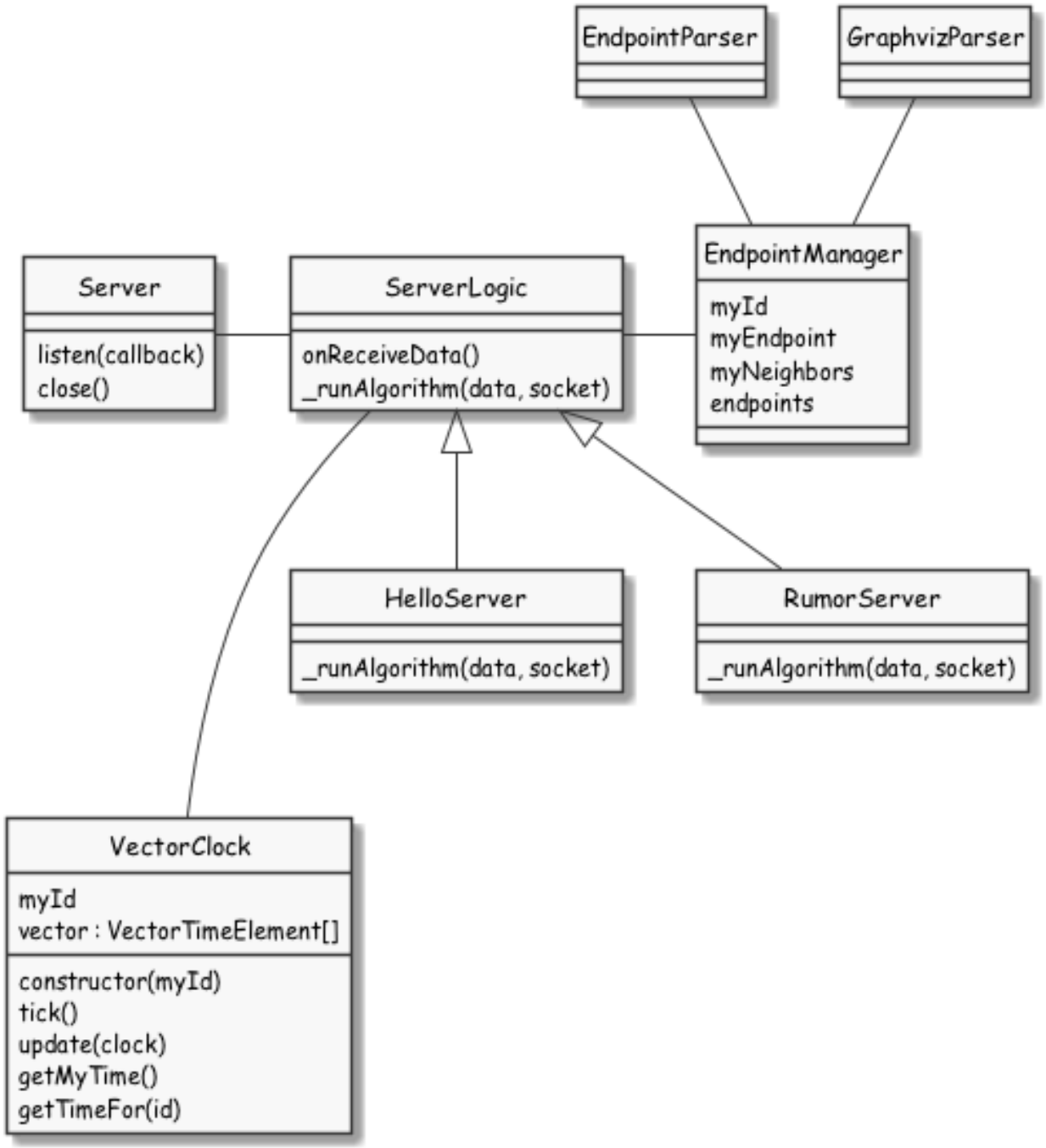

Figure 1: Komponenten der Anwendung

In der Basisklasse “ServerLogic” wird das Behandeln eingehender Nachrichten definiert. Sie kennt die verbundenen Endpunkte – also die Nachbarknoten – und besitzt eine Referenz auf den gestarteten Server. Verbunden werden beide Kompomenten, indem dem Konstruktor der ServerLogic die Referenz auf den Server übergeben wird. Anschließend wird die Methode “onReceiveData” der ServerLogic der “listen”-Methode des Servers als Callback übergeben. Auf diese Weise wird die Methode “onReceiveDate” bei jeder eingehenden Nachricht ausgeführt.

Handelt es sich bei einer eingehenden Nachricht um eine Kontrollnachricht, so wird diese von der ServerLogic direkt behandelt. Alternativ wird die *abstrakte* Methode “_runAlgorithm" aufgerufen, welche von der konkreten Implementierung “HelloServer” (bei Aufgabenteil a) bzw. “RumorServer” (späterer Aufgabenteil) überschrieben wird.

