Building a high throughput microscope simulator using the Apache Kafka streaming framework

Lovisa Lugnegård
Abstract

**Building a high throughput microscope simulator using the Apache Kafka streaming framework**

*Lovisa Lugnegård*

Today microscopy imaging is a widely used and powerful method for investigating biological processes. The microscopes can produce large amounts of data in a short time. It is therefore impossible to analyse all the data thoroughly because of time and cost constraints. HASTE (Hierarchical Analysis of Temporal and Spatial Image Data) is a collaborative research project between Uppsala University, AstraZeneca and Vironova which addresses this specific problem. The idea is to analyse the image data in real time to make fast decisions on whether to analyse further, store or throw away the data. To facilitate the development process of this system a microscope simulator has been designed and implemented with large focus on parameters relating to data throughput. Apart from building the simulator the framework Apache Kafka has been evaluated for streaming large images. The results from this project are both a working simulator which shows a performance similar to that of the microscope and an evaluation of Apache Kafka showing that it is possible to stream image data with the framework.
Populärvetenskaplig sammanfattning


Detta examensarbete har handlat om att utveckla en mikroskopsimulator som ska kunna användas för att testa andra delar av HASTE-systemet. Simulaturn är både enklare att starta, och billigare att köra, än det riktiga mikroskopet. Simulatorn gör det också möjligt att undersöka hur systemet hanterar datamängder som är större än de som mikroskopet nu kan generera. Eftersom stora datamängder ska behandlas har fokus varit att undersöka parametrar som påverkar datagenomströmning, i detta fall frekvensen som bilderna kommer med och storleken på bilderna.

Förutom design och implementation av simulatorn har även ramverket Apache Kafka utvärderats för strömning av stora bilder. Att behandla data i strömmar innebär att datan behandlas alltid som den skapas eller anländer. Apache Kafka utvecklades för strömning av små meddelanden och har ansetts olämpligt för strömning av större filer men är samtidigt ett av de mest använda ramverken för att strömma data. Det är därför av stort intresse att utvärdera om Apache Kafka verkligen är olämpligt eller om det kan hantera denna typ av data.

Projektet har resulterat i ovan nämnda mikroskopsimulator som baseras på 42 000 bilder av celler från AstraZeneca. I simulatorn kan användaren ange vilket tidsintervall bilder ska anlända med, om de ska komprimeras och vilka färgkanaler som ska inkluderas. Simulatorn kan startas via ett webgränssnitt eller via terminalen.

Utvärderingen av Apache Kafka visar att det med rätt inställningar går att strömma stora bildfiler med ramverket även om prestandan inte blir maximal. Slutsatsen från testerna är att det med mer hårdvaruresurser antagligen går att öka överföringshastigheten ytterligare.
1 Introduction

The techniques in microscopy imaging have evolved and microscopy imaging is now a very powerful method for investigating biological processes. Today’s microscopes can generate large amounts of image data in a short time. This makes new demands on data transfer, processing and analysis. Unless all the data can be analysed correctly, important information might be overlooked. The HASTE (Hierarchical Analysis of Temporal and Spatial Image Data) project is a collaborative research project between Uppsala University, AstraZeneca and Vironova which addresses the problem of handling and analysing large amounts of image data. This image data comes from so-called high content screening, a common method to analyse cell activity in both space and time [1]. The proposed approach is to analyse the data in real time and directly make decisions on whether to further investigate the data or not. The solution platform will allow for automation to a great extent and therefore facilitate the study of a wide range of parameters.

To be able to analyse the data in real time the data is processed as data streams. This means that data is processed as soon it arrives/is created instead of letting the processing units wait for all data to arrive and then process it in batches. Data streams are not only used in scientific settings but have many fields of application. Except real time analysis of other kinds of data it is also used when the datasets are too large to process at once, or when the data is produced continuously [2].

The main objective of this thesis was to design and develop a software which can be used to simulate a microscope, specifically the Yokogawa CV7000 microscope used at AstraZeneca. This software can then be used later on in the HASTE project for testing when developing algorithms and designing the whole system. The intention is that the simulator will be both easier and cheaper to run than the actual microscope and therefore facilitate the testing and development process. The simulator also handles the image data as data streams and modifications can be made in the simulator software, which is not the case with the real microscope. These two changes allow for real time processing and optimization of the software. The framework used for handling data streams was the widely used open source framework Apache Kafka [3].

As already mentioned, the data rates and amounts of data are the core problems addressed in this project. The images gathered by the real microscope fulfil the three keystones in the definition of big data. They are created in large volumes, at high velocities and with great variety [4]. To be able to further investigate the HASTE system’s ability to handle big data and high throughput the parameters in the simulator can be set to produce a higher data volume at a higher rate than the actual microscope.

One problem to solve is how to move all the image data from the microscope to the computing facilities, in this thesis Apache Kafka is evaluated for the task. The HASTE system is planned to be a so-called cloud native, i.e. having all its components designed to be deployed as cloud services. Cloud facilities are normally not located near the site where the data is produced. Depending on the implementation and deployment the core of the cloud network might be quite far away, which results in latency. This leads to a situation where it might be more efficient to do some faster calculations at the data acquisition point in a so-called edge environment. However the computer performance will
not be as good as in the core of the cloud network [5]. In the specific case of
the microscope the edge computing means that some kind of computing facility
needs to be installed either on the same computer as the microscope software
or nearby.

