Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projektbericht

Diamond und OpenTSDB

vorgelegt von

Benjamin Brahmer, Jan Synwoldt

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6684762, 6708629

Betreuer:	Michael Kuhn, Hendryk Bockelmann, Julian Kunkel, Eugen Betke

Hamburg, 17.03.2017

Inhaltsverzeichnis

1 Einleitung

1.1 Themeneinführung

In diesem Bericht präsentieren wir die Ergebnisse unserer Analyse von Diamond und OpenTSDB. Diamond, OpenTSDB und Grafana bilden ein System zur Überwachung von Rechnern. Solch ein Monitoring-System kann genutzt werden, um potentielle Probleme von Anwendungsprogrammen aufzuzeigen. Ein gutes Beispiel hierfür wäre ein „Memory Leak“, bei dem immer mehr Speicher vom Anwendungsprogrammen reserviert wird, ohne den nicht mehr verwendeten Speicher frei zu geben. Dies würde im Monitoring System durch einen stetig anwachsenden Speicherverbrauch auffallen.
Dabei wurde der TSDB Handler, welcher die Verbindung zwischen Diamond und OpenTSDB darstellt, analysiert und angepasst. Die vorgenommenen Anpassungen wurden auf ihre Eignung in Hinsicht auf ihre Leistung geprüft.

1.2 Aufgabenstellung

Zu Beginn des Projekts sollte die Eignung von Diamond in Kombination mit der OpenTSDB geprüft werden. Während einer ersten Analyse wurde festgestellt, dass der TSDB Handler, welcher die Verbindung zwischen Diamond und OpenTSDB darstellt, nicht Optimiert war. Dabei viel insbesondere auf, dass jede erfasste Metrik einzeln gesendet wurde, was zu einer hohen Last auf der CPU führte. Die Last, welche Diamond verursacht sollte Minimiert werden, da möglichst viel Leistung für Anwendungsprogramme zur Verfügung gestellt werden soll.
Im Anwendungsfall des Deutschen Klima Rechenzentrums (DKRZ) werden über 3000 Rechner (Knoten) für Klimasimulationen bereitgestellt. Klimaforscher nutzen diese Knoten, um unterschiedliche Simulationen zu entwickeln und zu testen sowie die Ergebnisse dessen auszuwerten. Um den Nutzen zu Maximieren, darf Systemsoftware die einzelnen Knoten nicht zu stark belasten. Diamond würde auf jedem Knoten laufen und dort Metriken sammeln. Je größer die Last ist, welche Diamond erzeugt, desto länger brauchen die Klimasimulationen. Daraus Entwickelte sich die Aufgabe, den TSDB Handler zu überprüfen und zu Optimieren.

Vorgehensweise

Zunächst haben wir ein Testsystem in Kapitel 2 aufgesetzt, mit dem wir unsere Änderungen am TSDB Handler Analysieren konnten.
Im Kapitel 4 stellen wir unsere Änderungen und die Ergebnisse unserer Analysen vor. Jede Optimierung wurde dabei in Vorstellung, Umsetzung und Analyse aufgeteilt. Zunächst erklären wir, welche Änderung wir planen und welches Ergebnis wir erwarten; dabei wurde nicht jedes mal unsere Erwartung erfüllt. In der Umsetzung zeigen wir, wie wir unsere Planung umsetzen konnten dies wird teilweise durch Programmabschnitte ergänzt. In der Analyse prüfen wir wie sich die Änderungen auswirken.

1.3 Diamond

Diamond ist ein in Python geschriebener, quelloffener Monitoring Deamon. Dieser kann zur Erfassung von Metriken auf einem Rechner eingesetzt werden. Diamond ist dabei modular aufgebaut und kann leicht um eigene Kollektoren zum Sammeln von Metriken sowie eigene Handler zur Verarbeitung erweitert werden.

1.4 OpenTSDB

OpenTSDB ist eine **t**ime **s**eries **d**ata**b**ase, kurz TSDB. Mit ihr können große Datenmengen in Form einer Zeitreihe gespeichert werden. Die OpenTSDB ist speziell dafür entwickelt, viele zeitabhängige Daten von verschiedenen Quellen zu erhalten.
OpenTSDB fungiert als Front-End für das darunterliegende HBase Cluster. Wird mehr Leistung benötigt, so können zusätzliche Rechner zum HBase Cluster hinzugefügt werden. Dies ermöglicht lineare Skalierung in Abhängigkeit zu den verwendeten Rechnern. OpenTSDB implementiert ein rudimentäres Grafisches Interface. Bietet jedoch eine API um andere Anwendungen anzubinden.

1.5 Grafana

Grafana ist eine Webanwendung mit der Zeitreihen Grafisch dargestellt werden. Die Oberfläche ist dabei modern und Benutzerfreundlich. Eine Anbindung an OpenTSDB ist sehr einfach zu realisieren.

2 Einrichten eines Testsystems

Unser Testsystem besteht aus zwei Rechnern. Auf dem ersten Rechner installieren wir OpenTSDB, HBase und Grafana, dies bildet unser Datenbank-Cluster ab. Auf dem zweiten Rechner installieren wir Diamond; dieser soll seine Metriken später an das Datenbank-Cluster senden.

2.1 OpenTSDB einrichten

OpenTSDB baut auf HBase auf, daher muss zunächst HBase eingerichtet werden.

2.1.1 HBase

In dieser Anleitung wird eine lokale Installation von HBase eingerichtet, diese reicht zum Testen aus. Für eine reale Installation benötigt man eine HBase Installation auf dem Hadoop Distributed File System (HDFS). Als Basis dient eine Ubuntu Server Installation Version 16.04.2. Eine Raute am Beginn eines Befehls bedeutet, dass dieser Befehl mit root-Rechten ausgeführt werden muss während Befehle mit einem Dollar Zeichen auch mit Standard-Rechten ausgeführt werden können.

Als erstes wird Java installiert.

```
# apt-get install java-1.8.0-openjdk
  ↪ java-1.8.0-openjdk-devel
```

Nach der Installation von Java laden wir HBase direkt von einem der Spiegelserver herunter.

```
$ wget 'http://ftp.fau.de/apache/hbase/stable/
hbase-1.2.4-bin.tar.gz'
```

Prüfsumme vergleichen.

```
$ sha256sum hbase-1.2.4-bin.tar.gz
$ wget 'http://ftp.fau.de/apache/hbase/stable/
hbase-1.2.4-bin.tar.gz.mds'
```

Archiv entpacken.

```
$ tar xzf hbase-1.2.4-bin.tar.gz
$ cd hbase-1.2.4
```

Damit HBase funktioniert, muss noch eine Umgebungsvariable gesetzt werden. Dies kann leicht über ein Skript von HBase erledigt werden.

```
$ vi conf/hbase-env.sh
```

Mit einem Texteditor muss in der Datei der folgende Eintrag hinzugefügt werden.

```
export JAVA_HOME=/usr
```

Dies sollte auf den meisten Systemen funktionieren.

