Synchronising Distributed Scraping

Author: Godfried Meesters
Student ID: 850021224

Presentation Date: August 27, 2021

Course code:
IM9906

Open Universiteit
www.ou.nl

Open University of the Netherlands
Faculty of Science, Master's Programme in Software Engineering

Synchronising Distributed Scraping

Author:
Godfried Meesters
850021224

Chairperson:
dr. ir. Harald Vranken

Supervising team:
dr. ir. Hugo Jonker
Benjamin Krumnow, MSc.

Presentation Date: August 27, 2021

Course code:
IM9906

Open Universiteit
www.ou.nl

Abstract

Price differentiation refers to a commercial strategy of charging different prices for the same product or service. A given e-commerce company can offer the same items through multiple outlets, such as a website or a mobile application. There have been rumors that there are price differences between equivalent items offered on different outlets. We would like to verify these rumors.

To assist in comparing outlets, data needs to be collected on a large scale simultaneously. Manual data collection can be used, however the amount of data that can be collected manually is limited. Another problem with manual extraction is that equivalent items from different outlets have likely not been extracted at exactly the same time.

In this study, a distributed and synchronized web scraping system is designed. An unlimited number of web bots taking jobs in a pub/sub system can be accommodated that synchronize to each other. To validate the design, an experiment with price differentiation in the travel industry is conducted with a focus on flight ticket prices.

Contents

1 Introduction **3**

2 Related work **7**
- 2.1 Price differentiation on desktop browsers . . . 7
- 2.2 Price differentiation on mobile devices . . . 8
- 2.3 Mobile Device Fingerprinting . . . 9
- 2.4 Web Scraping . . . 9
- 2.5 Web Crawling . . . 10
- 2.6 Synchronization . . . 10

3 Methodology **12**
- 3.1 Reasons to run an automated experiment . . . 12
- 3.2 Limitations of running an automated experiment . . . 12
- 3.3 Scraping Techniques . . . 13
  - 3.3.1 Scraping from mobile applications . . . 13
  - 3.3.2 Scraping from websites . . . 17
- 3.4 Equality of products . . . 18

4 Design **19**
- 4.1 Framework Requirements . . . 19
- 4.2 Architecture Design . . . 20
  - 4.2.1 Initial Design . . . 20
  - 4.2.2 Shortcomings of Initial Design . . . 22
  - 4.2.3 Distributed Scraping . . . 22
  - 4.2.4 Synchronization of Distributed Scraping . . . 26
  - 4.2.5 A Distributed and Synchronized Architecture . . . 29
- 4.3 From Architecture to Implementation . . . 33

5 Validation by means of a price comparison study **36**
- 5.1 Experiment Design . . . 36
- 5.2 Experiment Results . . . 37
- 5.3 Analysis . . . 40
  - 5.3.1 Air France . . . 41
  - 5.3.2 EuroWings . . . 44
  - 5.3.3 Opodo . . . 44
  - 5.3.4 Expedia . . . 46
  - 5.3.5 Kayak . . . 49
  - 5.3.6 Booking.com . . . 50

6 Conclusion **51**
6.1 Future work 51

Appendices **56**

A Implementation **57**
A.1 Analyst's Guide 57
A.2 Developer's Guide 59
A.2.1 Interaction via Queues 59
A.2.2 Bot Implementation 61
A.2.3 Controller Implementation 64

Chapter 1

Introduction

Imagine you and a friend want to go on holiday to Stockholm. You meet at his place to find flights and buy the tickets. Your friend finds the cheapest flight on his laptop: 150 EUR. He tells you the flight details and the website he used, and then waits so you can order the same ticket on your phone. You visit the very same site on your phone and find the flight, but the price offered to you is 190 EUR. Both you and your friend feel cheated, but since this is the cheapest flight, you both book your tickets anyway – yours being 40 EUR more expensive than your friend's ticket.

What happened here might have been price differentiation. Price differentiation [HTWH18] describes a marketing strategy to determine the price of goods on the basis of a potential customer's attributes like location, financial status, possessions, or behavior. Figures 1.1 and 1.2 show that the flight price on the mobile application of Air France is EUR 10 higher than for the same flight found on Air France's website. This data was collected by manually launching a query for flights from Brussels to Paris leaving on November 30, on the desktop website. The same query was launched manually, less then one minute later, on the mobile application of Air France, from the same IP address.

Price differences were also found on Opodo (an aggregator that compares flight ticket prices for several airlines). Manual queries revealed that Opodo's mobile application prices were higher for the same flight ticket than on its website.

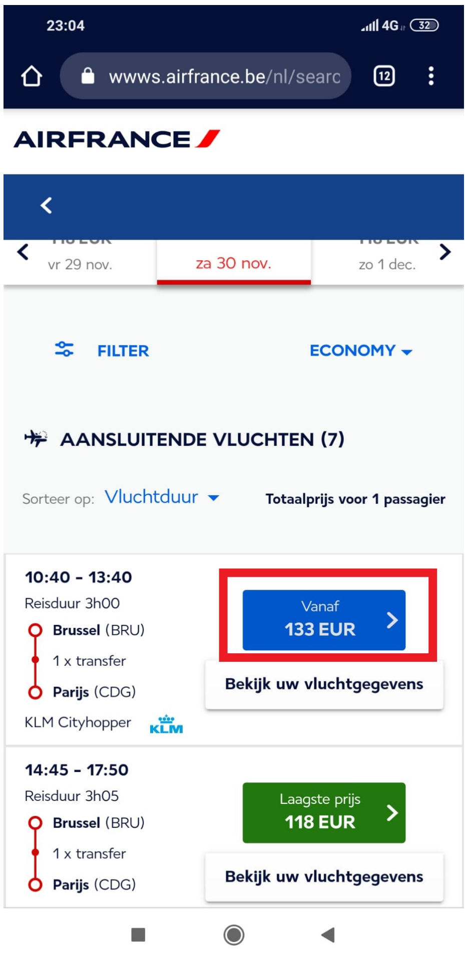
Is this a real price difference, or might the price difference come from the fact that we did not check on the website and the application at the same time? Or are there other confounding factors?