Apart from designing and implementing the simulator, the streaming frame-
work Apache Kafka was evaluated for streaming images from one location to
another, for example from the data acquisition to the computing facilities. Eval-
uating Apache Kafka is of great interest for the whole project since it is widely
used but mainly for streaming log messages of smaller sizes [3][6], here it is
evaluated for streaming large images. Can Apache Kafka also handle messages
of larger sizes? And if so, is it suitable for this task? Evaluating Apache Kafka
can help to answer the question in which situations cloud or edge computing
fits best.

2 Background

The HASTE project aims to improve the data handling of large datasets of
spatial and temporal image data. This is to find regions and events of interest
more easily and with higher precision and thereby help researchers to retrieve
the most valuable information from the images. Regions and events of interest
depend on the research questions asked. In the case of a time series experiment
this might be where and when something changes, for example when a cell starts
to produce a certain kind of protein. The goal will be achieved by developing a
smart system ranging from the data acquisition to the final image storage. To
develop this system two main assumptions have been made:

1. Not all images are valuable. Depending on the question the images are
supposed to answer the definition of valuable will differ. Some images
might also be of too low quality or replicates of previous images, which
will make them less valuable. Those which are not valuable cannot be
stored and processed due to time and cost constraints.

2. It is better to analyse the images in real time to make fast decisions
whether to throw away, store or analyse further as early as possible.

These assumptions have lead to a division of the project into three main
components: feature extraction, machine learning to predict the interestingness
and mapping of the data to e-infrastructure. An overview image of the system
can be seen in Figure 1. The feature extraction will be done on either all, or
a subset of all the images acquired. To reduce latency this will be done near
the data acquisition (on fog/edge nodes) if possible, otherwise the data will
be streamed to a cloud environment. The next component is a probabilistic
prediction of interestingness. This is to guide both the data acquisition and
the mapping to the storage. The idea is to use online learning which means
that the models are updated on the fly as new data is gathered. The decision
about which storage facility an image will be mapped into will be based on
the prediction of interestingness together with information about the different
storage alternatives. For example, the most interesting 1% of the images might
directly be further analysed and less interesting images go to storage for batch
processing depending on interestingness. Definitely useless data can be thrown
away [7].
The notion of interestingness depends on the question the experiment is supposed to answer. If it for example is a time series experiment where something is expected to happen images are not interesting if nothing changes compared to previous images, and therefore will not need further analysis. This kind of analysis of streamed time series images can also be used in a non research context connected to for example a surveillance camera. Determining interestingness is therefore an important question in itself and the definition of what is interesting and what is not will depend on the experiments and situations.

In this larger project this thesis fits at the very starting point. The software developed in the thesis acts as a substitute for the microscope and it is also used to evaluate streaming possibilities. The simulator can give answers to how much time the different parts of the system can take without creating bottlenecks in the system.

3 Theory

3.1 Microscope description

There are many different microscopes in use and they all have slightly different features. One of the most used microscopes at AstraZeneca is the Yokogawa CV7000 (Figure 2) therefore this has been used as the main model together with another widely used microscope, ImageXpress (Figure 3).
3.1.1 Overview of Yokogawa CV7000

The Yokogawa CV7000 is a so-called HCS (high content screening) microscope with a wide range of features and settings [8]. A HCS microscope provides automation of the biological investigation in both space and time allowing for fast biological processes to be studied [9]. Typically a HCS microscope can study multiple parameters and visualise quantitative data from the experiments. To do this fluorescence imaging is used which makes it possible to label specific cell components [10].

To optimize the experiments the user can customise a wide range of settings. The settings are both related the environment in the microscope, such as temperature, CO₂ and water concentrations, and related to the image gathering i.e. modes, frequency, binning [8].

3.1.2 Overview of ImageXpress

ImageXpress is another widely used microscope and it has slightly different settings and features compared to Yokogawa CV7000. One of the most important differences is that it has a so-called journal editor which allows for external scripts to change to software. This makes ImageXpress more flexible compared to Yokogawa CV7000 regarding how settings can be made.

3.2 Data streams

Data streams are streams of data which can be processed on the fly. Processing data as streams is the opposite of traditional batch processing, where large amounts of data are first gathered and then processed at once. In data streams each object is instead processed when it arrives or when it is created [13].

Figure 2: The Yokogawa CV7000 microscope which was used as a reference for the microscope simulator [11].
often the objects in a data stream are ordered either by time or some other attribute [14].

Data streams are used in many applications, they are needed when data is produced at high rates. Examples of applications that use data streams are log handling and processing of social media data. With data streams it is possible to get a real time, or at least near real time, handling of the data. Stream processing of data is also needed when it is impossible to wait for all data to be created, either because the data is created continuously and will continue to be created for an "infinite" time period (as for example tweets or sensor data), or because the dataset is too large to be processed at once [13].

In stream processing it is also possible to use the needed data and then throw it away or put it in very cheap and energy efficient storage when the computations are done. In the case of batch processing all this data needs to be available at the same time. In the long term, stream processing allows for both reduced environmental impact (since data storage needs electricity), and also reduced costs (both reduced electricity bills and reduced need for server halls). This is something relevant both in academia with limited means, and in profit driven companies.

