UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Tõnis Ojandu

Partitioning Enterprise Systems on Docker Platform

Bachelor's Thesis (6 ECTS)

Supervisor: Satish Narayana Srirama, PhD

Tartu 2016

Ärisüsteemide partitsioneerimine Docker platvormil

Lühikokkuvõte: Ärisüsteemid on keerulised arvutisüsteemid, mis loovad väärtust nende omanikele. Reegilina süsteemide väärtused koos oma olemustega erinevad üksteisest, kuid süsteemid ise on ehitatud geneerilisematest komponentidest, mis on kokku seotud tegelikku väärtust lisava äriloogikaga. Antud süsteemid võivad olla paigutatud suurtesse riistvaraklastritesse. Sellest tulenevalt võib selliste süsteemide loomine ja haldamine nõuda märgatavalt vaeva.

Käesolev lõputöö uurib Dockerit, mis on tarkvara konteinerite platvorm. Seda kasutedes on püütud standardiseerida ning automatiseerida tehisliku ärisüsteemi juurtamist. Ühtlasi fokuseerib antud uurimistöö ärisusteemide partitsioneerimise automatiseerimisele mitmele riistvara sõlmele. Selle saavutamiseks testitakse METIS graafi partitsioneerimise teekide ja tööriistadega.

Võtmesõnad: Docker, Docker Swarm, Ärisüsteemid, Graafi partitsioneerimine, METIS, Riistvaraklaster

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Partitioning Enterprise Systems on Docker Platform

Abstract: Enterprise systems are complex computer systems that create business value to their owners. This value and the nature of these systems varies but usually they are built from many more general software components that are tied together with custom and value-adding business logic. These systems can expand extensive hardware clusters. All this means that they often require noticeable effort to develop and manage.

This thesis makes a study of a software container platform named Docker by standardizing and automating the deployment of a mock enterprise system. Also this study focuses on automating the partitioning of these enterprise system into multiple hardware nodes. It aims to achieve this using METIS graph partitioning toolset.

Keywords: Docker, Docker Swarm, Enterprise Systems, Graph partitioning, METIS, Hardware Clusters

CERCS: T120 Systems engineering, computer technology

Contents

1 Introduction

1.1 Motivation

Enterprise systems are large complex computer systems that are usually tasked with data storing, processing and presentation among other things. Essentially they are custom-built systems that provide business value to the organizations managing them. Although the systems themselves and their purposes vary, components that they are built of themselves are more universal. For example often enough these systems contain some sort of database technology, a web server, integrate or provide an integration capabilities via some sort of web services, all of which are tied together with more or less custom-built business logic that is constructed using more general software development technologies.

Often these systems are demanding resource-wise because they are built to serve many users and/or external systems simultaneously. Also these systems might have to scale when the resources allocated to them can no longer accommodate them. Therefore it is natural to assume that these systems usually span multiple or have a need to span additional hardware nodes. Splitting a system requires decisions to be made concerning partitioning of the individual components. When partitioning a system there can be many constraints hardware-wise. For example a component may require a faster hard disk or direct access to some unique hardware component. There can also be restrictions specified by the requirements for high-availability and redundancy. Also partitioning a system will create added communication overheads between the components when these components end up in different hardware nodes. Therefore it will make sense to deploy components that have a higher communication requirements close to each other. In addition deploying components that demand the most processing power together can cause significant loss in performance. This work does not define any specific constraints and requirements for the partitioning and entirely focuses on the communication overheads and allocation of processing power.

As mentioned enterprise systems, though usually unique themselves, are build of more general components. This does not although mean that there are that less challenges when deploying such systems. Usually these general components themselves are customizable and/or environment-sensitive enough to invoke necessity for additional configuring. Therefore deploying such systems can be demanding when it comes to counting the man-hours required for each install. This effort quickly replicates, since in addition to the actual production environment there is often a staging environment, testing environment and some sort of a development or continuous integration environment, It is also possible that this one type of system can be deployed for other organizations. To combat this there is a need for some sort of automation which can be achieved by using some packaging technologies, elaborate scripts etc. Docker is such a technology that seeks to standardize software deployment by generalizing the processes for installing, upgrading, starting, stopping, observing the individual software components and by isolating them from other components without actually invoking a need for visualization. This project makes a study of Docker technology by using it as the deployment framework.

1.2 Related Works

This work is largely an extension to the work done by S. N. Srirama and J. Viil [SV14]. They partitioned scientific workflows by using a graph partitioning library named METIS. This work aims to adapt this knowledge to enterprise systems by partitioning components typically found in enterprise systems using the same graph partitioning library. Docker platform will be used as a underlying framework for ease of deployment and managing the partitions. The goal of the project is to show that there are advantages to using the same techniques when partitioning and scaling an existing enterprise system and attempts to do so by constructing a mock enterprise system, deploying it onto a Docker platform, measuring the communication between the individual components, constructing a graph based on these observations, partitioning the components and deploying them onto a distributed Docker platform. By doing so the project aims to show that there are performance advantages to this process.

2 Docker

2.1 Docker Image

Docker is an open-source platform that allows deploying software in isolated Linux containers without requiring additional virtualization overhead [DIS]. Using Docker involves first installing Docker Engine and launching the Docker Daemon on the host machine. Software is deployed to Docker Engine using Docker Images. Docker Client is a binary that is used for communicating with Docker Daemon. Docker Client provides necessary tools for Dockerizing software (i.e., creating Docker Images from software resources). This is done by creating a Dockerfile. The Dockerfile defines steps for installing software, network ports the software will expose outside from the container, commands run on the resulting container and initialization controls when container will eventually be created from the image being defined. New Docker Images can be created using previous Docker Images as a starting point. In addition running instances of these images can be turned turned into new images. Although this is not the recommended approach from the software development point-of-view since often enough this will clutter the images with unrelated variable data such as logs. Also it will make the image harder to reproduce and therefore harder to version control.

A single Docker Image is a definition of a base image and a list of changes applied to it [DIM]. These changes are stored as binary changes to image and correspond to the steps defined in the Dockerfile. Changes in Dockerfile file will result in a redefinition of the changes and their binary differences. With this method the need to redefine the entire images from scratch when changes are applied is bypassed but this also means that the order of the steps in Dockerfile should start with the ones that are the least likely and end with the ones that are most likely to change. For example Dockerfile could start with installment of all the necessary tools concatenated as single command in order to minimize the resulting amount of large binary differences and would end with separated steps for installing libraries and the developed software. The latter increases the build time since the developed software is more prone to change and usually is a lesser binary difference to the image than the libraries. Sample of a Dockerfile following these guideline can be seen on Figure 1.