Such observations warrant further investigation. There are anecdotal reports [1] about price differentiation in e-commerce and a study by Bertsch et al. [BMW17] that found prices on an airline's Android application to be up to 10% higher than prices for the same flight tickets on its website (accessed from a mobile phone). The study by Bertsch et al. however missed information about the circumstances in which prices were extracted, for example: what was the time difference between extractions?

We would like to do a structural investigation of price differentiation, by creating a framework that allows automating comparisons of different outlets against each other, including comparison of mobile applications and websites.

Doing such a comparison study and creating a supporting framework comes with several design decisions. It is important to look more than once to see what price is on offer, since missing a price update might mean that we miss a price difference. At one given time there may be no price differences at all, while another time a price difference is found; if we only test once a day we might wrongly deduce that there are no price differences.

[1] https://blog.blackcurve.com/price-discrimination-is-more-common-than-you-think


23:04 4G 32

wwws.airfrance.be/nl/searc 12

AIRFRANCE

vr 29 nov. | za 30 nov. | zo 1 dec.

FILTER ECONOMY

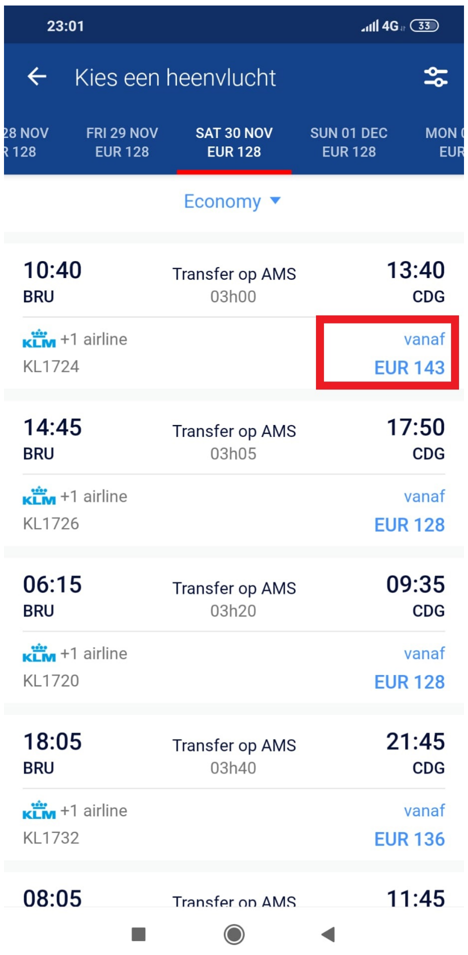
AANSLUITENDE VLUCHTEN (7)

Sorteer op: Vluchtduur Totaalprijs voor 1 passagier

10:40 - 13:40
Reisduur 3h00
Brussel (BRU)
1 x transfer
Parijs (CDG)
KLM Cityhopper
Vanaf 133 EUR
Bekijk uw vluchtgegevens

14:45 - 17:50
Reisduur 3h05
Brussel (BRU)
1 x transfer
Parijs (CDG)
Laagste prijs 118 EUR
Bekijk uw vluchtgegevens

Figure 1.1: Flight price found on Air France website


23:01 4G 33

Kies een heenvlucht

28 NOV EUR 128 | FRI 29 NOV EUR 128 | SAT 30 NOV EUR 128 | SUN 01 DEC EUR 128 | MON EUR

Economy

10:40 BRU — Transfer op AMS 03h00 — 13:40 CDG
KLM +1 airline KL1724 — vanaf EUR 143

14:45 BRU — Transfer op AMS 03h05 — 17:50 CDG
KLM +1 airline KL1726 — vanaf EUR 128

06:15 BRU — Transfer op AMS 03h20 — 09:35 CDG
KLM +1 airline KL1720 — vanaf EUR 128

18:05 BRU — Transfer op AMS 03h40 — 21:45 CDG
KLM +1 airline KL1732 — vanaf EUR 136

08:05 — Transfer op AMS — 11:45

Figure 1.2: Flight price found on Air France mobile application

If we had to test only once a day, manually checking prices without a framework would be feasible. However, to check prices with high granularity for multiple companies with multiple outlets (e.g. 10 companies * 2 outlets * 4 extractions per day), using a framework would be more efficient even though such a framework takes a considerable amount of time to create and also maintain.

Another consideration is, when comparing prices manually, querying prices on one outlet may not have been done at the same time as the other outlets. A price difference found might be the result of a price update that happened on all outlets at the same time, however this update was missed because the price from one outlet was extracted at a different time than the price on the other outlets. With automatic extraction instead of manual extraction, it is possible to extract prices from all outlets at (almost) the same time.

The main research question is: how to compare multiple outlets simultaneously?

In designing a supporting software architecture, we need to consider confounding factors.

From literature, several factors are known to possibly influence pricing. These include the location from which a visitor is accessing an outlet; for example it was found that changing the GPS location for the mobile 'Uber' application[CMW15] resulted in different prices. Another influencing factor is web browser state; e.g. a customer bought an expensive iPhone 4 on a website, after which a cookie is saved that marks the customer an 'affluent customer', resulting in higher prices for subsequent purchases.