Dealing with streams instead of batch data is somewhat different since the complete dataset can not be analysed at once. The fact that each data object arrives at once also puts new requirements on the processing. There are several ways of handling this problem and there are a wide range of frameworks to deal with stream processing [2]. Many companies which are handling large amounts of streamed data have developed their own frameworks for this task. For example Twitter has developed the framework Heron [15], LinkedIn has developed Apache Kafka and Apache Samza. There are also other widely used frameworks such as; Apache Storm, Apache Spark and Apache Flink [16]. Framework specifically developed for streaming of scientific data also exist, one example is HarmonicIO [17]. All of these frameworks have slightly different
use cases and solution methods. In this thesis Apache Kafka, which focuses on message handling, is further studied and evaluated.

3.3 Big data

Big data is one of the prominent trends in both computer science and statistics. By gathering definitions from numerous scientific papers De Mauro et al. have tried to establish the following definition: “Big Data is the Information asset characterised by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value.” [4]. This is not a univocal definition of the concept big data but it captures the ideas that many researchers have expressed.

The three V:s, volume, velocity and variety, are all related to the rate, the speed and how the data is created. Before the big data era, experimental data was created and gathered with great consciousness. This gave both the computer scientists and statisticians advantages when analysing the datasets. Now the opposite is often true, data is generated from multiple differentiated sources, and the generation is often automated meaning that there is a constant inflow of new data. This means that the data might be very heterogeneous, e.g. in the form of text, video, images etcetera [18]. By reason of this, new technologies specific to big data have been developed both regarding processing, storing and transmission of the data [4].

Within the HASTE project the definition suggested by De Mauro et al. captures the challenges regarding the data handling. The image data is high in volume, can be created in a very high velocity, and it shows a variability in terms of what kind of interesting phenomena one might find, what it may look like and where in the images it might be found. This means that new technologies and analytical methods must, and will, be developed during the project, with the main goal to transform the data in the images into value [19].

The main objective with the microscope simulator is to allow for studies of how the algorithms handle large and high-velocity datasets. The parameters which can be changed in the simulator are therefore parameters related to both data rates and the file sizes. It will also be possible to study cases where the simulator produces higher data rates than the microscope to prepare the system for the future. This is interesting to study since both the streaming framework used and the algorithms themselves are affected by the data rates. With this simulator in place a comparison between different streaming frameworks can be done.

This thesis focuses on the streaming framework Apache Kafka but there are other alternatives with slightly different features. One of the main alternatives is HarmonicIO, a framework specifically developed for scientific purposes [17]. The reason for evaluating Apache Kafka is mainly because it is widely used and seen as state-of-the-art for streaming small messages. The simulator can then help to answer the question if Apache Kafka, HarmonicIO, or maybe another framework is better suited for streaming this kind of data when focus is data throughput. The simulator will also be able to help to answer the question of how long time the image analysis and feature extraction can take without creating bottlenecks in the system.
3.4 Cloud and edge computing

The basic idea of cloud computing is to provide a user with on demand access to computer resources such as networks, servers, storage and services. A conceptual image of a computing cloud is shown in Figure 4. Five essential characteristics define cloud computing: on-demand self service, broad network access, resource pooling, rapid elasticity and measured service. These five characteristics result in a service which lets the users manage the resources by themselves and only paying for used resources. This paradigm also allows for large and easy automatic scaling when needed [20].

There are three main service models: software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). The three models have the idea of paying for what you use in common but they act on different levels. Software as a service means that the user can use provided software deployed on a cloud. Platform as a service is when the user can deploy different applications, but the provider controls infrastructure, operating system and storage. Lastly, infrastructure as a service is when only the underlying infrastructure (processing, storage and network) is provided, the user can then control the operating system, storage and applications [20].

There exist four kinds of clouds with different privacy settings, private, community, public and hybrid clouds. A private cloud is a cloud used by only one organization, but it can be managed by a third party and it has not have to be on site. A community cloud is a cloud used by a community of users [20], Swedish National Infrastructure for Computing Science Cloud (SSC) used in this project is an example of a community cloud [21]. A public cloud is open for everyone to use. A hybrid cloud is a combination of any of the above two meaning that the cloud consists of two separate parts which are bound together [20].

Edge computing is when instead of moving data to a centralised cloud the computations are done closer to the data source, at the edge of the network. The need of edge computing emerged from the need to reduce data transfer both to save time and resources. When the data rates and volumes grow, transferring everything to the core of the network becomes infeasible especially if the data is time sensitive. The need for edge computing will increase even more when the number of connected devices increases [5].

Between cloud and edge computing there is also fog computing, or fogging. If edge computing means pushing the compute facilities to the very edge of the network, where the data is acquired, fog computing is pushing the compute facilities to the local area network. A fog computing set-up is therefore equivalent to a small local cloud [23].

In the HASTE project cloud and edge computing are both of large importance. The idea is that the system should be cloud native, i.e. live in and be developed for the cloud. To reduce data transfer some calculations will be performed at the edge of the network, near the data acquisition, in this case the microscope. This will be implemented by either deploying the software on the same machines as the microscopes, or on machines nearby.
3.5 Swedish National Infrastructure for Computing Science Cloud and OpenStack

The cloud used for deployment in this project is the Swedish National Infrastructure for Computing Science Cloud (SSC). The cloud is a community cloud and it can be used by researcher affiliated to a Swedish higher education institution [21]. SSC offers infrastructure as a service and runs the open source cloud operating system OpenStack. SSC is hosted on hardware on three different locations where each location corresponds to an OpenStack region. The different regions are: HPC2N, UPPMAX and C3SE. They all have slightly different hardware and HPC2N is considered to be the fastest and most modern region. The different regions give the user the ability to build more resilient systems since the regions can be used to backup each other [24].