Docker Images created can be pushed to shared Docker Registries and also pulled from them [DRE]. Only the binary differences will be moved when pulling or pushing. Therefore the same care taken when defining the Dockerfiles will also benefit the network traffic when the images are moved. I can be observed that Docker Registry workflow mirrors Git version control workflow. Both are in their nature peer-to-peer. Like version-control systems it is possible to tag versions of images and return to them at a later date but unlike version-control systems it is

```
FROM java:8

RUN apt-get update \
    && apt-get install wget \
    && apt-get install uzip \
    && wget http://some-host/some-archive.zip \
    && unzip some-archive.zip

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./
   urandom","-cp","app/:lib/*","some.package.SomeMain"]

ADD lib lib
ADD classes app
```

Figure 1: Basic Dockerfile

not possible to return to previous revision since building, pushing or pulling an image with the same tag will overwrite the previous one. Therefore it is not recommended to rely on the Docker Registry as eternal repository because with simple missteps it is possible to overwrite the Registry. Therefore it is recommended as marked above to make the images as reproducible as possible using Dockerfiles and delegate the revision control to actual source code version control systems.

Docker community also provides a publicly usable registry with similar open source principals to Github named DockerHub [DHU]. Dockerhub already contains Docker Images for many open source projects. These include all from operating system base images for different GNU/Linux distributions, base images with runtime environments already installed (for example Java) to many popular deploy-ready pieces of software (like MongoDB, Elastisearch, Nginx etc). Dockerhub is also the default location when pulling images and no registry is explicitly defined.

Best practice is for the Docker Image to start only a single process when Docker Container is created [DPB]. This process would run in an isolated environment. This means it has its own file system and does not conflict with network ports of the other processes running on the host machine or in the other docker containers running. It should be noted that this is a perceived isolation because Docker gains by not utilizing virtualization but this also means that the processes running in the container are in fact running on the host machines. Since at this moment running Docker Container still requires root privileges then it is recommended to run the process as a less privileged user as demonstrated in the Dockerfile on Figure 2.

Usually it is necessary for Docker Containers to interact with each other or

```
FROM java:8

RUN apt-get update \
    && apt-get install wget \
    && apt-get install uzip \
    && wget http://some-host/some-archive.zip \
    && unzip some-archive.zip \
    && adduser some-user \
    && mkdir -p /var/log/some-app \
    && chown some-user: some-user /var/log/some-app

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./
   urandom","-cp","app/:lib/*","some.package.SomeMain"]

ADD lib lib
ADD classes app

USER some-user
```

Figure 2: Dockerfile With Custom User

with the host machine. To this end Docker Container can expose network ports on the private network address provided for it by the Docker Daemon. Also Docker Containers can be linked to other Docker Containers. This linkage will essentially map a hostname to the network address of the other Docker Container in the /etc/hosts file. In addition exposed ports can be mapped to the host environment network address if necessary. Exposing a port is shows in the Dockerfile on Figure 3.

Docker Containers can share data with the host environment by mounting directories as volumes [DVO]. Although here it should be noted that when doing this process user UID and GID can cause trouble with the file access rights because users created when the Docker Image is created will probably not match to the ones in the deployment environment. Therefore the proper file access have for the process will first have to be established and integrated. This integration can be achieved by either being generous with the file access rights, trying to align the UID and/or GID with the host environment or by using some tools like "gosu" [GSU]. In addition to the environment directory mounting it is also possible to create specialized Volume Containers that can be mounted to other Docker Containers.

```
FROM java:8

RUN apt-get update \
    && apt-get install wget \
    && apt-get install uzip \
    && wget http://some-host/some-archive.zip \
    && unzip some-archive.zip \
    && adduser some-user \
    && mkdir -p /var/log/some-app \
    && chown some-user: some-user /var/log/some-app

EXPOSE 8080

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./
   urandom","-cp","app/:lib/*","some.package.SomeMain"]

ADD lib lib
ADD classes app

USER some-user
```

Figure 3: Dockerfile With Exposed Port

2.2 Docker clustering and scaling

Multiple Docker environments can be clustered together into a larger system. There are a myriad of methods for achieving this. They are really dependent on the actual system that would be clustered. Docker provides the means for doing it manually by using the inter-Container communication described above along with the features provided by Docker Network. If possible though it is recommended to use existing and more automatic tools and frameworks like Docker Swarm and/or Docker Weave.

2.2.1 Docker Swarm

Docker Swarm is native clustering solution for Docker Engines [DSW]. Docker Swarm is made up of one Swarm Master node and any number of slave nodes. Essentially Docker Swarm can be created using the Swarm tool that itself can be launched as a Docker Container. Docker Swarm can be created manually or the process can be simplified and sped up by using Docker Machine. Once a cluster has been created new Docker Containers are started through Swarm Master,

```
FROM jessie:8

ADD vol /mnt/vol

VOLUME /mnt/vol
```

Figure 4: Dockerfile For Docker Volume

which spreads the containers across the slave node cluster. For this experiment more control was required when directing Docker Containers to Docker Daemons. Docker Swarm provides this capability. When Swarm nodes are created then the daemons can be launched with key value pair labels and when creating container it is possible to control where they end up by providing criterias.

2.2.2 Docker Machine

Docker Machine is tool for spinning up multiple instances of host machines with Docker Engines installed and Docker Daemons running on them [DMA]. It also acts as a simplified interface for starting and managing Docker containers already running on the machines that have been spun up by the Docker Machine. Docker Machine supports multiple drivers like VirtualBox [DMV], Amazon AWS [DME], Digital Ocean [DMO], VMWare [DMW] etc. This way it is easy to scale from development environment to production environment by just changing the driver in deployment scripts. For this project development was done using VirtualBox driver in local environment and measurements were conducted using Digital Ocean platform [DIO] and driver. Docker Machine can also deploy Docker Swarm when machines are spun up.

2.2.3 Docker Weave

Docker Weave is an open source framework build on top of Docker platform that is developed and maintained by Weavework company [DWE]. Using Docker Weave involves launching Weave images on all the Docker Engines that are to be clustered together and configuring Docker Daemon to use these images as command wrappers when issuing commands to Docker Engine. These Weave images maintain a private network and provide a Domain Name Service therefore it is not necessary to link the containers manually. Since Docker Swarm at this point does not support cross Docker Engine linking then Docker Weave can be used can be used to mitigate lack of functionality. Combining Docker Machine, Docker Swarm and Docker Weave creates a tool set where through a single interface it is possible to spin up extensive cluster with a system of Docker Containers running on top it.