A factor that gets special consideration our study is timing. For example, the prices extracted from Air France are not extracted at exactly the same time; between the moment that the price was extracted from the application and the website, the price may have been updated. This brings us to the question: How can we rule out 'false positives' because extractions were not done at exactly the same time?

How can extractions be synchronized? An additional requirement is that more than two types of outlets should be possible to compare, and web bots extracting data from every outlet should be able to run on a cluster of machines. Therefore, distribution must also be taken into account.

The main contribution of this study is the creation of a synchronized, distributed scraping system to compare multiple outlets:

- Our system is based on the principle of comparisons. In each comparison, two or more outlets are compared to each other. This means that to execute a comparison, two or more scraping bots run simultaneously, starting from a common search query. Each bot will then return offers that match the given search query.

- Workload of a comparison can be distributed. Scraping bots can run on multiple machines and work together on the same comparison.

- Scraping bots that work on the same comparison, can be synchronized before and after certain phases of the data collection. Bots will wait for each other until a certain point in the web scraping process has been reached. The design goal of the system is to extract prices from different outlets at the same time.

After the creation of a synchronized, distributed scraping system, the system is validated by means of a price comparison study.

A question one might ask is: what is the difference between what we are doing and what commercial shopping comparison tools have been doing for a long time?

Shopping comparison tools collect data provided by sellers, and allow a user to compare prices for a given product or service.

Tools used in academic price differentiation studies do not depend on data provided by sellers; academic studies attempt to appear as a real visitor, therefore data is collected manually or by means of a web bot.

In price differentiation studies, data can be extracted from different outlets. An outlet can be defined as any channel through which products or services can be offered.

Several price differentiation studies use web bots that visit a given outlet with specific parameters (e.g. with a specific Geo-location) and compare prices to the same outlet visited with different parameters. Differentiation studies comparing multiple different types of outlets are rare, even though finding differences between multiple different types of outlets is an interesting research topic.

Comparing between different types of outlets poses extra challenges. For example, how can two outlets be compared, when one outlet is a smartphone application and another outlet is a desktop website? What if scraping bots are distributed over several machines? How do scraping bots receive jobs and how is data collected? Besides, when multiple outlets are compared, it is important that comparisons are synchronized. For data changing with a high frequency (e.g. flight tickets), the moment that data is extracted from one outlet, needs to be as close as possible to the moment data is extracted from other outlets. This is not trivial, especially when comparing many outlets.

On what point in the web scraping process does synchronization need to happen? And what happens when one scraping bot is slower in execution than other bots? Another layer of complexity to synchronization of web bots is added, when bots are distributed over multiple machines. Synchronization primitives from multi-threaded programming, such as barriers and phases, could be transferred to distributed environments. However, multi-threaded programs on a single machine do not need to take into account uncertainties such as network connectivity between machines or differences in performance between individual machines. In addition, scraping bots extract data from outlets that are prone to crashing (such as apps on mobile devices) or bots continue running forever, potentially resulting in a deadlock when a bot never reports its status as 'finished'. There are several solutions to prevent deadlocks. For example, each bot can be assigned a time limit before which web scraping has to be finished. At the same time however, the time limit has to be small enough for the data not to become stale.

When a bot finishes successfully, it will return items which are stored in a central database. Any type of data can be stored, including but not limited to flight ticket offers. A problem when attempting to uncover price differentiation, is that one has to be compare offers that are equal. For example, imagine a flight offer that is cheaper than another flight offer leaving at the same departure time, and leaving from the same departure city. Upon inspection, however, the cheaper flight involves a train ride. Can these offers be considered equal? Since equality of items is dependent on the type of items being compared, it is up to the analyst to determine which items are equal; sample definitions of equality are provided in section 3.4.

Chapter 2

Related work

The topic of this thesis, synchronizing distributed web scraping with the aim of uncovering price differentiation, touches upon a broad range of subjects.

Several studies of online price differentiation have been done before that focus on uncovering price differentiation based on different desktop browser fingerprints. These studies are discussed on section 3.1.

A smaller number of studies have been done on price differentiation from different mobile application fingerprints; mobile device fingerprinting could also potentially be used in price differentiation.

Articles on web scraping techniques have also been included; a problem for example that is also relevant to our research, is the fact that outlets are continuously evolving and selectors need to be updated accordingly.

To design our distributed web scraping system, articles on distributed web crawling are also included since the techniques discussed to distribute web crawling work can also be used in distributing web scraping work over several machines.

The last section in this chapter discusses related work in synchronization. We want to run multiple scraping bots concurrently and synchronize them between different phases; in related work synchronization barriers are used to implement such a requirement.

2.1 Price differentiation on desktop browsers

Hannak et al. [HSL+14] investigated price steering and price differentiation on several e-commerce websites based on browser fingerprinting. Price steering is defined as directing users to goods with certain prices, while price differentiation is defined as serving different prices for the same good to different fingerprints. For our study, a distinction is made between price steering and price differentiation up to a certain level; a search query will be launched for a given product, and the cheapest price will be extracted from the results page. For example, if an Airline company is displaying only flight tickets of type ”flex” on its mobile application, and showing both ”flex” and cheaper ”semi-flex” type flight tickets on its website, then the website will be considered cheaper for the same product.

Hannak et al. [HSL+14] conducted a study with real-world users, to detect price differentiation of 16 e-commerce websites. The study attempts to find out whether past user behavior (did the user make a previous purchase?) is influencing future pricing.