OpenStack is a widely used operating system utilised for both private and public clouds by academia and large companies. OpenStack contains a wide range of APIs and services [25]. The two main abstractions used in this thesis are "instances" and "volumes". An instance is a virtual machine in an OpenStack cloud. A volume is a block storage (a virtual disk), it can be compared to a USB external hard drive which can be attached to one instance at a time. Except from storage volumes can also be used for booting up new instances [26].
3.6 Apache Kafka

The streaming framework used as the main streaming framework is Apache Kafka. The reason for this choice is mainly because it is used by a wide range of companies and organizations such as Mozilla Firefox, Netflix and Spotify [3]. The wide usage is probably because it is one of the most well known frameworks and it provides a reliable and high throughput handling of real time data [27]. It is though not well evaluated for the task of streaming large files.

Kafka is a distributed messaging system developed by LinkedIn to handle their log data with as low latency and high throughput as possible. As shown in Figure 5 the keystones of Kafka are the producers, brokers and consumers, and a key concept is the topic. A topic is a stream of messages and in one Kafka set-up there might be multiple topics. The producer writes messages to a topic, this can be an already existing data stream coming from an external API, or the stream can be created by the producer. After written by the producer the brokers take care of the messages. Together multiple Kafka brokers form a Kafka cluster. The cluster manages the data between the producers and consumers. Lastly the consumers subscribe to one or several topics to receive the messages [28]. The messages can then be processed and analysed in different ways, either by using Kafka Streams API or by using external stream processing engines such as Apache Storm or Apache Flink.

Kafka Streams API is a Kafka library which means that it has no other dependencies than Kafka. The API is built upon Java and is therefore easily implemented in Java applications [29]. There are also several clients making it possible to use Kafka with a wide range of other programming languages such as Python, C, Erlang and .NET [30]. In this project the kafka-python client has been used.

Kafka is a publish - subscribe based message handling framework where messages are written to a log in the order they arrive. Each message has an offset number which makes it possible to get the specific message’s position in the log. The offset number starts at 0 and is increased for each message.

To allow Kafka to handle more data than what fits in one machine it is possible to create so-called partitions, this means that one topic is divided and spread out on a cluster of machines. Each partition must on the other hand fit where it is hosted. The partitioning of the topics makes Kafka scalable regarding total data size [32]. In Figure 6 it is shown how Kafka producers write to a partitioned topic and how a consumer reads from the same topic. The partitions are also the keys to parallelism in Kafka since one partition cannot be split up between multiple consumers. This means that adding more consumers than there are partitions will not affect the processing time. If there are more partitions than consumers some consumers will process messages from multiple partitions. However, one producer can produce messages to multiple partitions. It is also important to mention that the ordering of the messages is guaranteed within each partition, but not between different partitions. The system with partitions is slightly different from most other traditional messaging systems were there usually is one queue per consumer which means that adding a consumer doubles the data size [31].

To make Kafka more resilient each message can be replicated over multiple Kafka brokers. The replication factor is defined when a new topic is created. The replication makes a broker failure less damaging. One broker acts as a
Figure 5: Shows a conceptual image of how Kafka producers, cluster (consisting of one or several brokers) and consumers are related. The producers produce messages, either they create them or they are connected to an API creating messages. The messages are published to the broker and the consumers read the messages from the broker [31].

Figure 6: Shows a conceptual image of how Kafka producers write to a topic and how the consumers read from the same topic. The topic is divided into two partitions. The partitions allow for scaling. If there are multiple machines in the cluster different partitions can be placed on different machines to let the topic be larger than what fits on one machine. If more partitions are added more consumers can read the topic in parallel, only one consumer can read from one topic [31].
leader and handles the partition, the other follows the leader. If the leader broker goes down one of the followers will automatically take the role as the new leader. Each broker within a replicated cluster acts as both leader and follower to spread out the workload.

The messages in a Kafka log are retained for a specified time period. The default value for this is one week. This retention time makes it possible for a consumer to re-consume messages if needed. After the specified time the messages are deleted to free up space [33].

4 Simulator Design

When designing the microscope simulator a number of choices have been made, mainly concerning which features are the most important to recreate in a simulator. Since throughput and latency have been of large interest, settings related to these properties have been favoured.

4.1 Important microscope settings

To get an overview of the microscopes (Yokogawa CV7000 and ImageXpress) and their most important features a study visit at AstraZeneca’s lab in Gothenburg was conducted in the beginning of the project. This included a demonstration of features and settings of the microscopes and their current software.

It would be out of scope for this thesis to simulate all features and settings of the microscopes and therefore a selection had to be made. As a first step, settings concerning throughput were favoured since throughput is assumed to have a large impact on the algorithms which will be tested with the simulator. These settings are frequency, binning and colour channels. A few more settings are described in the following section since the different settings depend on each other.

