Professur Rechnernetze
Institut Systemarchitektur
Fakultät Informatik
TU Dresden

Optimierung und Simulation des ZEEBus Smartgrid-Modells

Autor:
Dominik Pataky
<dpa@netdecorator.org>

Zeitraum:
Oktober bis
Dezember 2014

Zusammenfassung

Dieses Paper ist ein kurzer Bericht über meinen Arbeit im Rahmen einer SHK-Stelle des Projekts ZEEBus der Professur Rechnernetze, TU Dresden.

Während dieser Zeit habe ich u.a. ein unverwaltetes Repository in ein Gradleprojekt umgebaut, die Hardwarelandschaft mit Hilfe von Vagrant (und zeitweise Docker) virtualisiert und die Frontend-Androidapplication in HTML5 (in Kombination mit der Strophe.js-Bibliothek und der Web Worker-Technologie) überführt.

Meine Arbeit basiert auf dem Ergebnis des Komplexpraktikum *Internet of Things*, einer Demonstrationsmodelllandschaft zum Thema Smart Grid.

Inhaltsverzeichnis

1 Stand des Modells bei Übergabe

Bei Übergabe bestand das Projekt/die Simulation aus dem Modell mit Raspberry Pis, der Javasoftware für die Simulation und weiterer Zubehörsoftware, welche u.a. die Java-Jar vom Steuer-Pi aus auf den anderen Pis verteilt.

Zu diesem Zeitpunkt waren noch einige Features und Bugs offen. Darunter zählten bspw.

- schnellerer Start der Simulation bei Hochfahren des Modells,
- Reparatur einer korrupten SD-Karte, welche den Start der Holzfabrik verhinderte,
- die Organisation des Codes und seiner Abhängigkeiten von *mavenless* in Maven bzw. Gradle zu überführen,
- die App, welche die Übersicht über die Verteilung und den Stromverbrauch im Netz darstellt, responviser zu gestalten und
- herauszufinden, warum in manchen Situationen statt auf Nachbarstrom auf den Strom aus der externen Quelle zugegriffen wird.

2 Verbesserungen

2.1 Startbeschleunigung

Bei der ersten Inspektion des Startskripts wurde schnell klar, dass **Ansible** (das Tool, welches mit Hilfe eines playbooks mehrere Aufgaben beim Hochfahren des Modells ablaufend anstößt) optimiert werden konnte. Ansible wartete bspw. beim ersten Start auf alle anderen Pis, ohne dass ein Timeout dieser Wartezeit festgelegt wurde. Im Falle eines Ausfalles - wie der Holzfabrik - wartete der Control-Pi also unnötig lange, mehrere Minuten. Das Problem löste ein festgelegtes Timeout von 5 Sekunden.

Des Weiteren überspielte Ansible bei jedem Start alle nötigen Dateien des Simulationstools auf alle anderen Pis - auch, wenn sie nicht verändert wurden. Da jede Übertragung über SSH auf den Pis schon eine gewisse Zeit dauert (bzw. der Verbindungsaufbau), habe ich als ersten Schritt alle Dateien, die auf alle Pis verteilt wurde, in ein tarball gepackt und damit nur eine einzige Datei übertragen müssen. Das Verringern der SSH-Verschlüsselung auf den Pis hat kaum Geschwindigkeit gebracht. Gegen Ende meiner Arbeitszeit habe ich die Übertragung der Dateien ganz ausgeschaltet. Erst nur zu Testzwecken, aber letztlich wird in naher Zukunft wohl kein Update mehr gefahren werden.

Ein weiterer Trick war das Einschalten von parallelen Jobs. Eigentlich sollte Ansible in der default Konfiguration automatisch seine Tasks parallel auf allen Hosts anstoßen. Im Debug-Test war dies jedoch nicht der Fall, weshalb ich diesen Parameter ebenfalls in die Konfiguration einfügte.

Fazit: der Startvorgang wurde von ca. 12 Minuten auf 2-3 reduziert. Sicher geht es auch noch schneller, es ist aber auch gut möglich, dass die Pis in dieser Routine an ihre Grenzen stoßen (SSH, Java).