Für HBase sollte ein eigener Benutzer angelegt werden.

```
# adduser testhbase
```

Zum Abschluss muss noch eine Konfiguration angepasst werden.

```
$ vi conf/hbase-site.xml
```

```
<property>
    <name>hbase.rootdir</name>
    <value>file:///home/testhbase/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/home/testhbase/zookeeper</value>
</property>
```

Starten von HBase.

```
# ./bin/start-hbase.sh
```

Stoppen von HBase.

```
# ./bin/stop-hbase.sh
```

Nach dem Starten von HBase kann mit jps das erfolgreiche starten geprüft werden.

```
# jps
1719 Jps
1455 HMaster
```

Zusätzlich kann die Verbindung mit der Hilfe von Telnet geprüft werden.

```
# apt-get install telnet
$ telnet localhost 2181
```

In der Telnet Session dann.

```
status
```

Die Ausgabe sollte dann in etwa so aussehen.

```
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09
   ↪ GMT
Clients:
 /127.0.0.1:36504[0](queued=0,recved=1,sent=0)
 /127.0.0.1:36510[1](queued=0,recved=2,sent=2)
 /127.0.0.1:36502[1](queued=0,recved=6,sent=6)
 /127.0.0.1:36508[1](queued=0,recved=3,sent=3)
 /127.0.0.1:36498[1](queued=0,recved=67,sent=70)
 /127.0.0.1:36500[1](queued=0,recved=22,sent=24)
 /127.0.0.1:36506[1](queued=0,recved=4,sent=4)

Latency min/avg/max: 0/3/66
Received: 106
Sent: 110
Connections: 7
Outstanding: 0
Zxid: 0x2cc
Mode: standalone
Node count: 44
Connection closed by foreign host.
```

2.1.2 OpenTSDB

Für OpenTSDB können wir das entsprechende Paket direkt von GitHub herunterladen.

Vor der Installation muss noch gnuplot installiert werden, damit OpenTSDB über die Web-GUI Graphen aus den Daten erstellen kann.

```
# apt-get install gnuplot
```

```
$ wget 'https://github.com/OpenTSDB/opentsdb/releases/
download/v2.3.0/opentsdb-2.3.0_all.deb'
```

```
# dpkg -i opentsdb-2.3.0_all.deb
```

Zum Schluss müssen noch die Basis Tabellen für OpenTSDB eingerichtet werden.

```
# env COMPRESSION=NONE HBASE_HOME=/root/hbase-1.2.4
  ↪ /usr/share/opentsdb/tools/create_table.sh
```

OpenTSDB bietet eine eigene, einfache webbasierte GUI, welche wir über 127.0.0.1:4242 erreichen.

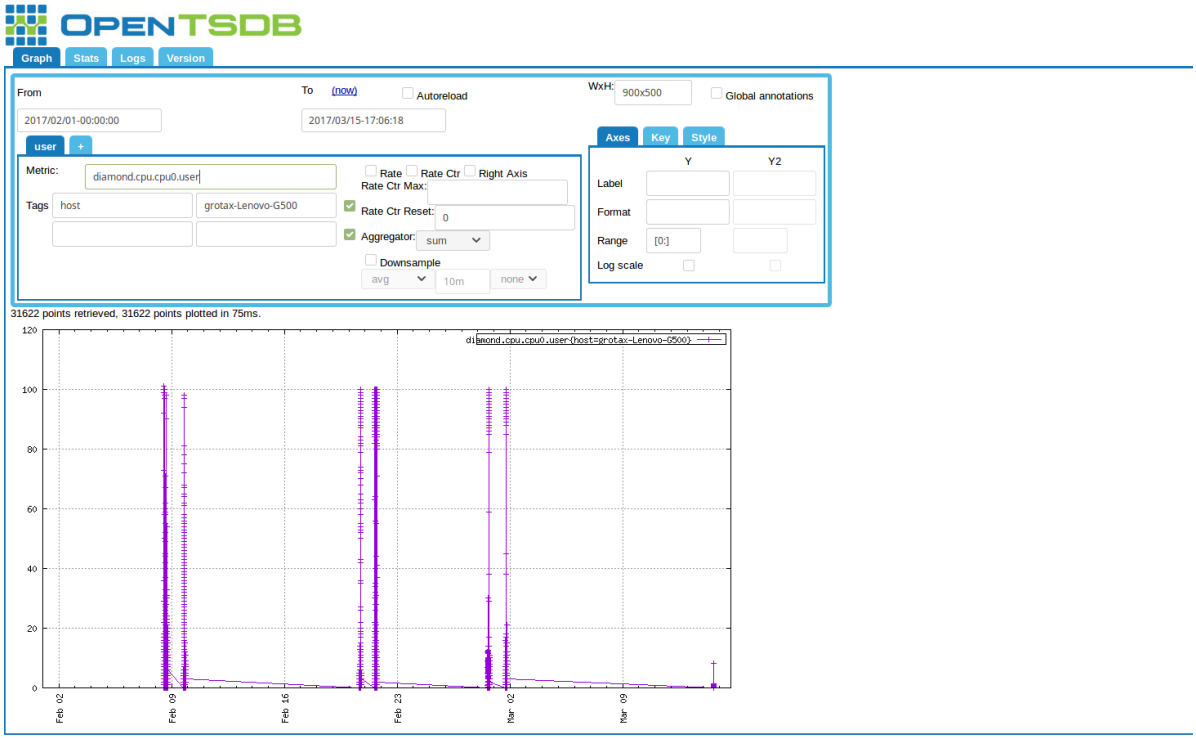


Abbildung 2.1: OpenTSDB Web GUI

2.2 Diamond

Diamond kann direkt aus dem Repository auf GitHub installiert werden.

Das Repository wird von GitHub geklont.

```
$ git clone https://github.com/python-diamond/Diamond
$ cd Diamond
```

Vor der Installation müssen noch einige Abhängigkeiten installiert werden.

```
# apt install make pbuilder python-mock python-configobj
  ↪ dh-python cdbs devscripts build-essential
```

Danach kann das Paket mit `make` erstellt werden.

```
$ make builddeb
```

Das fertige Paket muss nur noch installiert werden.

```
# dpkg -i build/diamond_X.X.X_all.deb
```

Für Diamond muss nun noch eine Konfiguration angelegt werden.

```
# cp /etc/diamond/diamond.conf.example
  ↪ /etc/diamond/diamond.conf
```

2.3 Grafana

Aktuelle Version von Grafana herunterladen.

```
$ wget 'https://grafanarel.s3.amazonaws.com/builds
/grafana_4.1.2-1486989747_amd64.deb'
```

Abhängigkeit installieren.

```
# apt install libfontconfig
```

Und Grafana selbst installieren.

```
# dpkg -i grafana_4.1.2-1486989747_amd64.deb
```

Start von Grafana

```
# service grafana start
```

Nach dem Start von Grafana kann man sich auf der Weboberfläche (`http://127.0.0.1:3000/`) mit dem Benutzernamen `admin` und dem Passwort `admin` authentifizieren. In der Abbildung 2.2 sieht man wie die TSDB an die Weboberfläche angebunden werden kann. In Dashboards, siehe Abbildung 2.3, können verscheide Möglichkeiten zur Visualisierung von Daten genutzt werden. Grafana bietet dann ein Interface um Anfragen an die TSDB zu formulieren (Abbildung 2.4). In Abbildung 2.5 wird ein Beispiel für ein fertiges Dashboard gezeigt.