4.1.1 Frequency

The camera’s frequency is regulated by the user. This frequency can (for Yokogawa) be set to at most 38 frames per second. The rate for ImageXpress is slightly lower [8]. Other settings such as exposure time, binning and autofocus settings have an impact on the frequency. The time period between images cannot be shorter than the exposure time. The same is true for the autofocus setting, the interval cannot be shorter than the time it takes for the autofocus to be set. A higher binning value can decrease the exposure time and thus allowing for a higher frequency. It is important to note that currently the frequency cannot be changed dynamically by the user, but this might change in the future.

4.1.2 Exposure time

The exposure time can be set to 11–9999 milliseconds (Yokogawa) and 70–9999 milliseconds for ImageXpress. To minimise the exposure time the binning can be set to a higher value.
4.1.3 Number of cameras

Yokogawa allows for one to three cameras. This means that up to three images can be taken simultaneously. The microscope currently used at AstraZeneca has two cameras installed. The ImageXpress microscope can on the other hand only have one camera installed.

4.1.4 Time-lapse modes

There are two possible time-lapse modes, burst mode and time-lapse measurement. The burst mode allows for many images to be taken in a short time whereas the time lapse measurement collects data over an extended time period. In a time-lapse measurement different settings can be stacked in a so-called timeline. This timeline can be repeated at a given interval and may include imaging with different settings (objectives, colour channels etcetera), and also burst modes. A time-lapse measurement can gather image data several days [34].

4.1.5 Colour channels

Up to five different so-called channels can be used at once. Each channel can have a different filter which allows for capturing different features. For example one channel could have a green filter, another a red filter and a third one could capture bright field images [8].

4.1.6 Binning

Binning is a technique to compress an image already in the camera. It is a way to reduce the number of pixels and therefore the size of the image. If the binning is set to 2x2 the photons in 2x2 pixels are combined into one pixel. Except from reducing the size of the image this also reduces the noise in the image but lowers the resolution. A visualisation of binning is shown in Figure 7 [35].

In Yokogawa CV7000 the binning can be set to 1x1, 2x2, 3x3 or 4x4. If multiple channels are used the recommendation is to have the same binning for all of them because even though it is possible to have different setting this may cause errors in the result [34]. The binning can be changed over time, and combined with different objectives it can satisfy requirements concerning both speed, magnification and resolution. Even though the binning can be changed over time it cannot be changed dynamically, the binning pattern must be set before the image acquisition is started [36].

4.2 Overall design

The simulator is based on a dataset of about 42,000 images of a time-lapse experiment of cells treated with different medicaments. The images are taken with a Yokogawa CV7000 microscope. This dataset is stored in a so-called volume on a SSC running the open source cloud operating system OpenStack. The basic design of the system is to use the images as a data source and from the images create a data stream. This stream can then either be used locally
Figure 7: Visualisation of how compressing by pixel binning works. The photons in nxn pixels are combined to one pixel, either by mean, maximum, minimum, median or total summation [35].

Figure 8: Visualisation of the overall cloud set-up for Kafka in this project. Machine 1 hosts the simulator and has access to the image dataset. Machine 2 hosts the Zookeeper and Kafka servers. Machines 3 and 4 host the Kafka consumers. All machines are located in the SSC.
or it can be managed and streamed by a streaming framework. The default streaming framework used is Apache Kafka.

To make the simulator more general and facilitate further research on streaming frameworks the simulator is decoupled from the streaming framework settings. This means that the actual microscope simulator produces a stream of images locally with settings defined by the user. If then the user would like to use Kafka for streaming it is possible to do so by specifying a "start-Kafka" parameter.

There are multiple ways to start and manage the simulator. Firstly, it can be started by using a web interface. Following code snippet is used to start the simulator:

```python
python3 simulator.py
```

By entering "127.0.0.1:5000/fileWalk" in the web browser the user will see the interface. This interface is built using the Python micro framework for web applications, Flask [37]. In the interface the user can specify all parameters and start the simulator, with or without Kafka. A screen shot of the interface is shown in Figure 9. Exactly the same thing can be done by importing a Python package and then calling a function with wanted parameters.

```python
import simulator
simulator.get_files(file_path, period, binning, color_channel, connect_kafka)
```

# seconds, color_channel given as list with up to 5
# channels i.e. ["1", "2", "5"], connect_kafka set to "yes"
# for Kafka server connection

Lastly, the simulator can be started by providing all parameters in a JSON-file and then give the JSON-file as an input to a start function. This approach allows for setting parameters for multiple consecutive runs at once.

```python
import profiling
profiling.timer_kafka("file_path_to_images.tif", "to_time", number_of_images)
```

# to_time can be "p": producer, "p2": producer already
# connected to Kafka or "g": simulator
# number_of_images: sets the number of images to stream

```python
profiling.time_kafka_consumer() # time Kafka consumer
profiling.time_kafka_100bytes() # time Kafka producer
```

# when sending small messages

The simulator is implemented in Python and the reason for using Python instead of any other language is that there are many tools for image analysis, machine learning and scientific computing available in Python, and an early decision within the HASTE project was to try to use only one programming language. In contrast to for example Matlab, Python is free to use and Python code can therefore be deployed on multiple virtual machines in the cloud without any difficulty.
4.3 Kafka set-up

Kafka allows for a wide range of customised settings regarding cluster size, message policy, message size, authentication etcetera. The settings are set either in the .properties file (here this is done for the Kafka server), or when a consumer/producer is created by the kafka-python client. In this project most of the default settings have been kept but due to project specifications some have been changed. The most important change is the increased message size (max.partition.fetch.bytes and max.request.size). The default value for both max.partition.fetch.bytes and max.request.size is 1048576 bytes, and the images streamed are up to 10MB (in one of the test cases). max.partition.fetch.bytes is changed by the python client when the consumer is created, and max.request.size is changed when the producer is created. Those values must also match corresponding values in the .properties file, otherwise large files will not be transferred. The settings changed by the client are now set to 10,000,000 bytes and by defining the variable ”max_msg_size” in the python file ”myvariables.py”, the value can thereby easily be changed all over the simulator.