2.2 Korruption

Das Modell besaß bei Übergabe keine Routine, welche die Pis ordnungsgemäß herunterfährt. Beim einfachen Abschalten des Modells wird den Pis somit der Strom entzogen und die SD-Karten in einem korrupten Zustand hinterlassen.

Die Karte ließ sich glücklicherweise mit *fsck -vc /dev/sdb2* (filesystem check mit bad blocks) reparieren.

Außerdem befindet sich auf dem Steuer-Pi nun ein *shutdown*-Playbook für Ansible. Dies ist jedoch bisher nur über eine angeschlossene Tastatur bedienbar. Zum Ausführen reicht folgender Befehl auf dem Steuer-Pi: *cd /home/pi/ansible; ansible-playbook -i inventory.ini –sudo shutdown.yml*. Nach Ausführung des Playbooks sollte dann noch der Steuer-Pi runtergefahren werden. Dafür reicht ein simples *sudo shutdown -h now*.

2.3 Mavenless wird Gradle

Die Strukturierung des Codearchivs und die Abhängigkeitsverwaltung waren zur Zeit der Abgabe im sog. *mavenless*-Ordner zu finden. Anscheinend wurde im Laufe des Projekts ein Ansatz mit **Maven** verworfen und die benötigten Bibliotheken manuell aus einem extra dafür angelegten Ordner hinzugeladen.

Meine ursprüngliche Aufgabe sollte erstmal sein, das Ganze wieder in Maven zu überführen. In Besprechung dieses Punkts kam aber auf, dass man im gleichen Zuge **Gradle** auszuprobieren, welches als Feature teilweise die Funktionen von Maven ausleiht.

Das Projektrepository ist folgend ein Gradleprojekt geworden. Gradle funktioniert mit sog. *Tasks*, die im Prinzip jede Routine automatisieren können. Für unseren Anwendungsfall mit Java gibt es ein dediziertes Plugin, welches Tasks wie *build* und *javadoc* mitbringt.

Hinzugefügt habe ich dann die Abhängigkeiten des Projekts, Maven-Repositories für Pakete aus inoffiziellen Repos und den Task *bundle*. Der bundle-Task erstellt – im Gegensatz zum build-Task – eine Jar aus allen kompilierten Klassen, also auch der Abhängigkeiten. Zu finden ist der Code dazu in der *build.gradle*. Es gibt noch Bibliotheken, die aus einem Unterordner geladen werden. Diese sind entweder in gar keinem Maven-Repository oder eine Betaversion, die noch nicht im Maven-Repository verfügbar ist.

3 Virtualisierung der Hardwarelandschaft

Da es für die Weiterentwicklung der Software, welche die Raspberry Pis steuert und das Smart Grid simuliert, nötig war, Patches direkt auf dem Modell auszurollen und erst so testen zu können kam die Idee der Virtualisierung recht schnell auf.

Es gibt dafür viele Softwareprodukte, u.a. Virtualbox, KVM, VMware und weitere. Mittlerweile ist aber die Technik der Containerisierung, bei der nicht mehr die Hardware virtualisiert, sondern nur noch der RAM (und ggf. Netzwerk, Massenspeicher) des Gastsystem isoliert wird, weiter fortgeschritten. Dies hat den Vorteil, dass die Ressourcen des Hostsystems nicht mehr zugewiesen und reserviert werden müssen. Dadurch steigt die Performance erheblich.

3.1 Erste Ideen

Da mir daran lag, die Simulation der Raspberry Pis so mobil und flexibel wie möglich zu gestalten, probierte ich eine Kombination aus VM und Containern aus.

Praktisch war mein erster Ansatz die Erstellung einer virtuellen Maschine mit Virtualbox und einem Image von **CoreOS** als Gast. CoreOS ist ein für Docker bereitgestelltes Minimalbetriebssystem, welches auf die Erstellung und Verwaltung von Dockercontainern zugeschnitten wurde.