> Hinweis: In JavaScript gibt es eigentlich keine abstrakten Klassen bzw. Methoden. Daher wird die Methode “_runAlgorithm" in der Basisklasse einfach mit einem leeren Rumpf definiert.
>
> Speziallfall - Knoten als ein Prozess: In der Aufgabenstellung war gefordert, dass der Knoten als ein Prozess und somit die Bearbeitung einer eingehenden Nachricht als eine atomare Aktion ausgeführt werden soll. Um dies in node zu realisieren, wurde eine Semaphore verwendet, die alle anderen eingehenden Nachrichten blockiert, während eine Nachricht behandelt wird.

Vektorzeit

Damit in der Ausgabe ersichtlich ist, in welcher Reihenfolge die – teilweise parallel abgesetzen – Nachrichten gesendet wurden, wird eine logische Zeit eingeführt: Die Vektorzeit. Diese Zeiteinheit besteht aus n-Zählern, wobei n gerade der Anzahl an Netzwerkknoten entspricht. Die Vektorzeit ist Teil einer jeden Nachricht, die versendet wird. Der eigene Zähler wird genau dann erhöht, wenn ein lokales Ereignis passiert (beispielsweise das Senden oder Empfangen einer Nachricht). Beim Empfangen einer Nachricht werden außerdem die Werte aller Zähler verglichen und das Maximum übernommen.

Implementiert wurde die Vektorzeit als eigenes Modul. Die Klasse VectorClock beinhaltet ein assoziatives Array, welches die Zähler für verschiedene Netzwerkknoten, die anhand einer eindeutigen ID identifiziert werden, speichert. Tritt ein lokales Ereignis auf, so wird der eigene Zähler mittels der Methode “tick(()” erhöht. Beim Empfangen einer Nachricht wird der darin übermittelte Vektor an die Methode “update(clock)” übergeben, was dafür sorgt, dass das Maximum der jeweiligen Zähler übernommen wird. Im initialen Zustand kennt das Modul lediglich den eigenen Zählerstand. Das Vektorfeld wird mit jeder ankommenden Nachricht von einem vorher noch unbekannten Netzwerkknoten entsprechend angepasst und erweitert.

Protokoll

Zum Austausch der Nachrichten zwischen den einzelnen Netzwerkknoten wird das JSON-Format verwendet. Eine Nachricht hat dabei folgende Komponenten:

- type: Typ der Nachricht (rumor oder control)
- msg: Inhalt der Nachricht
- from: ID des Absenderknotens (optional)
- time: Vektorzeit des Absenderknotens (optional)

Die Vektorzeit ist optional, damit bei dem initialen Gerücht keine Zeit mitgesendet werden muss. Alle zwischen den einzelnen Rumor-Knoten ausgetauschten Nachrichten beinhalten natürlich das Feld “time”.

Handelt es sich um eine Kontrollnachricht, so enthält das Messagefeld die Aktion (“STOP” zum Beenden des Knotens bzw. “STOP ALL” zum Senden der “STOP ALL” Nachricht an alle Nachbarn und zum anschließenden Beenden des Knotens).

Alle Nachrichten werden über TCP übertragen. Auch wenn dies – aufrund der TCP-Übertragung – nicht explizit notwendig wäre, werden alle empfangenen Nachrichten nach dem Erhalt mit einem leeren JSON-Objekt

bestätigt.

Beispiel einer Nachricht

```
{"msg":"g.2","from":22,"type":"rumor"}
```

Beispiel einer Kontrollnachricht

```
{"msg":"STOP","type":"control"}
```

Tests

Die dynamische Typisierung von JavaScript birgt die Gefahr von häufigen Laufzeitfehlern. Aus diesem Grund ist es gerade bei der Verwendung einer solchen Programmiersprache besonders wichtig, die Komponenten der Anwendung mit automatisierten Tests auf korrekte Funktionalität zu überprüfen. Außerdem helfen gerade Unit-Tests bei der Entwicklung von einzelnen Komponenten, da man sich durch diese von der korrekten Funktionalität der Komponente überzeugen kann, da diese dann gezielt ohne den Kontext der gesamten Anwendung ausgeführt werden kann.

Automatisierte Tests

Einige Hilfsfunktionen sowie kleinere Module der Anwendung, wie zum Beispiel der Parser für die Endpoint- bzw. graphviz-Datei oder der Kantengenerator, wurden mit Unit-Tests versehen. Die Tests befinden sich im Verzeichnis “tests” des Projektordners.

Die Tests können mithilfe des NPM-Skripts “test” ausgeführt werden:

```
$ npm run test
```

Test Coverage

Continuous Integration & Code Quality

Im Rahmen dieses Projekts wird TravisCI als Continuous Integration Lösung verwendet. Dieses System ist mit dem github repository verknüpft und sorgt bei jeder Änderung der Daten im Repository dafür, dass die Anwendung erstellt wird und alle automatisierten Tests ausgeführt werden. Die Datei .travis.yml beschreibt die Konfiguration des Testservers und besagt welcher Interpreter in welcher Version verwendet werden soll.