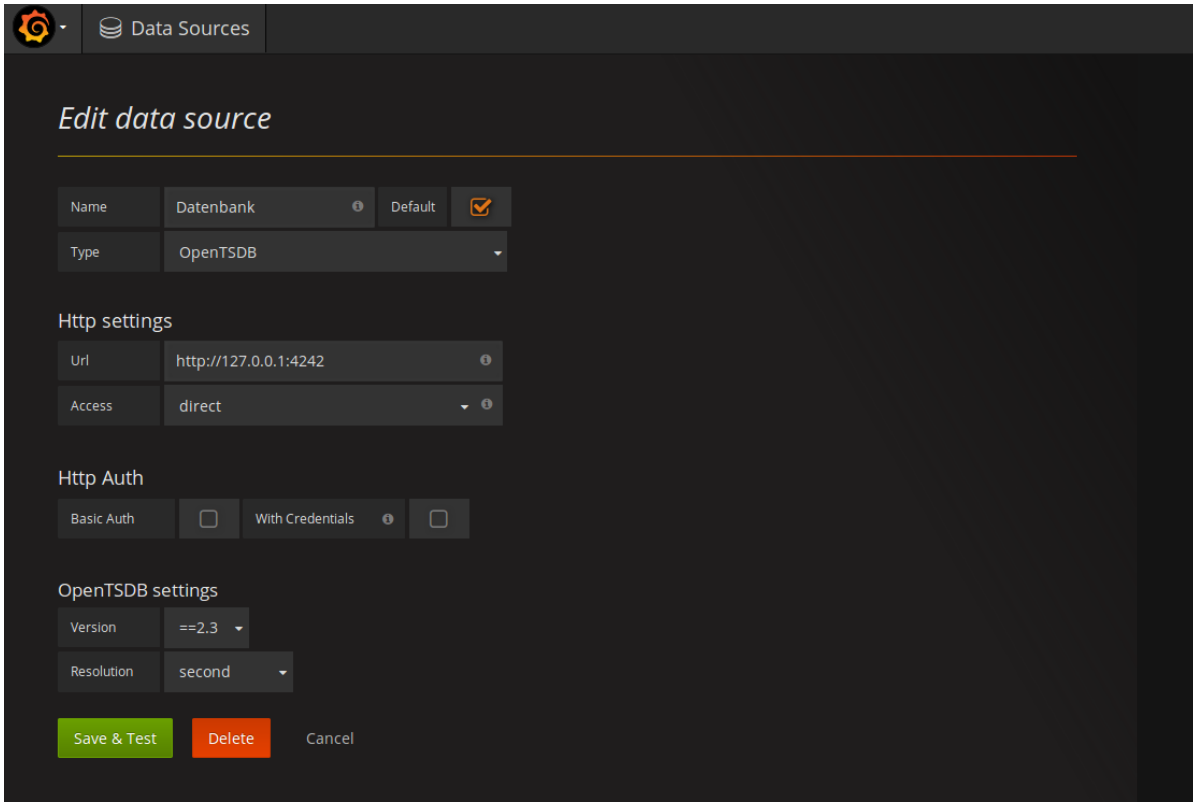


Abbildung 2.2: OpenTSDB in Grafana einbinden

Abbildung 2.3: Neues Dashboard

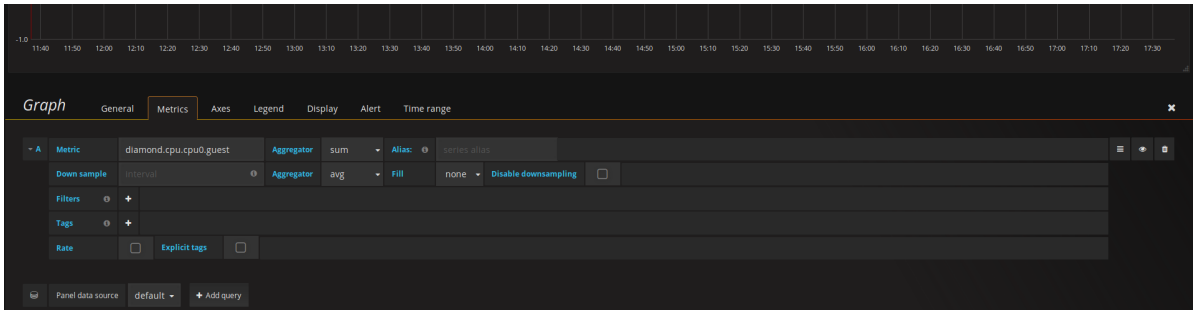


Abbildung 2.4: Neue Daten für Grafana

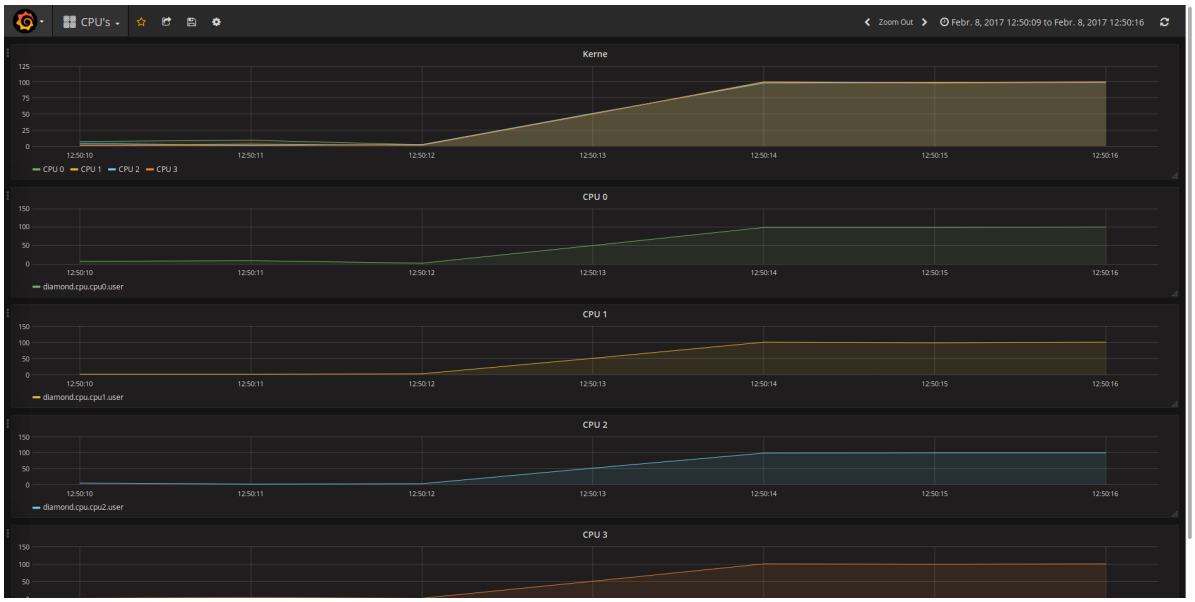


Abbildung 2.5: Grafana Beispiel

3 Erste Analyse

3.1 Diamond Aufbau

Diamond unterteilt sich im wesentlichen in drei Teile: die Kollektoren, eine interne Warteschlange und die Handler.