For all topics the replication factor has been set to one, simply because the cluster currently consists of one Kafka server.

4.4 The dataset

The simulator can be used with any dataset of tiff-images, but it is currently running with a dataset of 42,000 images from AstraZeneca. The reason to use a real dataset and not only one representative image, or file of accurate size is that this allows for testing of the image analysis algorithms and the system design in later steps.
The images in the dataset show cells which have been exposed to different kinds of potential medicaments, an example of this is shown in Figure 10. With image analysis the images can be automatically analysed to find out whether or not the medicament work and how the cells react to the treatment.

When evaluating the performance of the simulator, images from the real dataset were compressed and used to simulate images of smaller sizes. To reduce the image size one image was binned and saved multiple times. This image was then copied and datasets of 500 images were created, each dataset had images of one file size. The reason why one image was used and copied instead of using different images was that this approach makes it easier to study throughput as time unit per image since every image in each dataset is of the same size.

To test the strong and weak scaling of Kafka multiple images were added together to create larger files and datasets of 5,000 images were created.

5 Simulator validation

To validate the simulator some tests were conducted and since throughput is of main interest the tests focused on this aspect. The main test was to time the simulator to answer the question of how well the user-set frequency matches the actual frequency, and also to investigate how the file size affects the frequency. Except from investigating the frequency accuracy the maximum frequency was also studied. This was done by setting the time period to zero, i.e. no sleep between each iteration.

The timing was done by using the time.time() function in Python.\footnote{time.time()} is not considered to be the most precise way to time functions in Python, but for the purpose in this case it is considered to be precise enough. According to the Python 

Figure 10: An example of images from the dataset used in the simulator. The images show cells which have been exposed to different kinds of potential medicaments [8].
The function timed read one image at a time and binned each image. This was done on seven different datasets with 500 images each. The different datasets all contained images of different sizes (2.5MB, 1.1MB, 0.38MB, 0.19MB, 0.11MB, 74kB and 53kB).

These tests were done on an instance in the SSC (region HPC2N) running Ubuntu 16 with 4 VCPUs (virtual central processing units) and 8 GB of RAM (random access memory). The images used were stored on a volume attached to the instance.

6 Kafka evaluation

The evaluation of Kafka’s performance when streaming large image files was done in a similar way as the evaluation of the simulator. Multiple set-ups were investigated. The first had the Kafka server instance in one region and the consumer and producer instance in another region. This set-up can therefore be seen as having the computing facilities in the cloud and not at the edge of the network. The set-up was as following:

- Test dataset stored on volume connected to an instance running Ubuntu16 in the SNIC cloud (HPC2N)
  - RAM: 8GB
  - VCPUs: 4 VCPU
  - Disk: 80GB
- Kafka and Zookeeper servers running on Ubuntu16 instance in SSC (C3SE)
  - RAM: 16GB
  - VCPUs: 8 VCPU
  - Disk: 160GB

This first test evaluated the performance of the producer i.e. how fast the producer writes to a Kafka broker. As with the test of the simulator the timing was done by using the Python function time.time() before and after sending messages to Kafka.

```python
start = time.time()
function.to_time()
stop = time.time()
```

One drawback to be aware of with time.time() is that it might be affected by other programs running or even the garbage collector. It is also important to notice that time.time() behaves differently on windows and Unix systems. In Unix based systems the function returns the processor time as a floating point number where the precision depends on a C function. In Windows systems the function returns the number of seconds since the very first call of the function, here the resolution is normally better than one microsecond [38].
It is important to notice that the conversion from tiff to bytes is not included in the timing nor is the set-up of the connection to Kafka. The connection to the Kafka server is done only once and therefore considered not necessary to include in the timing.

The same test was run both in a set-up where both the producer/consumer machine and the server were deployed in the same region (HPC2N) and in a set-up where the producer/consumer machine was deployed in one region (HPC2N) and the server machine was deployed in another region (C3SE). These settings can be compared to the case of having all computing done in the cloud and a cloud/edge set-up where faster calculations are done near the data acquisition, and the data then sent to the core of the cloud for further analysis. The bandwidths between the instances in different regions and the bandwidths between the instances in the same region were tested with the tool iperf. The average bandwidth between the instances in region HPC2N and C3SE was 86.1 Mbits/second and the bandwidth within the HPC2N region was 4 Gbits/second.

Also, strong and weak scaling of the system was tested. Strong scaling is when the system is tested with a fixed size dataset. In this case datasets containing 5,000 images. The number of processing units is then increased and the processing time is supposed to decrease. Weak scaling is when the dataset’s size is increased linearly with the number of processing units. In this case the number of images increased with 1,000 for every five processing units (consumers) added. An optimal outcome of a weak scaling would be no increase in processing time.