3 Enterprise System

In order to demonstrate enterprise system partitioning a mock enterprise system was created that utilizes some of the popular technologies used in enterprise environments (See Figure 5). Essentially the system has a mocked integration point to a external system that generates simple events. These events are pairs of alphanumerical keys and numerical values. For ease of use and demonstration this integration point was implemented with HTTP being the underlying technology. The system exposed a RESTful API and provided a single page web front-end which was not explicitly used as part of the experiment but served more as a demonstration on how underlying the underlying RESTful endpoint would be used.

Mock enterprise system was used to demonstrate a scenario where there preexisted an enterprise system that had been deployed onto a single node. Now that the system had grown to a size that had trouble performing inside this single node environment there arouse a need to move it to a multiple node environment. This experiment aimed to demonstrate a method for doing this by first Dockerizing the components in that system and then using a combination of tools in order to deploy the system onto the multiple node environment in a performance saving efficient manner. This idea behind this demonstration was to prove that it could be applied to some generic enterprise system.

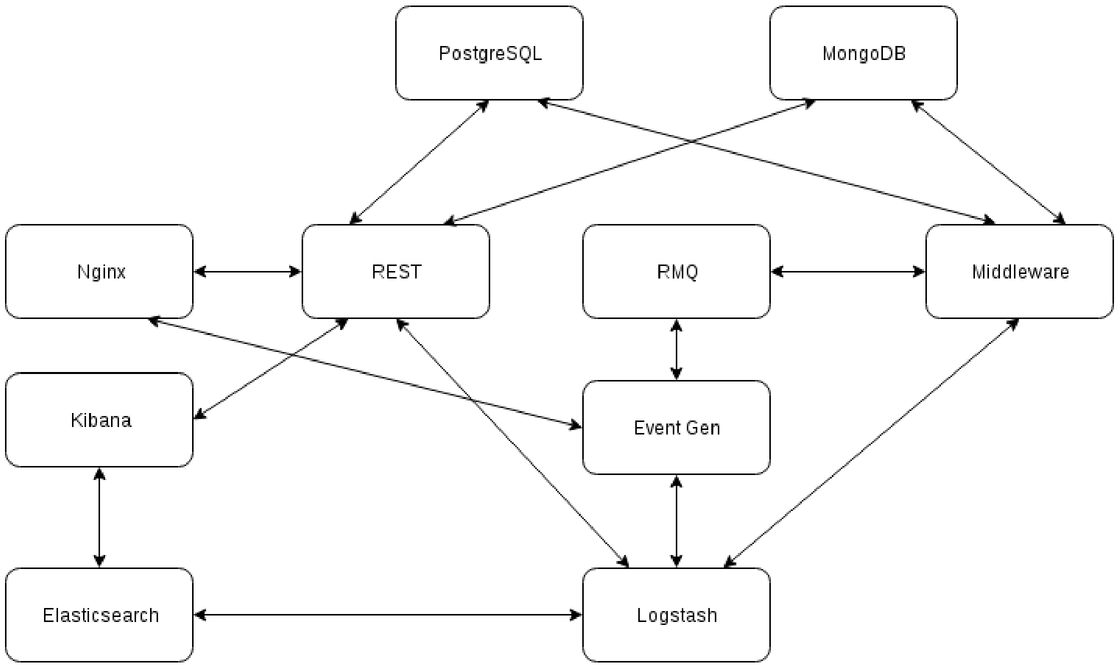

Figure 5: Mock Enterprise System

3.1 RabbitMQ

RabbitMQ is a widely used open source messaging broker that utilizes AMQP protocol (currently AMQP version 0.9.1) [RMQ]. For this experiment events events coming from a mock external system were directed through RabbitMQ messaging broker.

3.2 MongoDB

MongoDB is a widely used open source NoSQL database. It can be used to store and query JSON documents [MON]. It also provides tools for querying aggregated info about the the documents stored. For this experiment MongoDB was used as a store for aggregated information about events coming from the external system.

3.3 PostgreSQL

PostgreSQL is a widely used open source SQL database [PSQ]. It is mainly used to store relational data. For this experiment we are using PostgreSQL to store character counters describing the events coming from the external system.

3.4 Elasticsearch

Elasticsearch is a NoSQL JSON database and index, that is in principle aimed to scale elastically and provide fast query capabilities [ELS]. For this experiment Elastisearch was used to store log data from Stream Generator, Middleware and RESTfulService.

3.5 Logstash

Logstash is a event ETL and pipe service [ELS]. Essentially Logstash handles wide variety of input and output formats and provides tools for filtering or manipulating the events it processes. For this experiment Logstash was integrated a Log4j SocketAppender and Stream Generator, Middleware and RESTful Service were logging to it. Logstash collected the logs and stored them to Elastisearch.

3.6 Kibana

Kibana is a Rapid application development tool for visualizing data in Elastisearch [ELS]. For this experiment no visualization was defined acted as an appropriate gateway for log data in Elasticsearch.

3.7 Stream Generator

Stream generator was implemented as the mock integration point to an external system that is sending events. These events are alphanumerical key and numerical value pairs. Since this was a mock system then for the sake of simplicity the integration was implemented as a HTTP API that on invocation generated a new random event. The Stream Generators enqueued the Events generated into a RabbitMQ broker queue. Stream Generator was implemented as a Java Spring Boot [SBO] application. Logs created by the Stream Generator during runtime were sent via Log4j [L4J] SocketAppender to Logstash.

3.8 RESTful Service

RESTful Service acts as a back-end for web front-end. It provides endpoints for requesting the last 15 minutes of logs from Elasticsearch via Kibana, aggregated event data obtained from MongoDB and character counter data about the events from PostgreSQL. RESTful Service was implemented as a Java Spring Boot application. Logs created by the RESTful Service during runtime were sent via Log4j SocketAppender to Logstash.

3.9 Middleware

Middleware read events obtained from the event queue in RabbitMQ, aggregated them and stored the aggregation results to RabbitMQ. It also maintained counter for characters encountered in the events and persisted this information in PostgreSQL. Middlware is implemented as a Java Spring context [SFW] application. Logs created by the Middleware during runtime were sent via Log4j sSocketAppender to Logstash.