3.1.1 Kollektoren

Kollektoren sammeln Werte in einem definiertem Intervall auf dem Rechner ein. Beispielhaft sammelt der CPU Kollektor über `/proc/cpuinfo` Daten über die CPU Auslastung. Per Default sammeln die Kollektoren alle 5 Minuten neue Werte. Diese Metriken werden über eine Warteschlange an die Handler weitergegeben.

3.1.2 Warteschlange

Die Metriken aus der Warteschlange werden nacheinander an die Handler weitergegeben.

3.1.3 Handler

Handler verarbeiten die Metriken weiter, indem sie dauerhaft gespeichert werden. Mit einem Handler kann die Anbindung an eine Datenbank oder auch das Speichern in eine Datei realisiert werden.

3.2 OpenTSDB Handler

Der TSDB-Handler sendet die gesammelten Metriken an die OpenTSDB. Dabei wird jede Metrik einzeln versendet. Dies scheint nicht sehr effizient zu sein, da für jede neue Metrik auch eine neue Verbindung zur OpenTSDB aufgebaut werden muss.

4 Optimierungen

Für den Handler werden wir im folgenden verschiedene Anpassungen in Hinsicht auf ihre Leistungsverbesserung getestet. Nach jeder Anpassung prüfen wir, wie sich diese auf die Übertragungsdauer und die CPU auswirkt.

4.1 Methoden zur Leistungsanalyse

Für unsere Messungen haben wir zwei Rechner verwendet, welche über ein einfaches Layer-2 Switch mit einander Verbunden waren.

	Diamond Client	OpenTSDB Server
CPU	Intel Core i3-3110M 2 Kerne, 4 Threads	Intel Core i5-2500 4 Kerne, 4 Threads
RAM	8 GB	16 GB
Speicher	SSD 256 GB	SSD 256 GB
Netzwerk	100 MBit/s	1000 MBit/s

Tabelle 4.1: Spezifikationen der beiden Rechner.

Aus diesem Aufbau gingen die im Bericht aufgeführten Messungsergebnisse hervor.

4.1.1 Methode zur Messung der Übertragungsdauer

Die erste Methode prüft wie sich die Änderung auf die Übertragungszeit auswirkt. Dafür haben wir die Zeit gemessen welche zum Senden einer Metrik benötigt wird. Dazu gehören mehrere Vorgänge:

- Daten in JSON umformen
- Gegebenenfalls Daten komprimieren
- Senden der Daten an die DB
  - Zeitraum bis die DB die Daten geprüft und entsprechend geantwortet hat

4.1.2 Methode zur Messung der CPU-Last

Die zweite Methode prüft wie stark die CPU durch Diamond belastet wird. Für diese Messungen verwenden wir ein mit OpenMP parallelisiertes Programm zur Lösung von partiellen Differentialgleichung.

Skript zur Automatisierung der Messungen:

```
#!/bin/bash

for i in $(seq 1 10);
do
 echo $i;
 ./partdiff-openmp 4 2 350 2 2 250 | grep Berechnungszeit >>
    ↪ time.txt
done
```

Das Programm ermittelt selbst wie lange es zur Lösung der Gleichung benötigt. Diese Messung wiederholen wir zehn mal, um Unterschiede in der allgemeinen Systemlast auszugleichen. Aus den gemessenen Werten bilden wir einen Mittelwert, welchen wir dann mit anderen Ergebnissen vergleichen können. Die Parameter für das Programm müssen dabei an den verwendeten Rechner angepasst werden.

Grafik-Erläuterung

Die folgenden Darstellungen sind mit Ausnahme von Abbildung 4.4a Boxplots. für diese gilt: die Grenze zwischen hellrot und dunkelrot ist der Median der Daten, die Enden des Kastens jeweils unteres und oberes Quartil (25% und 75%), die Whiskerer haben eine Länge entsprechend dem eineinhalbfachen Interquartilabstand beziehungsweise bis zum äußersten Datenpunkt, falls dieser Abstand kürzer ist.
Die Größe „Zeit pro Eintrag“ ist die Division der für ein Paket benötigten Zeit durch die Paketgröße.

4.2 Metriken stapelweise senden

Metriken stapelweise zu versenden scheint effizienter als jede Metrik einzeln zu versenden. Diese Funktionalität wurde auch bereits von anderen Handlern implementiert. Die Annahme war, dass sich das Senden von mehreren Metriken pro Sendung an die OpenTSDB vorteilhaft auf die benötigte Zeit pro Metrik beim Senden auswirkt.

4.2.1 Implementierung

Um das Stapelweise senden zu ermöglichen wurde eine zusätzliche Option eingeführt, welche die Größe des Pakets bestimmt welches versendet wird. Per Default steht der Wert auf 1 und verhält sich somit wie der originale Code.

```
Batch = 1
```

Der Wert ist nach oben nicht begrenzt, jede Zahl kleiner als 1 führt zum sofortigem Versand. Um diese Funktion umzusetzen, wurde das verwendete Python Modul `socket` gegen `urllib2` getauscht. Dieses teilt unsere Anfrage bei Bedarf in mehrere passende Pakete auf - dies ist dann ein „chunked request“. Dies ergibt sich aus den Limitierungen der Nutzlast eines Ethernet-Frames. `urllib2` bestimmt dabei in Abhängigkeit zur OpenTSDB selbst die Größe der Pakete. Die HTTP-API von OpenTSDB benutzt die Standard HTTP Error Codes, so dass wir sehr einfach den Zustand der Anfrage überprüfen und etwaige Fehler abfangen können. Die Metriken werden als Dictionary angelegt und in einer Liste gespeichert, welche dann bei Überschreiten der Batch-Größe im JSON-Format an die OpenTSDB gesendet wird.

Ausschnitt aus dem Programm.

```
...
entry = {}
        entry['timestamp'] = metric.timestamp
        entry['value'] = metric.value
        entry['metric'] =
           ↪ (self.prefix+metric.getCollectorPath() +
                              '.'+metric.getMetricPath())
        entry["tags"] = {}
        entry["tags"]["hostname"] = metric.host
...
self.entrys.append(entry)

if (len(self.entrys) >= self.batch):
...
    data = json.dumps(self.entrys)
    self._send(data)
...
```

4.2.2 OpenTSDB

OpenTSDB unterstützt die chunked requests nicht per Default, bietet jedoch zwei Optionen um dies zu ändern.

Änderungen an der OpenTSDB.

```
tsd.http.request.enable_chunked = true
tsd.http.request.max_chunk = 16384
```

Wobei die zweite Option die maximale Wert in Bytes der Summe aller zur einer Anfrage gehörenden Pakete ist. Diese ist per Default auf 4096 Bytes festgelegt und muss für größere Anfragen nach oben angepasst werden. Hier kann auch die Kompression der Nutzdaten helfen, welche in Kapitel 4.5 näher betrachtet wird.