Die VM sollte also mit CoreOS als Gast Dockercontainer verwalten. In diesen Containern startet Docker dann sog. Images. Diese widerum sind ähnlich wie Snapshots, es gibt ein Basisimage (z.B. Debian Wheezy) und auf diesem basieren kann über eine Routine in einem Dockerfile[1] ein neues Kind-Image gebaut werden, das neue Daten und Software enthält.

In diesem Falle haben wir also Software auf den Raspberry Pis, laden von diesen die benötigten Komponenten (z.B. die KP IoT-Javaapplication) in die VM und erstellen ein Image *Debian Wheezy mit KP IoI-Javaapplication*. Starten wir dann einen Container mit diesem Image, kann z.B. über ein Skript die Application gestartet werden. Diese läuft dann im OS innerhalb des Containers.

3.2 Anschluss von Ansible

Beim Einsatz von CoreOS stellte sich jedoch heraus, dass es nutzerfreundlicher wäre, diese Maschine von außen zu steuern. Dafür nahm ich das im Projekt bereits eingesetzte **Ansible** in Anspruch. Ansible verbindet sich über SSH mit dem Host, läd seine Ablaufskripte nach und führt diese dann lokal aus.

Dafür jedoch benötigt Ansible einen Python-Interpreter auf dem Zielhost. Dieser ist in CoreOS nicht mitgeliefert. Es gibt dafür einen Workaround[2], der den pypy-Interpreter in CoreOS läd. Diese Lösung fühlt sich aber kaputt an, weshalb ich sie verwarf.

[1] `https://docs.docker.com/reference/builder/`

[2] `https://coreos.com/blog/managing-coreos-with-ansible/`

3.3 Wechsel von CoreOS zu Debian Jessie

Um Docker zu benutzen muss man schließlich nicht unbedingt CoreOS einsetzen. Es gibt z.B. im Debian testing release (jessie) mittlerweile das Docker.io-Paket, welches die Dockerengine installiert. Auf diesem Wege steht ein vollständiges Betriebssystem zur Verfügung, somit auch ein Pythoninterpreter für Ansible.

Der Jessie-Host braucht also ein gewisses Setup, um Docker einzusetzen und unsere Hardwarelandschaft 1:1 abzubilden. Hier kommt nun **Vagrant** ins Spiel.

3.4 Vagrant als Boxmanager

Vagrant ist ein Tool, mit dessen Hilfe man virtuelle Maschinen bzw. Container auf unterschiedlichen Virtualisierungs-/Containerisierungsplattformen (in Vagrant sog. **Provider**) teilen kann. Dies geschieht mit einem sog. **Vagrantfile**. In diesem steht, welche Basebox (z.B. eine Debianbox aus dem offiziellen Repository[3]) aufgesetzt werden soll und welche Schritte dann noch zusätzlich ausgeführt werden sollen. Ein kurzer Vergleich am Beispiel einer Virtualbox:

Fall 1: reine Virtualboxmaschine mit Export

1. Erstellen der VM mit Netzwerk, Datenträger etc.
2. Installation des Systems in dieser VM
3. Weitere Konfiguration des Systems innerhalb der VM über SSH
4. Export der Maschine an Zeitpunkt X als Exportimage
5. Verteilen dieses Exportimages
6. Andere Entwickler importieren nun dieses Image und haben die gleiche Box

[3] `https://vagrantcloud.com/`

Fall 2: Vagrant mit Virtualbox als Provider

1. Erstellen des Vagrantfile mit Konfigurationen zur Basebox
   (a) Welche Basebox soll bezogen werden?
   (b) Welche Netzwerkschnittstellen und welche IP soll die Maschine haben?
   (c) Sollen Ordner des Hostsystems im Gast eingebunden werden?
2. Ggf. Erstellen weiterer Setupskripte, die beim Erstellen der Virtualbox-Maschine ausgeführt werden. Z.B.:
   (a) 'Installiere die Java-VM und Ansible'
   (b) 'Füge den Benutzer pi hinzu und lege Dateien X in dessen Home'
3. Verteilen dieser Datei
4. Andere Entwickler können nun anhand des Vagrantfile die Box selber aufsetzen und wie eine normale VM benutzen. Es ist außerdem möglich, die Maschine einfach zu löschen (*destroy*) und wieder von neu aufzusetzen.