Real-world users were asked to configure a browser proxy controlled by Hannak et al. When visiting a web shop, the proxy fired off two requests, one with and one without user data. In addition, a “comparison” and “control” request is executed , to account for factors like A/B treatment, sudden updates or price differences among data centers.

In some cases, the price was lower when no user data was submitted, while the same good was more expensive when the e-commerce provider had access to past user behavior.

Hupperich et al. [HTWH18] created an automated price scanner, scraping prices from hotel and rental car providers using real-world browser fingerprints. They did find systematic price differentiation, when changing the location.

Mikians et al. [MGEL13] use a distributed approach to scrape e-commerce prices by letting internet users install a browser extension called "$heriff". They investigated price differences based on device fingerprint, location and personal information. Users are categorized as 'affluent' or 'budget minded' based on their browsing and/or purchasing history. There are indications that affluent customers are shown higher prices than budget minded users, and that prices differ among the locations where the price request is coming from. In addition, prices differ when a price search is originating from a price "aggregator" (a portal where prices are compared). Noise factors like currency, shipping and taxation differences were taken into account, which will also be taken into account for our study.

Vissers et al. [VNBJ14] researched flight price differentiation of online airline tickets, based on Browser/OS profiles, personal profiles (users are categorized as budget minded, affluent and flight com-parer; cookies are set accordingly and are loaded from so called "cookie jars" that include first and third party cookies). Also the influence of location on price was tested. After a three-week long experiment, spanning 25 different airlines, retrieving data from two geographical locations, no *consistent* price differentiation could be found. However, between individual user profiles, price differences were found; these differences were attributed to flight prices abruptly changing, while measurements between profiles were 'roughly one minute apart'. Price differences were also found when accessing the Argentinian website of American Airlines; the same flight on the Argentinian website of American Airlines was consistently higher than on the American website, which could be explained by an extra tax from the Argentinian government.

2.2 Price differentiation on mobile devices

Most research articles found on price differentiation only consider price differentiation within the same outlet, accessed from different visitor profiles. There are few research articles about price differentiation between native mobile applications and (mobile) web browsers.

Bertsch et al. [BMW17] researched flight ticket price differentiation on mobile browsers versus native Android applications. Flight price searches are executed on native Android applications and on mobile web browsers, both running inside an Android device emulator. Consistently higher prices were found for the same flight ticket on mobile applications. The interesting part of Bertsch' research is the use of a mobile application testing framework that is used to extract flight prices from different airline applications. With the testing framework 'Appium', all actions that a real user can do on a mobile application can be simulated using the WebDriver protocol. Appium also allows for setting different properties of the device fingerprint dynamically. And unlike other mobile application testing frameworks, Appium is a testing framework that allows for a black-box approach, meaning that applications can be tested as-is, without modifying the APK.

Le Chen et al. [CMW15] researched pricing algorithms when using the mobile application of Uber. Unlike Bertsch et al, the authors did not use black-box testing framework, but instead were able to intercept and analyse network traffic generated by the Uber application, resulting in a list of API signatures that could be used to call the Uber back-end directly. By sending API calls to the Uber back-end with spoofed GPS coordinates, a surge price algorithm is observed. A geological region is divided into different sectors, each with their own pricing according to supply and demand. It was found out that by moving the GPS position by only a few meters, customers could save 50% or more. For our study, directly calling API endpoints instead of using mobile device emulators would require less hardware resources than running Appium and an Android emulator. However, in Le chen's study only one mobile application was under consideration, while in our study we would have to extract API endpoints for 10 mobile applications.

2.3 Mobile Device Fingerprinting

Yang et al. [YY20] researched fingerprinting, i.e. uniquely detecting website visitors not based on cookies, but based on different characteristics such as screen size that can be revealed by JavaScript API's. Yang et al. compared if websites served on desktop devices but with the web browser user agent spoofed as a smartphone, are different from the same websites served on "real" mobile browsers on a real mobile device. More specifically, they attempted to detect tracking frameworks. It was determined that websites displayed on mobile devices sometimes employ tracking frameworks specific to mobile devices. For example, some mobile websites embed tracking frameworks that poll the mobile device's accelerometer.

2.4 Web Scraping

Thomsen et.al. [TEBS12] research the problem in web scraping that also took the majority of work in our research: web content that is continually changing. When a website changes, web element selectors may stop working. The authors designed a state diagram that updates web element selectors semi-automatically when website content is changing. The state diagram is based on selection functions and validation functions. Selection functions (for example XPATH) are preferably as generic as possible to capture selections even after a website update. Selections are passed on to validation functions (e.g. is the selection found by the selection function textual?). The state diagram begins with the state 'evaluate selection function' and ends with the final state 'accept' or 'abort'. When a selection function does not return anything, or when a selection function returns a selection, but the selection does not pass validation, a process of 're-induce selection function' is started to update the selection function. Validation functions can also be 're-induced'. 'Re-inducement' can be automatic, for example by taking outputs from selection functions and updating the validation function so that all selection function outputs pass. Re-inducement can also require user intervention. For our study, there are some mechanisms in place to 'survive' website updates and update selectors; more on that in the Design chapter.

To increase scraping performance Uzun [Uzu20] created a method to extract information from websites that does not depend on the DOM. Using a string extraction based method (e.g with regex) as opposed to creating a DOM tree and selecting a DOM element is found to be up to 60x faster. With additional information such as the starting position of an element, string based extraction of an element can be increased again by about a factor of 2.35. The problem of considering a web page as one static string, is that this extraction method may not extract dynamic content generated by client side scripting, which is a requirement in all websites considered by our study.