3.10 Nginx

Nginx is a widely used HTTP server [NGX]. For this experiment Nginx was Dockerized along with Web front-end static resources. Those were made up of mostly HTML, CSS and Javascript. Also Nginx was configured to act as reverse proxy request to RESTful Service and Stream Generator event generation endpoint. As a result Nginx served a Web front-end for the rest of the system and by collecting all the endpoint simplified the eventual load generation.

4 Final Implementation

A CLI environment was implemented that tied together all the technologies relevant to this experiment. Namely system deployment, measurements and partitioning. This environment was developed using Java as the programming language.

4.1 Enterprise System Deployment

Three types of Docker images were involved in the deployment of the mock enterprise system. In order to deploy RabbitMQ [DHR], MongodDB [DHM], PostgreSQL [DHP], Elasticsearch [DHE] and Kibana [DHK] Docker images were used without any alterations that are available in the publicly in DockerHub. Logstash [DHL] and Nginx [DHN] were also deployed using publicly available Docker Images but were altered. Namely to Logstash image configuration detailing the method of log collection was added. Nginx image was altered by adding the configuration for REST request proxies and static web resources like HTML, CSS and Javacsript files. These alterations were stored as custom Docker images in DockerHub named "vampnik/pesd-logstash" for Logstash and "vampnik/pesd-fe" for Nginx. Docker images for Middleware, Stream Generator and RESTful Service, were at most part, custom developments and actually implemented the mock business logic for the mock enterprise system. Docker images for these components were based on Java base [DHJ] Docker image that is available publicly in DockerHub. These images were also stored publicly for eventual use in distributed and remote deployment as "vampnik/pesd-mw" for Middleware, "vampnik/pesd-stream-generator" for Stream Generator and "vampnik/pesd-rest" for RESTful Service.

The CLI environment was capable of deploying the mock enterprise system as a single node local Docker Engine deployment and a remote multi-node Docker Swarm deployment. Local deployment was invoked using "start-local" command in the CLI environment. Local deployment could be torn down using "stop-local" command. When initiating a multi node deployment it was possible to choose either to opt for random partitioning or for a one defined in a partition file. Random partitioning is chosen by shuffling the list of components. Partition number for each components is defined as $(i \mod p) + 1$ where p is the the number of partitions requested and i is the index number for that component in the shuffled list. For example random deployment onto a Docker Swarm with 4 nodes could be invoked using "start-weave random4" and a deployment using predefined partition fail named "somePart.graph" could be invoked using "start-swarm somePart". For both cases Docker Swarm deployment could be torn down using "stop-swarm" command.

Local experiment environment was Debian 3.16.7 with Docker Engine installed running on 16GB of memory, 4 processor core Intel(R) Core(TM) i7-3520M CPU

@ 2.90GHz, 1TB SSD Disk. Remote experiment environment was a Docker Swarm that consisted of three Digital Ocean droplets with 2GB of memory, 2 processor cores and 40GB SSD Disk spun up by Docker Machine.

4.2 Resource Usage Measurement

For this experiment partitioning of the enterprise system was done based on the network traffic and processing power usage. Network traffic measurements were used as the weights for the edges and processing power proportions were used as the weights for the vertices to construct a graph representing the mock enterprise system.

There are various way to characterize network traffic. Since this project aimed to partition the components so that the ones communicating the most would end up in the same partition then the total amount of bytes communicated between components was used as the network traffic measurement characteristic. Each Docker container has its own isolated virtual network device. This makes it easier to observe network traffic from the container point of view. Since there was a mock enterprise system component per container as recommended by Docker best practices [DPB] then the network traffic observed for of each of the containers are only the communication to and from the component they specifically contain. Therefore Docker provides excellent isolation to determine network traffic of each of the nodes specifically.

In order to measure the number of bytes communicated between the containers 60 seconds of TCP dump output from the virtual network interface of the all the containers was collected concurrently. This information was in turn aggregated and used to define the weighed edges for the mock enterprise system communication graph.

This measuring process was established so as to there would be minimal prerequisites to the enterprise system being monitored. To this end common Linux tool "tcpdump" was used. Also Docker Images for the components were not altered in any way. Instead all the monitoring was achieved by using Docker Daemon "exec" tool that is meant to externally invoke commands in the Docker container shell. This way it was demonstrated that the same method could be applied to any system of Docker container and consequently enterprise systems. The CLI environment would perform this measurements for local deployment when invoking "measure-local" and for Docker Swarm deployment when invoking "measure-swarm" (as demonstrated on Figure 6). Output of this measurement is the number of total and pair-wise breakdown of the number of bytes communicated between the components.

There are also various methods for describing processing power used by components. Since Docker lacks virtualization layer then the processes running in the containers actually run on the kernel of the host machine. Therefore observing the CPU-usage in the process snapshot provided by the "ps" command it is possible to obtain an overview of the proportions of the processing power used by each of the mock enterprise system component. For this experiment "ps -e -o %cpu,%mem,cmd" was run for 60 seconds with 1 second intervals concurrently with network traffic measuring. For each process an average CPU-usage was calculated from the 60 snapshots thus obtained. These averages were in turn used as the weights of the vertices in the resulting graph representing mock enterprise system.

```
> measure-swarm
..
// measurement log
..
Total: 285354864 | MeasurementResult{cpu={logstash
   =14.334360567162491, rest=32.018150493800206,
   elasticsearch=13.907290124804861, nginx
   =12.124594566175698, rmq=3.359944351427209, evGen
   =2.1005718537930806, mw=9.918872792862743, kibana
   =0.682665631344381 4, mongoDb=3.561346889584497, sql
   =7.992202729044834}, memory={logstash
   =9.170139063205974, rest=13.25468145991264,
   elasticsearch=12.734575028866924, nginx
   =0.4518299111401176, rmq=4.178924644811486, evGen
   =8.129926201114513, mw=14.513780812289784, kibana
   =4.305437019930715, mongoDb=1.516140368492394, sql
   =31.744565490235445}, network=[rmq <-> mw : 1381473,
   mongoDb <-> rest : 2865360, elasticsearch <->
   logstash : 89247749, kibana <-> rest : 66529,
   logstash <-> rest : 7989785, rmq <-> evGen : 23847,
   mongoDb <-> mw : 22740, sql <-> rest : 39195302,
   logstash <-> evGen : 27760, sql <-> mw : 1055049,
   elasticsearch <-> kibana : 3380576, logstash <-> mw :
    76260980, rest <-> nginx : 63827730, evGen <-> nginx
    : 9960, rmq <-> rest : 24]}
```

Figure 6: Invoking Measurment

4.3 Partitioning

For the purposes of this experiment the mock enterprise system was represented as an undirected graph $G = (V, E)$ where $V = v_1, v_2, ..., v_n$ are the components in the system, the weight of the vertices represent the proportion of processing power used by the component and the weights of the edge $(v_i, v_j) \in E$ denote the amount of network traffic between the components v_i and v_j inside some fixed interval α [SV14]. The goal was to obtain fixed number of partitions of this graph so that the sum of weights crossing the partition is minimized and vertice weight sums of all the partitions are as smiliar as possible. This partitioning was to be used as a deployment layout for the mock enterprise system onto a distributed infrastructure. Aim of this was to demonstrate that the system performed better using a layout obtained by this means in comparison to layouts generated randomly.