Der Vorteil von Vagrant liegt also besonders darin, dass nur der Vagrantfile an andere weitergegeben werden muss. Selbst, wenn man ein eigenes Image als Baseimage[4] einsetzen will, brauch man dieses nicht an andere weiterzuverteilen, sondern kann das Image von einem zentralen Server beziehen. Des Weiteren ist das Setup der Box sehr viel übersichtlicher, in diesem Falle setze ich bspw. neben dem Vagrantfile nur ein weiteres Shellskript ein. Dieses installiert Pakete, Nutzer und kopiert Daten in die Maschine, ohne das der Entwickler selbst die VM konfigurieren muss.

3.5 Docker läuft, das Netz nicht

Zu diesem Zeitpunkt hatte ich also eine Vagrant-Virtualbox-VM, in welcher Docker.io installiert wurde und noch ein Baseimage erstellt werden musste. Dies tat ich testweise mit Hilfe eines Docker Debian-Baseimage[5]. Es ist möglich, in Docker direkt über z.B. *bash* in einem Container Kommandos auszuführen. Auf diesem Wege testete ich zwei Container, einen als Control-Pi, den anderen als einen der Clients.

Ich musste schnell feststellen, dass die Kommunikation der Container untereinander mit der Standardeinstellung der Javaapplication kollidierte. Docker erstellt beim Start automatisch ein eigenes Subnetz (ein 172.16/12) mit einer eigenen *docker0*-Bridge, an die die Container angeschlossen sind. Das Subnetz lässt sich ggf. ändern, es gibt aber keinen ersichtlichen Weg, die IP eines Containers beim Start festzulegen. Deshalb schei-

[4] `https://docs.vagrantup.com/v2/boxes/base.html`
[5] `https://registry.hub.docker.com/_/debian/`

terte der Versuch, das Subnetz aus der Application zu benutzen.

Einen Lösungsansatz überlegte ich mir unter Zuhilfenahme von *iptables*. Es ist möglich, über entsprechende Source-NAT (SNAT) und Destination-NAT (DNAT) die Quell- bzw. Ziel-Adresse eines IP-Pakets an der *docker0*-Bridge zu ändern. Man könnte also bei Paketen, die die Bridge betreten und eine 192.168.178/24-Adresse suchen, die Zieladresse zur Containeradresse des jeweiligen Empfänger-Clients ändern. Auf diesem Wege finden sich Clients trotz unterschiedlicher Subnetze. Sollten sie jedenfalls, funktionierte aber praktisch nur in eine Richtung, es kam kein Paket zurück.

3.6 Weiter ohne Docker

Um weitere unberechenbare Experimente zu verhindern, entschied ich mich an diesem Punkt gegen die weitere Verwendung von Docker. Docker wäre in diesem Falle nur zuverlässig einsetzbar gewesen, wenn man die Kommunikation zwischen den Javaapplication-Clients der Umgebung angepasst hätte. Dies hatte ich jedoch von vornherein ausgeschlossen.

Folgend erstellte ich das im Repository zu findende Vagrantfile. Es baut auf Basis eines Debian anhand eines Skripts bis zu sieben Virtualbox-VMs, eine für den Control-Pi und sechs weitere für die Häuser. Jede Maschine ist mit ihren eigenen Dateien bestückt, d.h. die Clients verhalten sich genau so, wie die Pis auf dem Hardwaremodell. Auch das Subnetz lässt sich so emulieren, es Bedarf keiner Anpassung der Software.

Sieben VMs aufzusetzen ist hier übrigens gar nicht nötig, denn zum Testen neuer Versionen der Application reicht es, den Control-Pi und einen Client zu erstellen, da sich alle anderen Clients genau wie der erstellte verhalten würden (einziger Unterschied ist die Konfiguration der im Client enthaltenen Geräte).