4.2.3 Auswertung

Übertragungsdauer

Durch die Bündelung der zu sendenden Datensätze ließ sich viel Zeit und CPU-Leistung einsparen. Je größer das Paket ist, das zusammengesetzt wird, desto weniger Zeit wird für jeden einzelnen Eintrag eingespart, siehe Abbildung 4.1. Es wird damit das Problem umgangen, dass bei jedem aufgenommenen Datenpunkt eine neue Verbindung vom Client zur Datenbank aufgebaut wird. In Abbildung 4.1 sind zwecks Übersichtlichkeit Paketgrößen 1 und 2 entfernt, da diese die Darstellung sehr stark in die Breite ziehen würden.

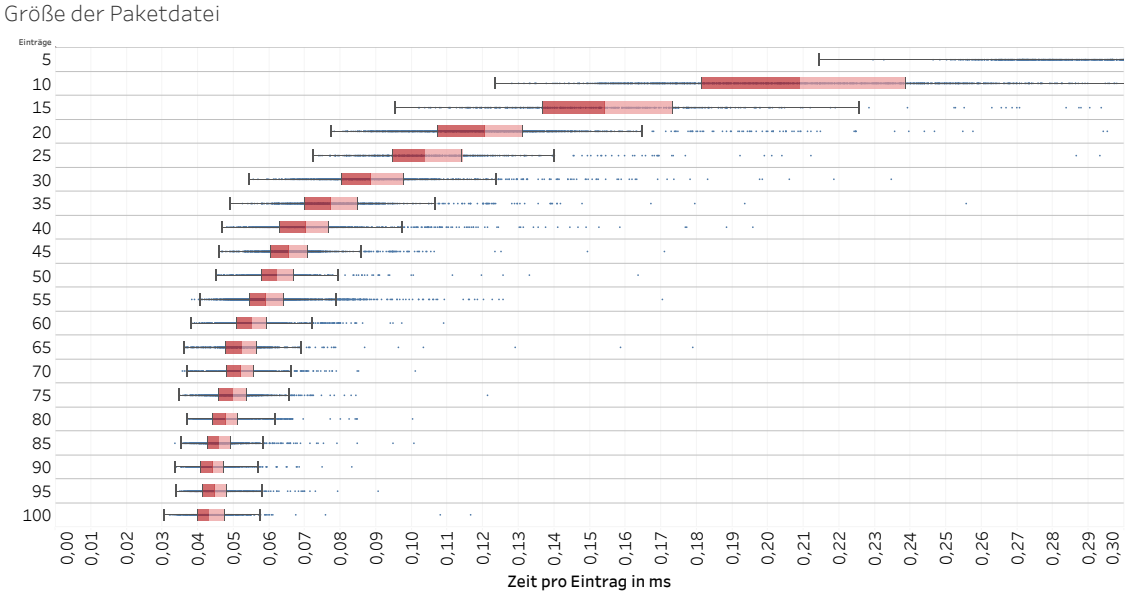


Abbildung 4.1: Vergleich der Sendedauer pro Eintrag bei verschiedenen Paketgrößen. Einträge über 0,3 Millisekunden nicht in der Grafik enthalten.

CPU-Last

4 Kollektoren jede Sekunde		
2 Kollektoren alle 5 Minuten		
Kein Diamond	Batch = 1	Batch = 100
68,396136	73,432572	68,722765
68,836141	73,596032	68,755164
67,706762	73,613584	68,859119
67,765697	73,562483	68,764157
67,591877	73,401159	69,456291
67,707352	73,692761	69,384195
67,714675	73,582522	68,769354
67,754873	73,349845	68,83361
67,831872	73,706098	70,184227
67,603385	73,385837	69,269608
Mittelwert		
67,890877	73,5322893	69,099849

Tabelle 4.2: Batch CPU-Last

Wie man an der Tabelle ablesen kann, führt das Nutzen von höheren Batch-Werten zu einer Verringerung der verursachten CPU-Last. Dies wird durch selteneres Senden der Daten verursacht. In diesem Versuch wurden vier Kollektoren jede Sekunde aufgerufen und zwei weitere alle fünf Minuten.
Jede Sekunde: CPUCollector, LoadAverageCollector, MemoryCollector, NetworkCollector.
Alle fünf Minuten: DiskSpaceCollector und DiskUsageCollector
Wiederholt man diese Messung und ruft dabei die Kollektoren seltener auf, so nähert man sich dem Wert ohne Diamond weiter an.

Batch = 100		
2 Kollektoren alle 5 Minuten		
Kein Diamond	4 Koll. alle 5 Sek.	4 Koll. alle 10 Sek.
68,396136	68,685426	67,718308
68,836141	68,00604	67,765941
67,706762	68,001878	67,779562
67,765697	68,365752	67,682232
67,591877	68,078658	67,682247
67,707352	67,854212	67,642552
67,714675	67,94821	67,899254
67,754873	67,600948	67,738901
67,831872	68,009863	70,341175
67,603385	68,685274	67,871268
Mittelwert		
67,890877	68,1236261	68,012144

Tabelle 4.3: Batch CPU-Last 2

Hier können wir ablesen wie stark sich die Veränderung des Kollektor Intervalls auf die CPU Last auswirkt. Werden die vier Kollektoren nur noch alle 10 Sekunden aufgerufen, erreichen wir fast die Leistung welche wir ohne Diamond erzielen konnten.

4.3 Verbindung offen halten

Das relativ häufige Öffnen und Schließen der TCP-Verbindung zur OpenTSDB sorgt für eine CPU-Last in Abhängigkeit zur Häufigkeit. Dies konnten wir bereits im Kapitel 4.2 feststellen. Es gibt die Möglichkeit, eine TCP-Verbindung aufrechtzuerhalten und damit eine Session zu erzeugen. Innerhalb einer Session müssen keine neuen TCP-Handshakes durchgeführt werden. Da Diamond regelmäßig sendet, müssen weiterhin keine Keepalive Pakete gesendet werden. Es bietet sich also an, diese Technik auf ihre Tauglichkeit zu prüfen.

4.3.1 Implementierung

Mit der Bibliothek `requests` ist es möglich, eine TCP-Verbindung offen zu halten. Dabei wird bei der ersten Verbindung ein TCP-Handshake durchgeführt und alle folgenden Sendungen können innerhalb dieser Session durchgeführt werden.

Hierfür muss in der `__init__()` Methode des Handlers eine Session angelegt werden.

```
...
self.connection = requests.Session()
...
```

Beim Senden wird dann die entsprechende Session verwendet.

```
...
r = self.connection.request(method='POST',
    url="http://"+self.host+":"+str(self.port)+"/api/put",
    headers=self.httpheader,
    data=content)
...
```

4.3.2 OpenTSDB

In der OpenTSDB Konfiguration muss für dieses Feature die folgende Option aktiviert sein.

```
tsd.network.keep_alive = true
```

4.3.3 Auswertung

Übertragungszeit

Wie Abbildung 4.2 zeigt konnte die Übertragungszeit mit diesem neuen Ansatz nicht verringert werden.