Related to our study of finding price differentiation, Oancea et. al. [ON19] attempted to create the Consumer Price Index for Romania, with data extracted using a web bot, scraping products from e-commerce websites that are found in the country's official CPI. Problems encountered included offline websites and updated websites, and bot detection.

The authors observed price differentiation due to geographical location. After data extraction and cleaning, similar products were clustered using distance based methods (matching strings with Levenshtein gave the best results). Our study also has to match products, however this is done differently (see Methodology chapter).

2.5 Web Crawling

Related to this study is the design of web crawlers. Web crawlers start with a seed URL and crawl up till a certain depth, and distribute crawling jobs over many machines to increase performance. Our system does not support crawling in depth, but crawling horizontally (e.g. navigating through a product list with multiple pages) is supported. Our system supports distributing scraping jobs over multiple machines, not with the purpose to increase performance, but with the purpose to execute scraping jobs as simultaneously as possible.

Ye et. al. [YJHC18] discuss the design of a lightweight distributed web crawler based on Scrapy-Redis[1]. Requests to be processed by Scrapy crawlers distributed over several machines are put on a central Redis 'requests queue' and items generated by crawlers are put on a central Redis 'item queue'. Crawling tasks are created by a scheduler, or can also be created manually by a user through a 'crawling management component' (which also includes ongoing tasks and crawler status monitoring). See Design chapter on how queues are used in our study to distribute scraping jobs.

In another study, Xie et. al. [XWGY19] attempted to classify countries all around the world according to political, economical, financial, business environment and legal risk. To do this, they created a distributed web crawler that crawls for example the website of the United Nations for indicators ('Satisfaction to government', 'Trust in government', . . .) of political risk. The distribution of crawler tasks is based on Hadoop MapReduce[2]. Hadoop MapReduce is a framework for 'parallel processing of large datasets', where the workload can be distributed among many machines. In our study, for the distribution of web scraping jobs, Kubernetes is chosen instead of Hadoop MapReduce, since the author is more familiar with Kubernetes. In this study, Singapore wins the highest credit rating.

Kc et. al. [KHT08] create a distributed crawler with one master machine and several 'slave' machines. To minimize network traffic, instead of sending every url to be retrieved from the master to a selected slave, the master sends a 'seed URL' to the crawler machine geographically closest to where the web server to be crawled is located. The crawled data is compressed and sent back to the master machine. In addition to data compression, crawlers support loop detection (preventing crawlers from running infinitely), state safety (allowing crawlers to resume operations) and recovery from situations where the connection with the master machine is broken. Crawlers are 'polite', respecting 'robots.txt' and pausing between requests to the same web server.

2.6 Synchronization

A requirement of this study is to create a distributed web scraping framework that allows bots that may be distributed among several machines, to synchronize to each other. Scrapers 'synchronize on search', meaning that all bots are stopped just before 'clicking the search' button and then have to wait for each other.

In 'A new Exercise in Concurrency', John A. Trono [Tro94] proposes a solution to the 'Santa Claus' concurrency problem. The 'Santa Claus' problem involves Santa Claus

[1]https://github.com/rmax/scrapy-redis
[2]https://hadoop.apache.org

itself, elves and reindeer. The barrier synchronization aspect, common to our study's case, exposes itself in the requirement that Santa can only leave with his sleigh to deliver the presents when exactly 9 reindeer arrive to pull the sleigh. Trono presents a solution where this condition is taken care of using a reindeer counter and a semaphore on this counter (the semaphore prevents more than 9 reindeer pulling the sleigh).

In 'Loose Synchronization for Large-Scale Networked Systems' Albrecht et. al. [ATSV06] also research the synchronization concept of barriers, with 'participants' (processes) running on multiple machines with varying hardware specifications, communicating over slow or unreliable TCP/IP connections. Waiting for every participant to reach the barrier, before allowing all participants to continue, could negatively influence the 'liveness' factor of the entire computation process if there are participants who never or after a very long time reach the barrier. To prevent slow or crashing participants from negatively influencing the liveness factor, the authors introduce the concept of 'partial barrier'. With a partial barrier, not all participants need to arrive anymore at the barrier; continuation after the barrier is triggered when for example 90% of participants arrive on time. In the case of our web scraping study, it is preferred that 90% of bots reach the finish line successfully, rather than one bot blocking everything (which is common). The authors also introduce the concept of 'rate of release' to ensure that when participants receive a signal that they are allowed to continue after the barrier, not all participants continue their work at the same time to prevent performance problems (the authors give an example of all participants downloading a file from one host). In the case of our distributed bot, this concept is better avoided to make the time difference of product information extraction between each bot as small as possible.

Barcelona-Pons et.al [BPSAP+19] designed a framework('CRUCIAL') that supports synchronization among computations executed as Functions As A Service (FAAS). FAAS allows us to execute computations, without worrying about provisioning hardware resources. FAAS is stateless; functions cannot communicate to each other directly. The authors create a high-performance centralized object store, in order to let FAAS functions communicate and synchronize to each other. With minimal changes, existing multi-threaded Java code can be ported to function in a distributed FAAS environment. Starting a local Java thread corresponds to calling a server-less function. All synchronization methods from java.util.concurrent, such as cyclic barriers and semaphores, are also available in CRUCIAL.

For the purpose of modeling and simulation of synchronization , Vijay Gehlot[Geh19] explains Colored Petri Nets. Colored Petri Nets are used to model and simulate how scraping is synchronized among different outlets.