The set-up was Kafka and Zookeeper servers running on an instance on the SSC region HPC2N. The server instance had Ubuntu 16 on 8 virtual CPUs and 16GB RAM. There was also one instance running the producer. This instance was also running Ubuntu 16 but had 4 virtual CPUs and 8GB RAM. The last two instances were the consumer instances which had the same set-up as the producer instance. Both the producer and consumer instances were in the same region as the server, HPC2N. The bandwidth between the instances was tested using the tool iperf, the Kafka/Zookeeper server was set as the server and the consumer and producer were set as clients. The results from iperf showed an average bandwidth of 4 Gbits/second. This sets an upper limit for the performance when transferring images between the instances.

The strong scaling was done on three different datasets each containing 5,000 images. The three sets had images of the sizes 2.5MB, 4.9 MB and 9.8MB. The number of consumers was increased and the total processing time was studied. The producer retrieved images from a volume attached to the instance and formatted the images from tiff to bytes. The consumer on the other hand only received the images in bytes form and no processing was done. The scaling was done by adding more and more consumers, five at a time. The consumers were run in parallel using the Python package "multiprocessing". Up to 15 consumers were run on each consumer instance before another instance was added. It is important to notice that since the producer side was not parallelized the producer produced all messages to the server before the consumers were started. This was to avoid having the producer as a bottleneck when timing the consumers’ performance. The number of images each consumer received was also counted. This was to study how well Kafka did the partitioning of the topic.

The weak scaling was done in a similar way. The same producer, server and
consumer instances were used. The scaling was also this time done on images of the sizes 2.5MB, 4.9MB and 9.8MB. The producer produced images to the broker and when all images were produced the consumer was started, this was to avoid the producer as a bottleneck. The datasets used was the same as in the strong scaling but this time the number of images depended on the number of consumers. The number of images was increased linearly with 1,000 for every fifth consumer.

7 Result

7.1 Results for the simulator

The simulator itself was tested both to investigate the maximal speed when producing messages but also to test if the set frequency was accurate. In Figure 11 one can see that the processing time increases as the file size increases. These results are setting an upper limit for the simulator if it is used without any parallelization, it will not produce images at a higher speed than this. What is also interesting is how accurate the frequency setting is, this is shown in Figures 12 and 13. The figures summarise multiple iterations with images of different sizes and different frequency settings. What can be seen is that for longer time periods (lower frequencies) the simulator performs better, i.e. closer to the intended time period. It can also be seen that the standard deviation is quite low throughout all frequencies and image sizes, even though the relative error \( \frac{T_{\text{expected}} - T_{\text{input}}}{T_{\text{input}}} \) is high. This means that even though the actual frequency is lower than the intended frequency this is something that can be fixed rather easily by compensating for the overhead for reading and processing the images.

7.2 Kafka results

7.2.1 Producer performance

The results from the producer tests are shown in Figures 14 and 15. The tests are conducted in the same ways in both figures but Figure 14 shows the case where both the producer and server are located in the same region, meaning the same geographic location and within the same local network. In Figure 15 the server and producer are located in different regions and therefore they cannot connect via a local network. When comparing the two figures it can be seen that the case with the producer and server in the same region outperforms having the producer and server in different regions. It is therefore reasonable to conclude that edge or fog computing would be favourable if possible.

7.2.2 Strong Scaling

The results from the strong scaling of the Kafka consumers can be seen in Figure 16. The maximum throughput was 328.9MB/second for 2.5MB sized images, 335.6MB/second for 4.9MB sized images and 374.0MB/second for 9.8MB sized images. This can be compared with the average bandwidth which was 4Gbit-s/second (0.5GB/second). The time performance reaches a saturated level when ten consumers are working in parallel.
Figure 11: Results from running the simulator at maximal frequency without streaming with Kafka. Included in the processing is retrieving the images from the storage and converting them from tiff to bytes. The datasets used consist of 500 images each, each with different image sizes; 2.5MB, 1.1MB, 0.38MB, 0.19MB, 0.11MB, 74kB and 53kB. Each data point is plotted with its standard deviation.

Figure 12: Results from running the simulator without streaming with Kafka, when the time period between each iteration is specified. Included in the processing is retrieving the images from the storage and converting them from tiff to bytes. The same datasets as in Figure 11 are used (each consisting of 500 images with the images sizes; 2.5MB, 1.1MB, 0.38MB, 0.19MB, 0.11MB, 74kB and 53kB).
Figure 13: Results from running the simulator without streaming with Kafka, here the time period between each iteration is specified. Included in the processing is retrieving the images from the storage and converting them from tiff to bytes. The $y$-axis shows the relative difference between the input and output time periods, $\frac{T_{\text{output}} - T_{\text{input}}}{T_{\text{input}}}$. 

Figure 14: Results from testing the Kafka producer. Results from running the simulator at maximal frequency when streaming with Kafka. Here no processing is done on the images, only streaming. The datasets used consist of 500 images each, each with different image sizes: 2.5MB, 1.1MB, 0.38MB, 0.19MB, 0.11MB, 74kB and 53kB. Each data point is plotted with its standard deviation. Both the server and producer are running in the same region, HPC2N.
Figure 15: Results from running the simulator at maximal frequency when streaming with Kafka. Here no processing is done on the images, only streaming. The datasets used consist of 500 images each, each with different image sizes; 2.5MB, 1.1MB, 0.38MB, 0.19MB, 0.11MB, 74kB and 53kB. Each data point is plotted with its standard deviation. The server and producer are running in different regions, C3SE and HPC2N.