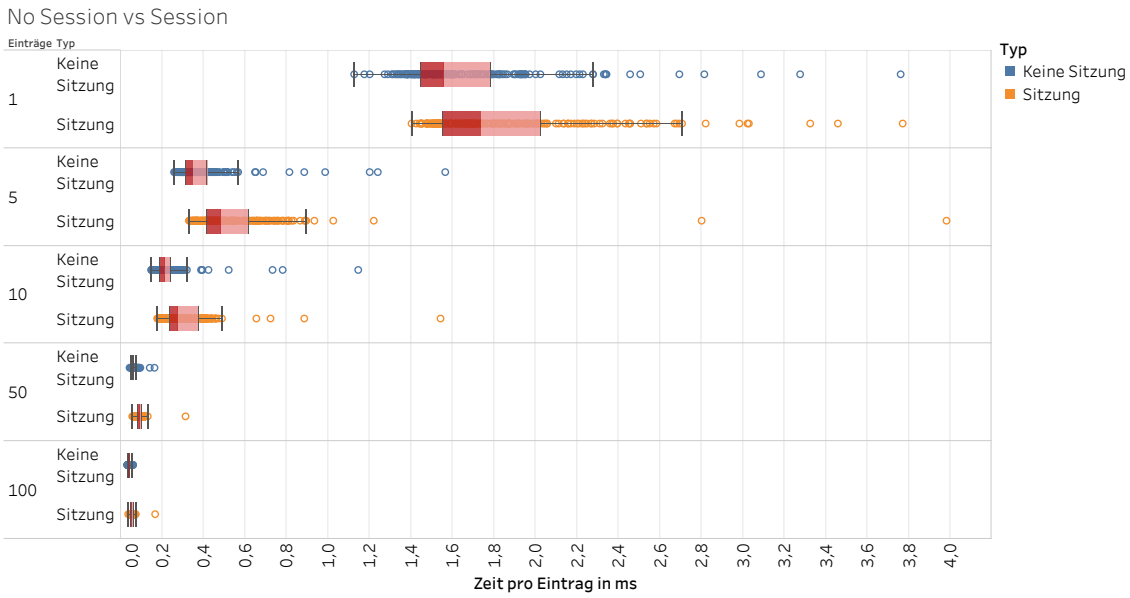


Abbildung 4.2: Zeitbedarf der Session Methode und dem alten Ansatz

CPU-Last

Auch für diese Änderung haben wir die CPU-Last gemessen. Wieder wurden die gleichen Intervalle für die Kollektoren verwendet, um eine Vergleichbarkeit zu ermöglichen.

Ohne Diamond	Batch = 1	Batch = 100
68,396136	77,037946	68,86047
68,836141	77,223595	68,793905
67,706762	78,435282	68,902232
67,765697	76,800976	69,133617
67,591877	77,896317	68,892275
67,707352	76,520224	69,218198
67,714675	76,52768	68,82607
67,754873	76,610482	68,995457
67,831872	77,422738	70,801031
67,603385	78,681146	68,924949
Mittelwert		
67,890877	77,3156386	69,1348204

Tabelle 4.4: CPU-Last bei erhalten der TCP Verbindung

In dieser Tabelle können wir erkennen, dass wir mit unserer neuen Methode die CPU stärker belasten insbesondere fällt auf, dass bei Batch = 1 die CPU-Last deutlich höher ist als bei dem vorhergegangen Test. Dieser Ansatz konnte sich also nicht durchsetzen.

Die Ursache für die schlechtere Leistung könnte der zusätzliche Aufwand sein, der durch die Verwaltung der Session erzeugt wird, die hohe CPU-Last würde dafür sprechen. Auch Versuche mit größeren Paketen und Beobachten des Netzwerkverkehrs brachten keine neuen Erkenntnisse. Probleme bei der verwendeten Umgebung können jedoch nicht völlig ausgeschlossen werden.

4.4 Vorregistrierung von Metriken

Jeder Metrik, jedem Tag und jedem Tag-Wert muss von der OpenTSDB eine eindeutige ID zugewiesen werden. Außerdem lädt die Datenbank diese Zuweisung in den Cache, um eine schnelle Verarbeitung zu ermöglichen. Die IDs können manuell per Kommandozeile vor dem Übertragen der Daten angelegt werden. Alternativ können sie auch automatisch von der OpenTSDB registriert werden. Dafür muss der folgende Eintrag in der Konfiguration eingetragen werden.

```
tsd.core.auto_create_metrics = true
```

Ist dies der Fall, so wird sobald eine neue Metrik mit Tags an die OpenTSDB gesendet wird, die entsprechenden IDs angelegt. Die Zuordnung von String zu ID wird dann in den Cache der Datenbank geladen und ist damit effizient nutzbar. Das Erzeugen der IDs kostet etwas Zeit und somit dauert das erste Eintragen der Metrik länger. Nach dem Neustart einer OpenTSDB ist der Cache zunächst leer und füllt sich erst mit der Zeit, sodass die ersten Sendungen etwas

mehr Zeit benötigen. Der Cache lässt sich auch direkt beim Start füllen. Dafür muss Folgendes in der Konfiguration eingetragen werden:

```
tsd.core.preload_uid_cache = true
tsd.core.preload_uid_cache.max_entries = 300000 #Default
```

Dies erfordert jedoch genügend Speicher und eine saubere Datenbasis, damit keine unnötigen Daten geladen werden. Eine weitere Möglichkeit für das Anlegen der IDs stellt die HTTP-API dar. Die Metriken können am API-Endpunkt `/api/uid/assign` registriert werden. Als Rückmeldung erhält man eine Antwort mit den Zuweisungen von Metrik zu ID sowie Tag zu ID und Tag-Wert zu ID.

Beispielhafte Anfrage:

```
{
    "metric": [
        "diamond.cpu.cpu0.idle",
    ],
    "tagk": [
        "host"
    ],
    "tagv": [
        "local",
    ]
}
```

Und die Entsprechende Antwort:

```
{
    "metric": {
        "diamond.cpu.cpu0.idle":"001337",
    },
    "tagv": {},
    "tagk_errors": {
        "host": "Name already exists with UID: 0004E6",
    },
    "tagk": {
        "local": "000042",
    }
}
```

Durch das Registrieren der Metriken könnte Diamond den Cache der DB füllen und somit beim Senden einen Geschwindigkeitsvorteil gewinnen.

4.4.1 Implementierung

Die Implementierung für diese Funktion ist dem normalen Senden der Metriken sehr ähnlich. Es wird ein Dictionary angelegt, welches den Namen der Metrik sowie die dazugehörigen Tags mit den entsprechenden Werten enthält.

```
...
if entry['metric'] not in self.registered:
    reg = {
        "metric": [
            entry['metric'] # Der Name der Metrik
            ],
            "tagk": [ # Die Tags
                "host",
            ],
            "tagv": [ # Die Werte fuer die Tags
                metric.host,
            ]
    }
    self._register(json.dumps(reg))
...
```

Über ein Dictonary können wir aufzeichnen, welche Daten wir bereits registriert haben.