Since graph partitioning is a NP-complete problem then computing an deterministically optimal solution becomes increasingly unviable as the size of the graph increases [SV14]. In order for this method to be useful for actual enterprise systems, it has to function for larger or increasing number of components. To combat this heuristical algorithms can be used that do not produce the most optimal but good-enough partitioning. To this end graph partitioning tool METIS, that implements number of such heuristical partitioning algorithms, was used for this experiment [KK98]. Both multilevel k-way partitioning and multilevel recursive bisection algorithm were applied to determine the better candidate [KK11].

A CLI was developed to conduct this experiment. Among other capabilities it could apply METIS algorithms to the graph obtained by the network traffic measurement. This could be achieved by invoking "part-local *n name*" where *n* denotes the number *name* will be used as the base name like demonstrated on Figure 7. When partitioning was invoked on the CLI the it would measure the communication and immediately perform partitioning using the multilayer k-way algorithm (denoted as $TMIN$ in the output) and mulitlayer recursive bisection algorithm (denoted as REC in the output). Here it should be noted that logarithm was applied to the network measurement or the edge weights before partitioning in the following manner $\alpha \log(w_{ij})$ where $\alpha > 1$ is constant for increasing the resulting value. METIS requires integer weights [KK11] and by multiplying the weight with a constant the resulting weights avoid being equal after rounding (for experiments $\alpha = 100$ was used). Logarithm helped avoid METIS having to deal with large integers while still retaining some proportional relativity among the weights. This way any possible integer overflows within METIS were avoided. Also when the weights were different by a large order of magnitude the resulting partitions tended to be chaotic. After applying the logarithm quality of the partitions increased significantly.

5 Approach

5.1 Process

Aim of the project was to show that by partitioning the nodes in a manner where components communicating the most among each other would end up in the same partition while ensuring that the processing resources are as evenly distributed as possible would yield better overall performance. Mock enterprise system deployed for this project was so designed that in quiet mode there would be little to no network traffic. Namely when there are no external requests towards the system then the only traffic between the components would be background heartbeats This means that external HTTP request would create almost all of the network traffic between the components.

In order to generate load for the system a custom load testing system was developed. This system continuously created new load testing threads that concurrently made HTTP requests specified endpoints. For each of the endpoint there was maximum request duration defined. Generation of new load testing threads stopped for endpoint when the average duration for the request that have ended in previous 60 seconds has reached this maximum value. Using the number of load testing threads as a metric it was possible to compare the performance of different partitioning layouts. For this experiment the maximum duration defined for the requests was 1 second.

The load was created through single interface, namely through Nginx via HTTP. Consequently this made the load on the system more easily managed and quantified. For load creation four endpoints were used:

1. REST GET /api/stat request for event statistics in MongoDB;
2. REST GET /api/chars request for event key character counters in PostgreSQL;
3. REST GET /api/logs request for logs from Elasticsearch (via Kibana);
4. REST GET /evgen request to initiate an event in stream generator.

First step was to to determine an effective partitioning of the components. Since in the mock enterprise system contained ten components so the number of partitions was fixed to be three. Three nodes would create enough possible layout variations so to demonstrate that directed partitioning has a advantage over a random layout. In order to create such a partitioning the mock enterprise system was first deployed to a local Docker Engine. Load generation was initiated. After generation of new load testing threads had capped the TCP dump information and CPU usage was collected and aggregated. Resulting graph was partitioned using METIS and the partition was stored for later use.

Three layouts generated randomly and layouts that were provided by both algorithms implemented in METIS would be deployed onto a Docker Swarm deployment with three nodes. For all these deployments load generation would be initiated. Once number of load testing thread generation would have capped for all the endpoints then these numbers would be recorded for performance comparison. In this case when comparing two partitioning layout the one layout that could handle more concurrent load testing threads, meaning it could service more clients in a timely manner, was to be considered to perform better.

5.2 Results

Using the above mention method for local deployment network traffic graph edges were measured as shown on Table 1 and CPU usage was measured as show on Table 2 (also see figure 8):

First Component	**Second Component**	**Network Traffic**
RabbitMQ	Middleware	~ 1MB
MongoDB	RESTful Service	~ 150kB
Elasticsearch	Logstash	~ 69MB
Kibana	RESTful Service	~ 25kB
Logstash	RESTful Service	~ 405kB
RabbitMQ	Stream Generator	~ 12kB
MongoDB	Middleware	~ 22kB
PostgreSQL	RESTful Service	~ 3MB
Logstash	Stream Generator	~ 37kB
PostgreSQL	Middleware	~ 913kB
Elasticsearch	Kibana	~ 6MB
Logstash	Middleware	~ 65MB
RESTful Service	Nginx	~ 3GB
Stream Generator	Nginx	~ 12kB

Table 1: Network Traffic Measurement Breakdown On Local Deployment

When applying multilevel recursive bisection algorithm the resulting partitions were as show in Table 3 and when multilevel k-way partitioning was applied then the resulting partitions were as shown on Table 4. It can be observed that both algorithms essentially provided the same partitioning only with different indexes. Therefore this single layout provided by METIS along with three random layouts (show in Tables 5, 6 and 7) were deployed onto a Docker Swarm cluster. After

Component	**CPU usage**
Logstash	11.27%
RESTful Service	32.24%
Elasticsearch	19.02%
Nginx	10.85%
RabbitMQ	4.68%
Stream Generator	3.32%
Middleware	13.9%
Kibana	0.92%
MongoDB	0.79%
PostgreSQL	2.95%

Table 2: CPU Usage On Local Deployment

the load testing thread generation had stopped thread counts per endpoint were recorded. Number of concurrent load testing threads that could be served by these layouts in a 3-node Docker Swarm cluster can be seen in Table 8 and Figure 9.