3.7 Fazit

Für die Simulation der Hardware steht also ein Vagrantfile mit mehreren VMs im Repository zur Verfügung. Was hier noch fehlt, ist der Input, der in der Hardwarelandschaft mit Hilfe der Steuerboards an jedem Haus ausgelöst wird. Man muss zum Testen der Abläufe innerhalb der VM-Umgebung selbst dem Jabber-Kanal beitreten und die regelmäßigen Inputs des Control-Pi eingeben.

4 Überarbeitung des Frontend

4.1 Bestandsaufnahme der Visualization-App

Beim Inspizieren der Visualisierungs-Application, welche auf einem Cubieboard mit Android läuft, stellte ich fest, dass diese bloß WebViewer benutzt, um eine HTML-Seite darzustellen. Ich nehme an, dass sich hier bei der Entwicklung gegen die ursprüngliche Implementierung einer Python-flask-App entschieden wurde, weil in der Android-Variante der Code für den XMPP-Bus wiederverwendet werden konnte. Dieser nutzt die Smack-Bibliothek, der Cubie verbindet sich also als eigenständiger Benutzer mit dem XMPP-Server auf dem Control-Pi und betritt die für ihn angelegten Kanäle. In diesen werden vom Control-Client JSON-Daten ausgegeben, welche die App verarbeitet und mit Javascript und der Highcharts-Bibliothek darstellt.

Da für die weitere Entwicklung des Frontends eine Einarbeitung in Android-APIs u.a. Frameworks nötig ist, aber letztendlich nur eine HTML-Seite dargestellt wird, entschied ich mich hier, den Overhead der App aus der Software zu entfernen. Das Frontend sollte zukünftig nur noch eine HTML-Seite sein, die der Cubie über den nachinstallierten Chrome-Browser aufrufen kann. Zu erreichen ist die Seite über den Webserver des Control-Pi, *http://192.168.178.203*.

Wie aber sollte eine statische HTML-Seite sich mit dem XMPP-Bus verbinden? Hierfür implementierte ich einen clientseitigen XMPP-Client mit **Strophe.js**.

4.2 Auslagerung der Arbeit in Web Worker

Strophe.js läuft bei normaler Benutzung (per Einbindung über ein <*script*>-Tag) im normalen Browsercontext parallel zu allen anderen Skripten. Das bedeutet, dass sich alle Skripte der Seite einen Thread teilen und sich gegenseitig beeinflussen. Das behindert in unserem Falle z.B. die flüssige Ausführung des Sliders, der über Touchgesten nach links und rechts die Inhalte wechselt.

Um dieses Problem, das bereits in der App besteht, zu umgehen, habe ich die Skripte von Strophe.js und weitere Teile des Datenaufbereitungs-Skripts ausgelagert. Die hier verwendete Technologie ist das **Web Worker**-Feature, welches von allen Browsern unterstützt wird[6]. Web Worker sind eigenständige Threads, in denen der Browser Javascript-Code ausführt. Der In- und Output dieser Threads geschieht über *Worker.postMessage()* und *Worker.onMessage()*. Man kann also – wie auch in meiner Implementierung zu finden – einen XMPP-Thread starten, der im Hintergrund die Verbindung zum XMPP-Server verwaltet, und nur bei einer eintreffenden Nachricht diese an den Haupt-Thread schicken. Dieser kann damit dann weiter arbeiten und Daten verarbeiten.

Ein großes Problem gibt es dabei jedoch: Web Worker sind aus Sicherheits- und Performancegründen in ihrer Ausstattung sehr eingeschränkt. D.h. konkret, dass Web Worker nicht auf das DOM (den HTML-XML-Baum) des Dokuments zugreifen können

[6] `http://caniuse.com/#feat=webworkers`

und auch keine eigenen DOM anlegen dürfen. Strophe.js jedoch benutzt für die Verarbeitung von Stanzas (XMPP-XML-Paketen) Funktionen eines DOM[7], wie dem Iterieren über einer Node und deren Children.