Chapter 3

Methodology

The main research question of this thesis is how to compare multiple outlets simultaneously. This research question can be split into the following sub-questions:

- Why automatic extraction instead of manual extraction from outlets?
- How can data automatically be extracted from different types of outlets? E.g. How can product offers be extracted from mobile applications?
- When comparing items from different outlets, how can we be sure that the same items are compared? When are items equal?
- How to design a distributed system that can compare outlets simultaneously?

In the following sections, these research questions are discussed. The discussion of the software architecture to compare outlets simultaneously is put in a separate chapter 4 .

3.1 Reasons to run an automated experiment

There are several reasons why an automated experiment is chosen over a manual experiment.

An argument against an automated experiment is that pricing data could be extracted manually from outlets. If the number of outlets and frequency of data extraction is limited, why would one bother to develop an automated price extraction tool?

Developing a scraping bot for one outlet takes around two days, and when an outlet changes, the scraper needs to be updated accordingly. Manually extracting information from a given outlet takes around one minute. Furthermore, humans will not be stuck when an outlet is updated (a simple popup with COVID-19 information can already crash a bot).

A reason why an automated experiment is chosen is the problem of simultaneity. Different outlets have to be compared to each other within the smallest time interval possible. Scraper bots can be run simultaneously, either in different threads or on different (distributed) machines or a combination of both.

3.2 Limitations of running an automated experiment

Running an automated experiment comes with challenges.

These challenges include web bot detection and website cloaking [HTWH18]. Web bot detection can be done in several ways, including behavioral difference detection between bots and humans, analysis of traffic generated by web bots and analysis of (web browser) properties. If a web bot is detected as such, possibly a 'cloaked' version of the website with different content will be served, that is optimized for search engines. To avoid being blocked or to get cloaked content, using different techniques, a web bot can be tweaked to appear as a real user. Making a web bot appear as a real user is a cat-and-mouse game where website owners continuously modify their tactics. Several techniques exist that make website owners believe that their website is being accessed by a real visitor instead of a web bot, however achieving 100% success rate is probably impossible[1].

Another major problem in automated web scraping, is that outlets are continuously changing. The slightest change to a website can break a scraper. There are techniques to make web bots more resilient to continuously changing websites [TEBS12], however according to our knowledge there is no method to make a web bot adapt to website changes automatically without any human intervention at all. Hybrid smartphone applications (e.g. Android applications utilising 'WebView') face the same problem.

Another limiting factor is the amount of hardware resources available. To prevent bot detection, this study's scraping framework only runs headful scrapers, as opposed to headless scrapers that require less hardware resources. To run a headful scraping session on a command-line based Linux distribution, an X-Window server has to be started to supply a 'virtual' display[2].

In this study's configuration, there is only one scraping thread per machine (each machine has limited resources); in order to compare several outlets simultaneously, several machines were needed. The cluster configuration consisted of four 'machines': a Kubernetes[3] cluster with two nodes specialized in desktop website scraping, a Windows VM with a smartphone attached and a home laptop with a smartphone attached. This enables us to compare for a given company, for example, its German website versus its French website versus its German mobile application versus its German mobile web browser.

A last limitation of automated scraping is the number of scraping bots that can be built and maintained. Building and maintaining scrapers is time consuming. Ideally, scrapers are built for all companies and all outlets in the world. However, the bots in this study are built and maintained by one person. Creating a new bots takes several days and a bot also needs to be maintained.

3.3 Scraping Techniques

In this study, data is extracted from three types of outlets: smartphone applications, mobile websites that run on smartphones, and desktop websites.

3.3.1 Scraping from mobile applications

Scraping data from smartphone applications is not well researched. Therefore, part of this study was to explore all options available to scrape information from smartphone applications.

Intercepting and replaying HTTP web service calls

One method considered by this study to scrape data from mobile applications, is the interception and replay of HTTP requests from a mobile application. The idea is to re-route all HTTP traffic from a mobile device to a proxy server and look for useful HTTP requests (for example, one HTTP request that retrieves all flights for a given flight itinerary).

[1] https://www.npmjs.com/package/puppeteer-extra-plugin-stealth
[2] https://medium.com/dot-debug/running-chrome-in-a-docker-container-a55e7f4da4a8
[3] https://kubernetes.io/

Unfortunately, when a person using a smartphone application queries for a flight, 10s of requests can be fired to different domain names, making it challenging to find 'useful' HTTP requests.

In addition, requests are usually SSL encrypted. SSL encryption can be bypassed, by forcing a mobile device to use an SSL certificate created by the proxy server, allowing the proxy server to decrypt all requests[4]. Some (Android) apps try to prevent this man-in-the middle attack, by using a technique called 'SSL pinning' [SR15]. With SSL pinning, an application author embeds his own certificate to encrypt communication from his application to his own server. SSL pinning can still be bypassed by for example reverse engineering, but doing this for many apps is out of scope for this study.

Even after extracting useful HTTP requests, challenges remain.

To replay a HTTP request, extra effort may be required. For example, intercepted HTTP requests could include authentication tokens that are impossible to reuse (e.g. One-time cookies [DCAT12]) .

Another barrier to replaying HTTP requests, is the use of anti-bot frameworks. During interception of requests to Vueling Airlines, the author discovered the use of Akamai's Bot Manager Premier SDK[5]. Bot Manager Premier SDK is using an undisclosed amount of information originating from a mobile device to determine whether a bot is accessing the back-end API server. The information used includes 'device characteristics, device orientation, accelerometer data, touch events, etc.'. This makes it very difficult to use the API from a scraping bot.

However, when all HTTP requests of interest have been extracted, parameters are well understood and sessions can be replayed or spoofed, this method of extracting information from mobile apps is very efficient on hardware resources (e.g. there is no need to provision hardware resources for an Android emulator: HTTP requests can be directly sent to the API server).