Partition	**Components**
1	Elasticsearch, Logstash, Kibana
2	Middleware, Nginx, RabbitMQ, MongoDB, PostgreSQL, Stream Generator
3	RESTfulService

Table 3: Recursive Partitions

Partition	**Components**
1	Elasticsearch, Logstash, Kibana
2	RESTful Service
3	Middleware, Nginx, RabbitMQ, MongoDB, PostgreSQL, Stream Generator

Table 4: k-way Partition

It can be seen from the results that there was no difference for number of threads supported for "/api/logs" and "/api/evgen" between all of the layouts. All could only sustain a single load testing thread. This was caused by the fact that request times for these endpoints during the load testing were always above 1 seconds. Since the criteria for adding new load testing threads was that the average request time should be under 1 second then no new threads were created. Therefore no meaningful comparison can be made based on the results for these two endpoint. However the simulated load created through these endpoints during the load testing was still necessary, since without them some of the components

Partition	**Components**
1	Logstash, RESTful Service, PostgreSQL, Stream Generator
2	Nginx, Elasticsearch, RabbitMQ
3	Middleware, MongoDB, Kibana

Table 5: Random 1

Partition	**Components**
1	Elasticsearch, Logstash, RESTful Service, Stream Generator
2	RabbitMQ, Kibana, PostgreSQL
3	Nginx, Middleware, MongoDB

Table 6: Random 2

would not have experienced any stress. This in turn would have made the results for other endpoints unrealistic.

Results for "/api/stat" and "/api/chars" endpoint show that layout provided by METIS outperformed random layouts. This shows clearly that there is an advantage when using the described method for partitioning an enterprise system.

5.2.1 Scaling

Extension to the scenario where an enterprise system is on a single node and has to be partitioned to multiple nodes in order to scale performance-vise is the scenario where an enterprise system is already on a multiple node system and has to scale to additional nodes due to lack in either performance and/or resource capabilities. To this end 3 node layout already provided by METIS was deployed onto a Docker Swarm cluster. Same type of load was generated. After the load testing thread generation had capped 60 seconds of network traffic and processing power usage was collected. This data was in turn used to create a new graph with weighed edges and vertices and partitioned into 4-node layouts. These partitions provided by METIS and 3 random partitions were deployed to Docker Swarm cluster. Maximum number load testing threads that could be service in timely manner were determined for all of these layouts. Results can be seen in Table **??** and Figure 10. As previously data from "/api/logs" and "/evgen" endpoints cannot be used to make any meaningful distinctions between the layouts but are still needed to simulate load to all the components. Results "/api/stats" and "/api/chars" demonstrates that the same method can be reapplied to further scale the an enterprise system deployment.

Partition	**Components**
1	Middleware, MongoDB, Kibana, Stream Generator
2	Nginx, Logstash, PostgreSQL
3	Elasticsearch, RabbitMQ, RESTful Service

Table 7: Random 3

	/api/stat	**/api/chars**	**/api/logs**	**/api/evgen**
METIS	378	371	1	1
Random 1	275	337	1	1
Random 2	232	266	1	1
Random 3	313	327	1	1

Table 8: Concurrent Load Test Threads Supported in 3-node Swarm Cluster

	/api/stat	**/api/chars**	**/api/logs**	**/api/evgen**
k-way	872	646	1	1
Recursive Partitioning	636	547	1	1
Random 1	376	454	1	1
Random 2	276	239	1	1
Random 3	387	388	1	1

Table 9: Concurrent Load Test Threads Supported in 4-node Swarm Cluster

```
> part-local 3 part
..
// partitioning log
..
> part-local 3 part
Total: 3046931636 | MeasurementResult{cpu={logstash
   =11.270594108836745, rest=32.24716496683546,
   elasticsearch=19.02592539761786, nginx
   =10.851579773197342, rmq=4.688502959845945, evGen
   =3.32626060908637, mw=13.909492903501889, kibana
   =0.925397617858926, mongoDb=0.7979102774409805, sql
   =2.9571713857784774}, memory={logstash
   =7.168458781362015, rest=43.39406610911987,
   elasticsearch=8.064516129032254, nginx=0.0, rmq
   =2.090800477897253, evGen=6.869772998805253, mw
   =28.82815611310235, kibana=2.389486260454, mongoDb
   =0.8960573476702519, sql=0.29868578255675}, network=[
   rmq <-> mw : 1224342, mongoDb <-> rest : 153324,
   elasticsearch <-> logstash : 72649760, kibana <->
   rest : 25505, logstash <-> rest : 415231, rmq <->
   evGen : 12183, mongoDb <-> mw : 22738, sql <-> rest :
    2681303, logstash <-> evGen : 37494, sql <-> mw :
   935046, elasticsearch <-> kibana : 5785629, logstash
   <-> mw : 67634007, rest <-> nginx : 2895342754, evGen
    <-> nginx : 12320]}
REC: [[ELASTIC_SEARCH, LOGSTASH, KIBANA], [MIDDLEWARE,
   NGINX, RABBIT_MQ, MONGODB, SQL, EVENT_GENERATOR], [
   REST]]
TMIN: [[ELASTIC_SEARCH, LOGSTASH, KIBANA], [REST], [
   MIDDLEWARE, NGINX, RABBIT_MQ, MONGODB, SQL,
   EVENT_GENERATOR]]
```