4.4.2 OpenTSDB

Bei der OpenTSDB müssen keine Änderungen vorgenommen werden.

4.4.3 Auswertung

Übertragungszeit

In der folgenden Grafik haben wir drei Methoden mit einander verglichen:

- Vorregistrieren der Metriken
- kein Vorregistrieren der Metriken
- das vorherige Verfahren, ohne Registrierung

In Abbildung 4.3 sind diese Varianten bezüglich ihrer Dauer gegenübergestellt.

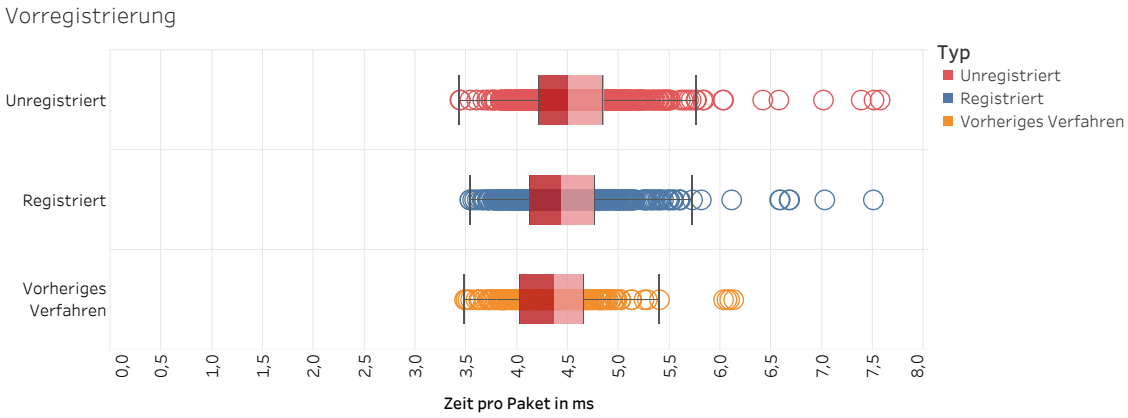


Abbildung 4.3: Zeit für Unregistriert, Registriert und das vorherige Verfahren

Ein Vorteil lässt sich in dieser Grafik nicht ausmachen. Der Unterschied zwischen den Varianten ist marginal, während der zusätzliche Code, welcher für diese Funktionalität benötigt wird, eher einen Nachteil darstellt.

CPU-Last

4 Kollektoren jede Sekunde		
2 Kollektoren alle 5 Minuten		
Ohne Diamond	Ohne Registrieren	Mit Registrieren
68,396136	69,744716	68,968032
68,836141	69,725074	68,975672
67,706762	71,648338	69,063688
67,765697	69,273633	69,123811
67,591877	69,025644	69,336269
67,707352	68,91375	69,713814
67,714675	69,318537	69,167713
67,754873	69,772654	69,074335
67,831872	69,926235	69,113223
Mittelwert		
67,922820	69,705397	69,170728

Tabelle 4.5: CPU-Last durch das Registrieren

Die Vorregistrierung der Metriken erbrachte leider nicht den erhofften Leistungsgewinn. Die OpenTSDB Dokumentation sagt dazu, dass sich das Registrieren von Metriken vor allem dann lohnt, wenn man in größeren Intervallen große Mengen von Daten sendet. Der Cache ist in diesem Moment entweder leer oder die Metriken sind neu und müssen erst registriert werden.

Dann lohnt sich das Registrieren der Metriken und Tags, da man dann in diesem Fall beim Senden der eigentlichen Daten Zeit spart. Dieser Anwendungsfall tritt jedoch in der Umgebung von Diamond voraussichtlich nie auf. Diamond ist dafür ausgelegt, in regelmäßigen, kleinen Abständen immer die gleichen Metriken zu senden. Daher wurde dieser Ansatz verworfen.

4.5 Komprimierung

Die Datenmengen, welche beim Versand der Metriken anfallen, sind für heutige Maßstäbe klein. Für eine einfache Anfrage mit nur einem Tag in etwa 20KB an Daten an. Dies ist natürlich stark von den Metriken sowie den Tags abhängig. Dennoch kann sich eine Komprimierung der Daten anbieten. Die OpenTSDB beschränkt die maximale Größe einer Anfrage siehe Kapitel 4.2. Dieses Limit wird angewendet, bevor die Datenbank die Anfrage dekomprimiert. Damit können effektiv mehr Daten in einer Transaktion übermittelt werden.

4.5.1 Implementierung

Die OpenTSDB-API unterstützt mit `gzip (GNU zip)` komprimierte Anfragen. Diese können wir auch mit Diamond leicht stellen.

Hierfür müssen wir einen neuen Header hinzufügen.

```
if self.compression >= 1:
    self.httpheader["Content-Encoding"] = "gzip"
```

Zusätzlich komprimieren wir die JSON Daten vor dem Senden.

```
if self.compression >= 1:
    data = StringIO.StringIO()
    with gzip.GzipFile(fileobj=data,
       ↪ compresslevel=self.compression, mode="w") as f:
            f.write(json.dumps(self.entrys))
            self._send(data.getvalue())
```

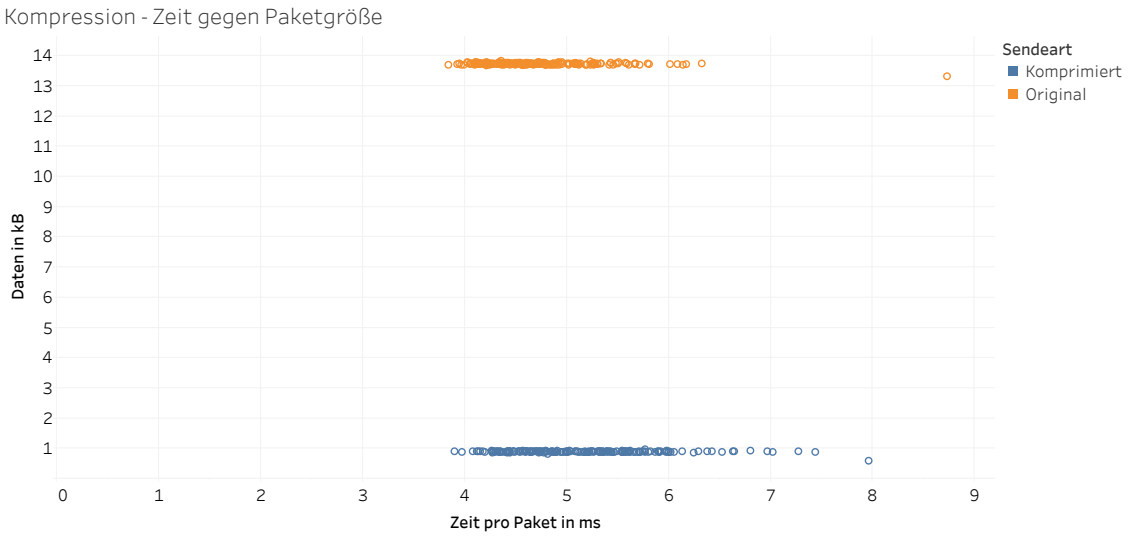
4.5.2 OpenTSDB

Die OpenTSDB hat bereits die Möglichkeit implementiert, gzip-verpackte Pakete anzunehmen. OpenTSDB erkennt automatisch ob es sich um komprimierte Daten handelt.