4.3 Strophe.js als Web Worker

Strophe.js wurde nach Auslagerung in einen Web Worker zwar ausgeführt, konnte aber keine Nachrichten verarbeiten und somit nicht zu einem Server verbinden, geschweige denn Nachrichten empfangen und versenden. Da ich aber die Idee, einen völlig unabhängigen Thread für XMPP starten zu können weiter verfolgen wollte, suchte ich nach Lösungen, die Einschränkungen von Web Workern zu umgehen. Wenn man schon beliebige Skripte ausführen kann, gibt es doch sicher auch eine Möglichkeit, ein DOM zu emulieren?

Tatsächlich gibt es dafür eine passende Bibliothek: **xmljs**[8]! xmljs emuliert ein W3C standardisiertes DOM in Javascript. Leider waren auch hier noch ein paar Anpassungen nötig, da xmljs teilweise beim Erstellen von Nodes die zugehörigen Children nicht direkt über die Node zugänglich macht, sondern diese erst über *Node.childNodes._nodes* zu finden sind. Ich habe hier die nötigen Zeilen in Strophe.js eingefügt.

Und siehe da – es funktioniert! In der Implementierung zum Stand Ende Dezember 2014 gibt es also ein Haupt-Skript, dass die Graphen u.a. verwaltet, ein Web Worker für die Aufbereitung der Chartdaten und einen Worker für XMPP.

Tests beim Ausrollen auf der Hardware ergaben zum Abschluss jedoch ein sehr ernüchterndes Ergebnis. Das XMPP-Skript verliert bei Ausführung auf dem Cubie wohl Pakete, was dazu führt, dass der XMPP-Client sich nicht mit dem XMPP-Server verständigen kann. Ist hier die Hardware zu schwach? Das verbaute Cubieboard2 sollte mit seinem Dualcore eigentlich genug Power haben, um die Visualisierung auszuführen.

[7] Genauer gesagt, Strophe.js verwendet XMLHTTPRequests für BOSH. Diese jedoch liefern keine von Strophe.js benötigte *xmlresponse*, da der notwendige Parser nicht zur Verfügung steht

[8] `http://xmljs.sourceforge.net/`

5 Stand Ende 2014

Folgende Punkte sind für die weitere Entwicklung des Modells interessant:

- Der Start der Simulation kann u.U. noch weiter verbessert werden. Es kann aber auch sein, dass die Optimierung des Playbooks bereits das Ziel erreicht hat und eine weitere Verbesserung nur mit besserer Hardware zu erreichen ist.
- Die Visualisierungs-App ist teilweise noch mit altem Code aus der Android-App bestückt. Neu gebaute und wiederverwertete Teile habe ich dokumentiert und geordnet, aber gerade in der *index.html* und der *template.html* sind noch Codestücke, die sehr unsauber aussehen. Des Weiteren könnte man die Daten der App so zusammenbauen, dass nur eine einzige Config ausreicht, um die Ausstattung der Häuser, die Datenarrays der Graphen und die Konfiguration des Templates zu steuern. Auch wäre es dann evtl. von Vorteil, das *nunjucks*-Templaterendering rauszuwerfen.
- Da noch eine Steuerung der Pis in der Simulation fehlt, könnte man diese ebenfalls in XMPP implementieren und in die Visualisierungsoberfläche einbauen.
- Sollte die Implementierung mit Web Workern weiterhin nicht funktionieren, könnte man an den Control-Pi einen WLAN-Adapter hängen, mit dem sich externe Geräte verbinden, über DHCP eine Adresse beziehen und dann auf *http://192.168.178.203* die Oberfläche selbst anzeigen können. Das sollte das Performanceproblem lösen (da die Seite auf meinem Notebook bspw. ohne Einschränkung funktioniert).
- Vagrant erstellt zu diesem Zeitpunkt bis zu sieben virtuelle Maschinen. Vagrant hat jedoch auch eine Schnittstelle, mit der man Docker als Provider verwenden und konfigurieren kann. Das könnte man zusätzlich in das Vagrantfile einbauen. Problem hierbei: die Dockerengine ist noch nicht für jede Plattform als fertiges Package verfügbar, daher ist ein Fallback auf Virtualbox sinnvoll.