Figure 16: Results from strong scaling of the Kafka consumers. Datasets containing 5,000 images each of the sizes 2.5MB, 4.9MB and 9.8MB were studied. 1, 5, 10, 15, 20 and 30 consumers were run in parallel on up to two machines. The maximal throughput was 328.9MB/second (2.5MB sized images), 335.6MB/second (4.9MB sized images) and 374.0MB/second (9.8MB sized images).
The number of messages in each partition/consumer when testing on 5, 10, 15, 20, 25 and 30 partitions/consumers. This is how Kafka scales, by dividing messages between different partitions.

The partitioning of the images between the different consumers was also studied. This is something that is done by Kafka automatically when messages are produced. The number of messages each consumer received can be seen in Figure 17.

7.2.3 Weak scaling

In Figure 18 the results from the weak scaling are presented. The results shows weak scaling for datasets with images of 2.5MB, 4.9MB and 9.8MB. It can be seen that the scaling differs between the three datasets and the scaling is better for the dataset with smaller images.

8 Discussion

From the results of the simulator benchmarks seen in Figure 11 one can conclude that the simulator can produce up to approximately 40 images per second. This result is on the cloud and hardware used in this project, depending on the resources this might vary. Compared to the Yokogawa CV7000 microscope, which can produce up to 38 images per second, this is slightly faster.

The strong and weak scaling focussed on testing Kafka for throughput. In Figure 16 it is possible to see how the Kafka consumers scale. From the figure one can draw the conclusion that the performance saturates after about ten consumers. It is important to notice that up to 15 consumers are running in parallel on one instance with limited hardware resources. It is therefore reasonable to believe that changing the experiment by adding more instances to run on, and having less consumers on each instance would improve the performance. The
Kafka server also has limited hardware resources and by adding more brokers on different machines the performance would probably also improve. Since part of the processing time goes to size independent tasks it is not surprising that larger file sizes show better performance when the total throughput in bytes is considered.

The weak scaling in Figure 18 shows how the Kafka consumers scale when more workload is added in the same rate as more consumers. If the scaling were perfect the processing time would not increase since each consumer is supposed to have as many images to process in each try. As for the strong scaling the results are affected by the fact that up to 15 consumers are running in parallel on each instance, more than there are CPUs. Another reason for the results of the weak scaling might be the rather unusual datasets. Kafka was built for small log messages (up to 10kB) and here message up to almost 10MB are streamed.

The simulator has room for many improvements which have been out of scope for this thesis. One important feature is to be able to change the streaming settings while the simulator is streaming without first stopping and then restarting the stream. This feature is important since some settings already now can be changed dynamically in the real microscopes, and others might get this feature in a near future. This dynamic change of simulator/microscope settings allow for a feed-back loop depending on interestingness in the images. Another improvement would be to add more settings to the simulator, and also coordinate the settings to better mimic the microscope. This would for example be making the frequency depend on the binning and exposure time.

Also the Kafka implementation leaves room for improvements. The implementation is now based on the client python-kafka. python-kafka is not the only client to connect python and Apache Kafka and other clients might be faster, or have other benefits compared to this one. The benchmarks were also conducted

Figure 18: Results from weak scaling of Kafka. 1,000 images of the size 2.5MB, 4.9MB and 9.8MB was streamed for every five consumers. 5, 10, 15, 20 and 30 consumers were run in parallel on up to two machines.
in a specific setting with only one broker in the Kafka cluster, this is normally not the case in real production. Suggestions for this would be to benchmark Kafka in a larger cluster on multiple machines. This would probably improve the results at the consumer side since more machines would not only add more CPU and power but also more bandwidth in the connections from the cluster to the consumers. The same would be true for the producers if they can be implemented to run in parallel. In an HASTE specific setting this could be implemented in a way where different producers produce images from different colour channels.

9 Conclusion

The goal of this thesis was to design and develop a microscope simulator to be used within the HASTE research project. As it is now the simulator is connected with two different streaming frameworks, the widely used Apache Kafka and the newly developed scientific framework HarmonicIO. The user can from both an interface and the command line start and stop the simulator, and at the same time choose which streaming framework to use. The simulator allows the user to set the frequency, binning and colour channels. The simulator is deployed in SSC to be a part of the complete HASTE system. The code can be found on Github using the address https://github.com/LovisaLugnegard/exjobb.

The streaming framework Apache Kafka has also been evaluated for streaming images. The conclusion is that Apache Kafka is able to stream large image files but the performance can probably be improved by adding more machines to the cluster thus adding more hardware resources and bandwidth. By testing different set-ups it can be concluded that edge or fog computing, i.e. having the computing facilities close to the data acquisition point, is favourable in the case of faster computations.

10 Acknowledgements

Firstly I would like to thank Andreas Hellander for supervising and helping me during this project. I would also like to thank my subject reviewer Carolina Wahlby for suggesting this project from the very start and helping me along the way. A special thank you to Alan Sabrish and Johan Karlsson at AstraZeneca for the visits at AstraZeneca’s lab in Gothenburg. I would also like to thank Salman Toor for the help during the project, and Ben Blamey for helping me writing better code. Also, thanks to the entire HASTE team for the support and great discussions.
References


[38] time | Time access and conversions; Cited: 11/12/2017. Website. Available from: https://docs.python.org/3.5/library/time.html#time.clock.