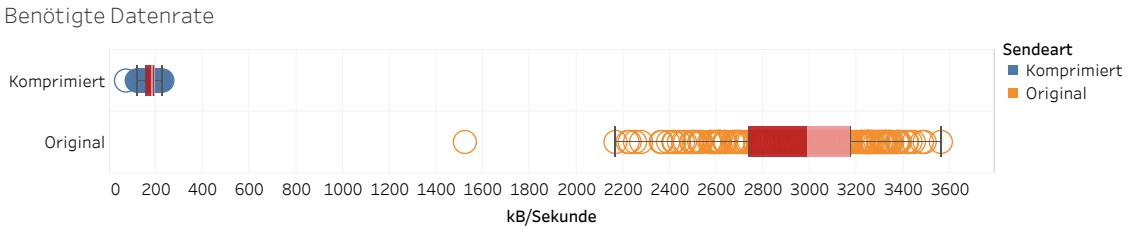
4.5.3 Auswertung

Übertragungszeit

Wie in Abbildung 4.4a zu sehen kostet die Komprimierung der Daten nur wenig Zeit, hier bei Paketgröße 100 etwa 0,5 Millisekunden. Dabei war das Kompressionslevel, welches bestimmt wie stark die Daten komprimiert werden sollen auf dem maximalen Level (9). Dafür sind die komprimierten Daten um das Vierzehnfache kleiner.



(a) Benötigte Zeit im Vergleich zur Datengröße mit Unveränderten und mit Kompressionsstufe 9 verkleinerten Daten



(b) Vergleich der benötigten Datenrate

Abbildung 4.4: Unterschied zwischen komprimierter und unkomprimierter Sendung

In der Folge wird die Netzwerkschnittstelle weniger ausgelastet. Dadurch wird insbesondere bei großen Systemen das Grundrauschen im Netzwerk minimiert und der Einfluss auf andere Kommunikation innerhalb des Netzwerks verringert.

CPU-Last

4 Kollektoren jede Sekunde		
2 Kollektoren alle 5 Minuten		
Ohne Diamond	Mit Kompression	Ohne Kompression
68,396136	69,443787	68,877779
68,836141	69,926206	69,010501
67,706762	68,948782	69,257462
67,765697	69,09808	69,246329
67,591877	69,169683	68,987489
67,707352	70,761737	68,698664
67,714675	69,556971	69,216947
67,754873	68,796754	69,122706
67,831872	69,53741	71,301987
67,603385	69,962206	69,083615
Mittelwert		
67,890877	69,5201616	69,2803479

Tabelle 4.6: CPU Last bei Komprimierung mit Level 9

Die CPU-Last ist trotz des stärksten Kompressionslevels nicht viel größer als die Verarbeitung ohne Kompression. Zusätzlich erhält man bereits bei einem Kompressionslevel von 6 eine ähnlich stark Komprimierte Datenmenge und spart dabei CPU-Zeit.

4.6 Kleinere Anpassungen

4.6.1 Authentifizierung

Eine Einfache Authentifizierung wurde durch das Hinzufügen eines Headers gelöst.

Authorization Header

```
...
if self.user != "":
    self.httpheader["Authorization"] = "Basic " +\
        base64.encodestring('%s:%s' % (self.user,
            ↪ self.password))[:-1]
...
```

4.6.2 Tags

Auch wurde die Verarbeitung der Tags, welche jeder Metrik hinzugefügt werden, angepasst. Diese können nun in der Konfiguration als Tupel aus Tag und Wert eingetragen werden.

Tag Konfiguration.

```
tags = datacenter=Hamburg site=DKRZ system=Mistral
```

Diese Tags werden mit einem Regulären Ausdruck gefunden und der Metrik hinzugefügt.

Tag Konfiguration

```
tags = "datacenter=Hamburg site=DKRZ system=Mistral"
```

Tags einlesen

```
...
self.tags = []
        pattern = re.compile(r'([a-zA-Z0-9]+)=([a-zA-Z0-9]+)')
        for (key, value) in re.findall(pattern,
           ↪ str(self.config['tags'])):
            self.tags.append([key, value])
...
for [key, value] in self.tags:
            entry["tags"][key] = value
...
```

4.6.3 Präfix

In der Konfiguration kann ein Präfix definiert werden, der vor jede Metrik gestellt werden soll.

```
prefix = "diamond"
```

5 Schlussfolgerung

Das Ergebnis dieses Projekts ist ein deutlich verbesserter TSDB Handler. Der Quellcode steht hier zur Verfügung:
`https://github.com/Grot4x/Diamond/tree/feature/tsdb_batch`
Ein Pull Request wurde bereits gestellt.

Eine weitere Verbesserungen wäre die Nutzung von mehr Tags zugunsten von weniger Metriken. Der Vorteil hierbei ist, dass der Metrikname sich verkürzt und das Filtern über die Tags erledigt wird. Beispielhaft würde dann aus

```
diamond.cpu.cpu0.guest_nice, Host = local
```

die folgende Metrik.

```
diamond.cpu.guest_nice, CPU = 0, Host = local
```

Hierdurch wird die Zahl der benötigten Metriken und IDs verringert. Dies kann bei großen Systemen durchaus relevant sein, da die Anzahl der IDs begrenzt ist. Hierfür existiert bereits eine Implementierung, welche aber nicht alle Kollektoren abdeckt.
Die Komprimierung könnte weiter verbessert werden, indem in die OpenTSDB und Diamond eine Unterstützung für einen schnelleren Komprimierungsalgorithmus wie LZ4 zum Übertragen der Anfragen implementiert wird.
Außerdem müssten die Kollektoren hinsichtlich ihrer Leistung geprüft werden.

Literaturverzeichnis

[Diaa] Diamond Dokumentation. https://diamond.readthedocs.io/en/latest/. Zugriff: 10.02.2017.

[Diab] Diamond GitHub-Seite. https://github.com/python-diamond. Zugriff: 10.02.2017.

[HBa] HBase Dokumentation. https://hbase.apache.org/book.html. Zugriff: 10.02.2017.

[Opea] OpenTSDB Dokumentation. http://opentsdb.net/docs/build/html/index.html. Zugriff: 10.02.2017.

[Opeb] OpenTSDB GitHub-Seite. https://github.com/OpenTSDB/opentsdb/. Zugriff: 10.02.2017.

Anhänge

Verwendete Software

Text- und Codebearbeitung	Atom Editor
Tabellenkalkulation	Excel, LibreOffice
Grafikerstellung	Tableau
LaTeX-Umgebung	Overleaf
Versionsverwaltung	git
Programm für Laufzeittest	partdiff-par aus HLR

Abbildungsverzeichnis

Codeabschnittsverzeichnis

Tabellenverzeichnis