Figure 7: Invoking Partitioning

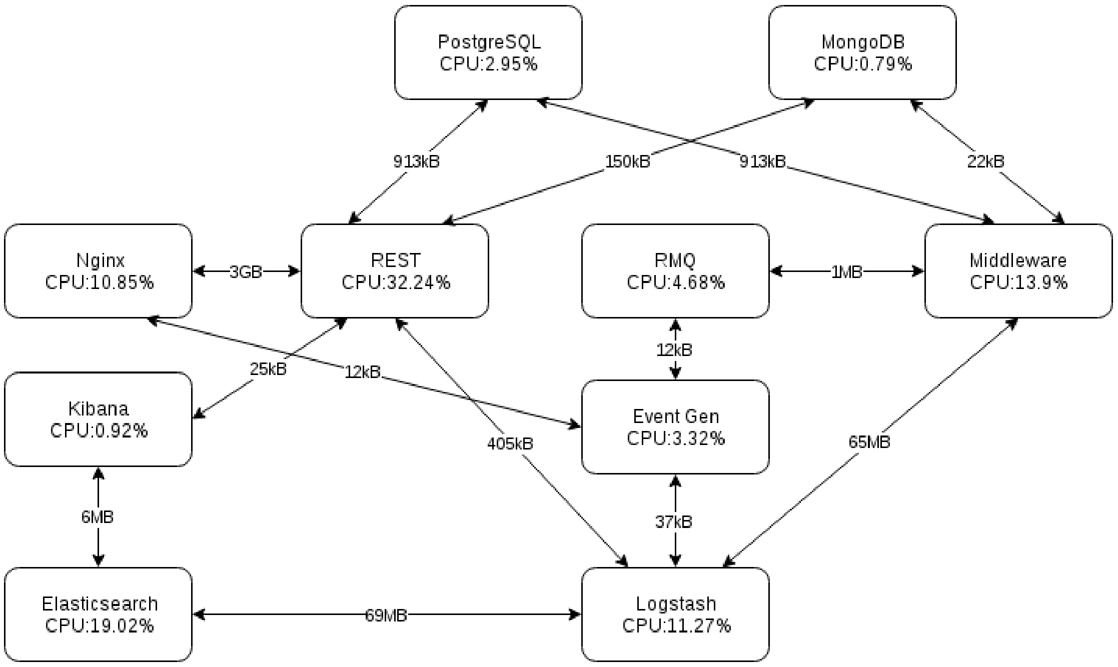

Figure 8: Mock Enterprise System Weighed Graph

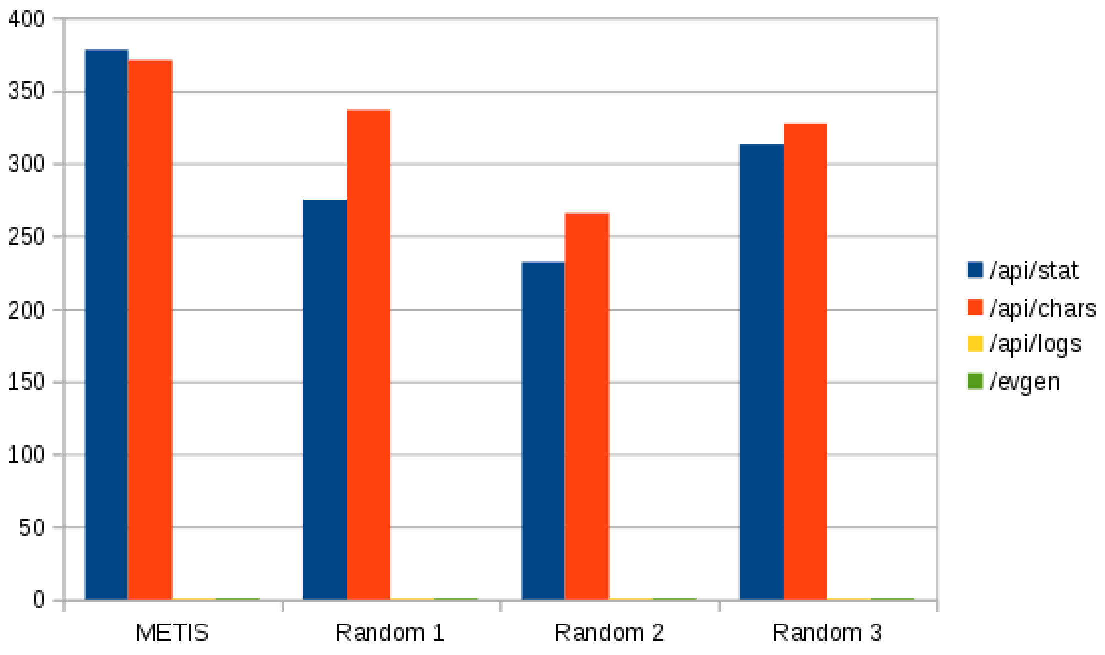

Figure 9: Concurrent Load Test Threads Supported in 3-node Swarm Cluster

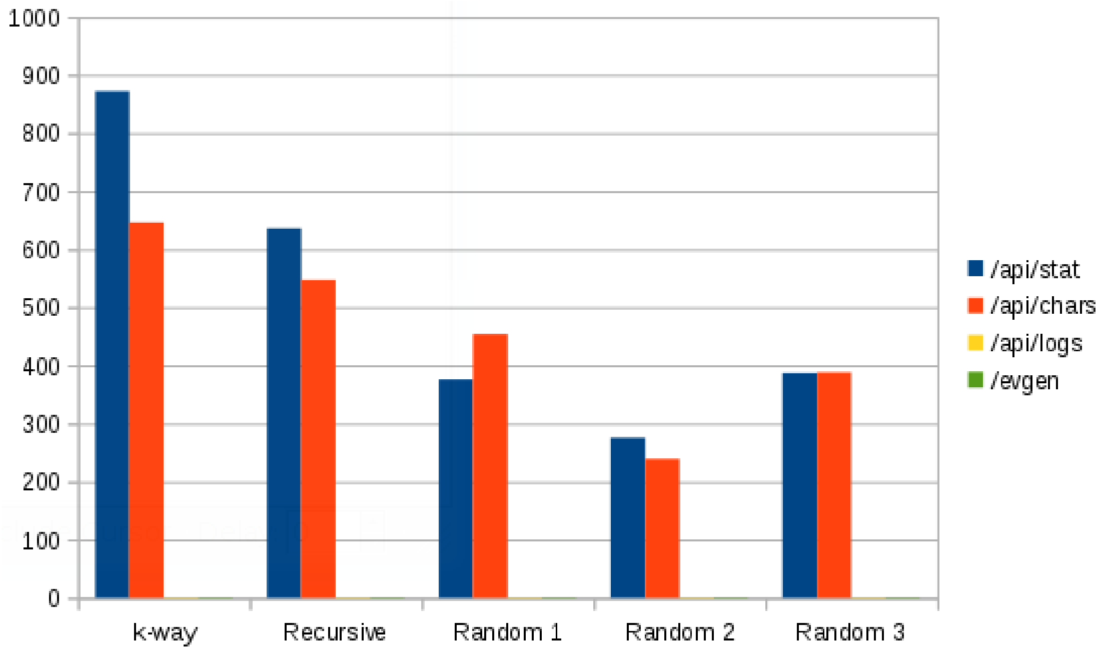

Figure 10: Concurrent Load Test Threads Supported in 4-node Swarm Cluster

6 Conclusion

Docker proved to be very clean platform for software deployment. Initial use of Docker does not pose a steep learning curve. However as the complexity of the system increases then additional challenges can be encountered. For example file access rights can cause problems when sharing volumes with the host machine while at the same time running processes as a non-superusers. Dockerizing software inefficiently is a common hurdle. Inefficiencies can noticeably slow development processes. Therefore it is recommended to invest time into understanding how Docker deployment works. Because Docker does not utilize virtualization then during the initial experimentation it is possible to corrupt the host machine to the extent that a system reset is required. At the same time the main advantages for using Docker come from the same fact that there is not an absolute isolation between the host machine and Docker Containers. This in turn means that there are far less resource overheads while still providing isolation like a separated filesystem and a separated network device. Thus Docker is not limited to any specific workflow and can be used for those that require high performance as well as to those employing microservice software architecture. It should also be noted that technologies around Docker are currently actively changing and not all the features have been fleshed out. For example Docker native clustering technology Docker Swarm still currently does not support cross-host container linking. This invokes a need for alternatives like ambassador-pattern or Docker Weave that are not ideal.

METIS was shown to viable toolset for partitioning enterprise systems. There was noticeable performance improvements when deploying layout provided by METIS compared to randomly generated layouts. During the experiments it became apparent that enterprise system specifically benefited from the fact that METIS could partition graphs with weighted vertices. Namely many enterprise system components may have little communication between each other but are more demanding for other hardware resources. Mainly for processing power. This means that by deploying together components that require more processing power can significantly reduce the overall performance. Since METIS could partition graphs with weighted vertices in a manner where in addition to minimizing the edge cut an attempt was made to ensure that resulting partitions had as equal a vertex weight sum as possible then this issue was mitigated.

Source code for the experiments is available at https://bitbucket.org/Vampnik/pesd

6.1 Future Directions

Since Docker ecosystem is still rapidly developing then there are new technologies and solutions still around the corner that wait to be explored and experimented with.

CLI developed for this experiment could also be expanded. Namely the management portion of the source code refactored to meet better coding practices. Current solution is partially hard-coded around the experiment therefore the management could be extracted to act as a stand alone tool or a library.

Also this experiment was limited by using network traffic and CPU as metrics when describing an enterprise system. However in an realistic scenario there often are additional criterias. For example all the hardware nodes might not be equal, there can be additional restrictions where a component can be deployed. In addition other hardware resource metrics like memory usage may play a role when deploying. There is room to investigate how well METIS can be adapted to these additional criterias.

References

[DHE] Docker Hub Elasticsearch Image. https://hub.docker.com/_/elasticsearch/. Accessed: 2016-08-10.

[DHJ] Docker Hub Java base Image. https://hub.docker.com/_/java/. Accessed: 2016-08-10.

[DHK] Docker Hub Kibana Image. https://hub.docker.com/_/kibana/. Accessed: 2016-08-10.

[DHL] Docker Hub Logstash Image. https://hub.docker.com/_/logstash/. Accessed: 2016-08-10.

[DHM] Docker Hub MongoDB Image. https://hub.docker.com/_/mongo/. Accessed: 2016-08-10.

[DHN] Docker Hub Nginx Image. https://hub.docker.com/_/nginx/. Accessed: 2016-08-10.

[DHP] Docker Hub PostgreSQL Image. https://hub.docker.com/_/postgres/. Accessed: 2016-08-10.

[DHR] Docker Hub RabbitMQ Image. https://hub.docker.com/_/rabbitmq/. Accessed: 2016-08-10.

[DHU] Docker Hub. https://hub.docker.com/. Accessed: 2016-08-10.

[DIM] Docker Images. https://docs.docker.com/engine/userguide/eng-image/baseimages/. Accessed: 2016-08-10.

[DIO] Digital Ocean. https://www.digitalocean.com/. Accessed: 2016-08-10.

[DIS] Understanding Docker. https://docs.docker.com/engine/understanding-docker/. Accessed: 2016-08-10.

[DMA] Docker Machine. https://docs.docker.com/machine/overview/. Accessed: 2016-08-10.

[DME] Docker Machine Amazon AWS driver. https://docs.docker.com/machine/drivers/aws/. Accessed: 2016-08-10.

[DMO] Docker Machine Digital Ocean driver. https://docs.docker.com/machine/drivers/digital-ocean/. Accessed: 2016-08-10.

[DMV] Docker Machine Oracle VirtualBox driver. `https://docs.docker.com/machine/drivers/virtualbox/`. Accessed: 2016-08-10.

[DMW] Docker Machine VMware vSphere driver. `https://docs.docker.com/machine/drivers/vsphere/`. Accessed: 2016-08-10.

[DPB] Docker Best practices. `https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/`. Accessed: 2016-08-10.

[DRE] Docker Registry. `https://docs.docker.com/registry/introduction/`. Accessed: 2016-08-10.

[DSW] Docker Swarm. `https://docs.docker.com/swarm/overview/`. Accessed: 2016-08-10.

[DVO] Docker Volumes. `https://docs.docker.com/engine/tutorials/dockervolumes/`. Accessed: 2016-08-10.

[DWE] Docker Weave. `https://www.weave.works/`. Accessed: 2016-08-10.

[ELS] Elasticsearch. `https://www.elastic.co/`. Accessed: 2016-08-10.

[GSU] Gosu. `https://github.com/tianon/gosu/`. Accessed: 2016-08-10.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KK11] George Karypis and V Kumar. Metis manual. *University of Minnesota/Department of Science/Army HPC Research Center*, 2011.

[L4J] Log4j. `http://logging.apache.org/log4j/2.x/`. Accessed: 2016-08-10.

[MON] MongoDB. `https://www.mongodb.com/`. Accessed: 2016-08-10.

[NGX] Nginx. `https://nginx.org/`. Accessed: 2016-08-10.

[PSQ] PostgreSQL. `https://www.postgresql.org/`. Accessed: 2016-08-10.

[RMQ] RabbitMQ. `https://www.rabbitmq.com/`. Accessed: 2016-08-10.

[SBO] Spring Boot. `http://projects.spring.io/spring-boot/`. Accessed: 2016-08-10.

[SFW] Spring Framework. `https://projects.spring.io/spring-framework/`. Accessed: 2016-08-10.

[SV14] Satish Narayana Srirama and Jaagup Viil. Migrating scientific workflows to the cloud: Through graph-partitioning, scheduling and peer-to-peer data sharing. In *16th IEEE International Conference on High Performance and Communications (HPCC 2014) workshops*, pages 1137–1144, 2014.