However, because of the aforementioned difficulties and because multiple mobile application outlets are considered in this study, another method has been chosen.

Using testing frameworks for scraping

A method which does not involve interception of backend requests, is the use of mobile application testing frameworks. Bertsch et.al. [BMW17] extracted prices from mobile apps with Appium. Appium is a black-box testing framework, meaning that apps do not need to be modified in order to allow testing. Appium [6] is using vendor provided testing frameworks (for Android: UiAutomator2[7]), running a server that exposes its functionality through a REST API. Tests can either be executed on a real mobile device or on an emulator.

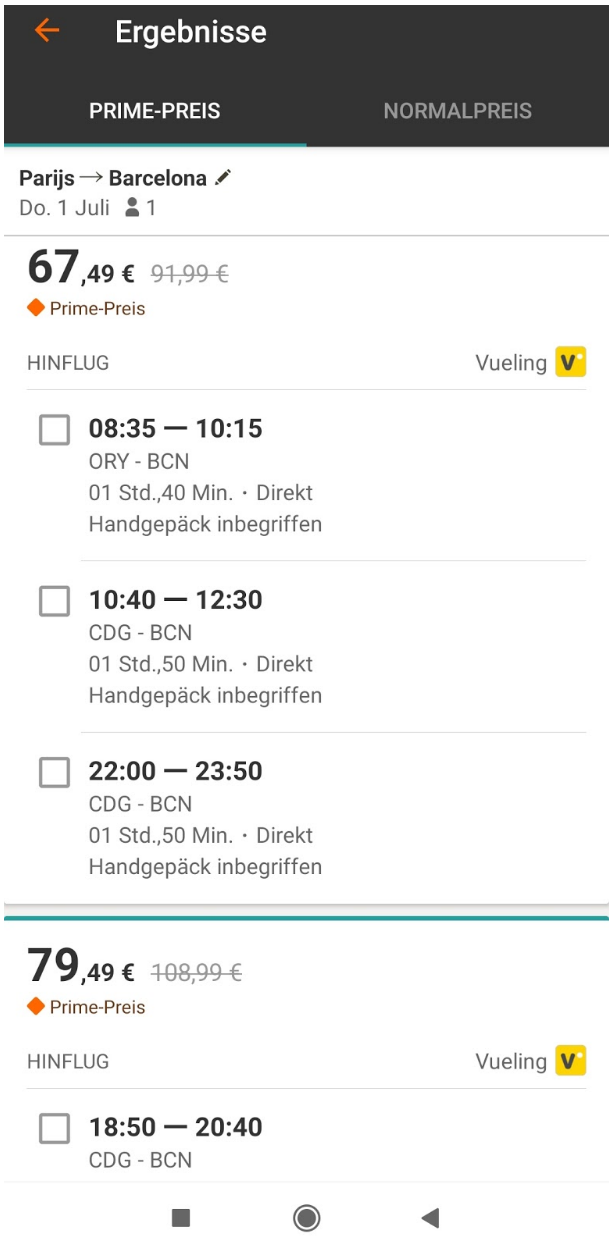
With Appium, running tests on a mobile application is comparable to running tests on a website, with one important difference: on a mobile application elements that are off screen cannot be selected. This can be illustrated with an example in Figure 3.1:

[4]https://portswigger.net/burp/documentation/desktop/getting-started/proxy-setup/certificate

[5]https://developer.akamai.com/tools/sdk/bot-manager

[6]https://appium.io/docs/en/about-appium/intro/?lang=en#appium-concepts

[7]https://appium.io/docs/en/drivers/android-uiautomator2/


Ergebnisse

PRIME-PREIS NORMALPREIS

Parijs → Barcelona
Do. 1 Juli 1

67,49 € ~~91,99 €~~
Prime-Preis

HINFLUG Vueling

08:35 — 10:15
ORY - BCN
01 Std.,40 Min. · Direkt
Handgepäck inbegriffen

10:40 — 12:30
CDG - BCN
01 Std.,50 Min. · Direkt
Handgepäck inbegriffen

22:00 — 23:50
CDG - BCN
01 Std.,50 Min. · Direkt
Handgepäck inbegriffen

79,49 € ~~108,99 €~~
Prime-Preis

HINFLUG Vueling

18:50 — 20:40
CDG - BCN

Figure 3.1: Scrolling offers on the Opodo Mobile Application

As can be seen in Figure 3.1, there are three flights available from Vueling Airlines at a price of €67,49 per ticket. This is followed by one flight at €79.49. However, one cannot be sure if there is only one flight at €79.49, because only what is on screen (in the viewport), can be selected.

A scraping bot has to scroll down, pixel by pixel, to retrieve more possible flights. A possible solution can be seen in listing 3.1:

Listing 3.1: Code to extract all flights from a scrollable list on a mobile device

```
async scrapeFromSearch(inputData) {
    await this.clickElementByResource('com.opodo.reisen:id/search');
    var rect = await this.appiumClient.getWindowRect();
    var rectX = rect.width / 3;
    var rectY = rect.height / 1.1;
    var flightOffersOnScreen = [];
    var flightOffers = [];
    var equalCount = 0;
    while (true) {
```

```
10     var oldFlightOffersOnScreen = flightOffersOnScreen.slice();
11     flightOffersOnScreen = [];
12     var prices = await this.getElementsByResourceId(‘com.opodo.reisen:id/flights_price’);
13     var departureTimes = await this.getElementsByResourceId(‘com.opodo.reisen:id/departure_hour
           ’);
14     var arrivalTimes = await this.getElementsByResourceId(‘com.opodo.reisen:id/arrival_hour’);
15     var originsDestinations = await this.getElementsByResourceId(‘com.opodo.reisen:id/
           departure_and_arrival’);
16     var airLines = await this.getElementsByResourceId(‘com.opodo.reisen:id/airline_name’);
17     for (var i = 0; i < departureTimes.length && i < arrivalTimes.length && i <
           originsDestinations.length; i++) {
18       var flightOffer = new FlightOffer();
19       var bounds = await departureTimes[i].getAttribute(‘bounds’);
20       const departureY = parseInt(bounds.match(/\d+/g)[1]);
21       if (prices.length == 1) {
22         bounds = await prices[0].getAttribute(‘bounds’);
23         const priceY = parseInt(bounds.match(/\d+/g)[1]);
24         if (departureY > priceY) {
25           flightOffer.price = await prices[0].getText();
26         }
27       }
28       else if (prices.length > 1) {
29         for (var j = 0; j < prices.length; j++) {
30           bounds = await prices[j].getAttribute(‘bounds’);
31           const priceY = parseInt(bounds.match(/\d+/g)[1]);
32           if (departureY > priceY) {
33             flightOffer.price = await prices[j].getText();
34           }
35         }
36       }
37       if (airLines.length == 1) {
38         bounds = await airLines[0].getAttribute(‘bounds’);
39         const airLineY = parseInt(bounds.match(/\d+/g)[1]);
40         if (departureY > airLineY) {
41           flightOffer.airline = await airLines[0].getText();
42         }
43       }
44       else if (airLines.length > 1) {
45         for (var j = 0; j < airLines.length; j++) {
46           bounds = await airLines[j].getAttribute(‘bounds’);
47           const airlineY = parseInt(bounds.match(/\d+/g)[1]);
48           if (departureY > airlineY) {
49             flightOffer.airline = await airLines[j].getText();
50           }
51         }
52       }
53       const deptT = await departureTimes[i].getText();
54       flightOffer.departureTime = deptT;
55       const arrT = await arrivalTimes[i].getText();
56       flightOffer.arrivalTime = arrT;
57       const txt = await originsDestinations[i].getText();
58       flightOffer.origin = txt.split(‘-’)[0].trim();
59       flightOffer.destination = txt.split(‘-’)[1].trim();
60       flightOffersOnScreen.push(flightOffer);
61       if (_.findWhere(flightOffers, flightOffer) == null) {
62         var screenshot = await this.takeScreenShot(‘OpodoAppScraper’);
63         const screenShotFlightOffer = { ...flightOffer };
64         screenShotFlightOffer.screenshot = screenshot;
65         flightOffers.push(screenShotFlightOffer);
66         logger.info(‘adding new flight offer’);
67       }
68       else {
69         logger.info(‘skipping flight offer’);
70       }
71     }
72     if (_.isEqual(oldFlightOffersOnScreen, flightOffersOnScreen)) {
73       equalCount++;
74       if (equalCount > 3)
75         break;
76     }
77     await this.appiumClient.touchAction([
78       { action: ‘press’, x: rectX, y: rectY },
79       { action: ‘wait’, ms: 500 },
80       { action: ‘moveTo’, x: rectX, y: rectY * 0.9 },
81       ‘release’,
82     ]);
83   }
84   return flightOffers;
85 }
```

In Listing 3.1, the code is shown for scraping flight offers from Opodo's application. The first part, which is filling out input data, is not shown; listing 3.1 only displays the second part of the scraping process (*scrapeFromSearch*), which is ‘tapping’ the search button after the origin, destination and flight date have been entered.

After tapping the search button (line 2 in Listing 3.1), a list of offers will be displayed. All flight offers are extracted from the current view port, with the complexity that flight offers are grouped per price and per airline. Extracted flight offers from the current view port are added to a global list of *flightOffers*(declared on line 67 in Listing 3.1), after which the scrolling down is executed by a fixed number of pixels (line 77 in Listing 3.1).

Scrolling is done in small steps, so as not to 'overshoot' any offers. Because scrolling is done in small steps, it is possible that the same offer is found multiple times. However, only offers that have a unique combination of airline, departure time, arrival time, and price will be added to *flightOffers* (line 65).

Note that at the end of the loop (line 72), we verify whether the same list of offers is displayed compared to the offer list before the last scroll. If there is no difference, scrolling stops, however, because offers may not be displayed consecutively (for example there are advertisements in between offers), scrolling is done three more times even when no new offers have been discovered.

3.3.2 Scraping from websites

Website scraping is trivial, however appearing as a human user instead of a web bot is not.

Desktop Website Scraping

A common way to scrape the web, is to program a script that drives a headless browser. Unfortunately, automated headless browsers can be detected as such [JKV19]. Detection is based on deviations from 'normal' user behavior. This can be as simple as checking static properties such as User Agent Strings, or can go as far as comparing behavior, such as tracking mouse cursor movements. Researching how to bypass bot detection is out of scope of this study. Instead, the framework 'puppeteer-extra-plugin-stealth'[8] has been used; the author describes its purpose as follows: 'It's probably impossible to prevent all ways to detect headless chromium, but it should be possible to make it so difficult that it becomes cost-prohibitive or triggers too many false-positives to be feasible'.

It is shown that Puppeteer with headless Chrome and Plugin Stealth activated, cannot be detected as bot (except for MQ_SCREEN: 'Use media query related to the screen' [9]). In our experiments, for all companies researched, this setup was still detected by one company website as a bot, namely, Opodo, which displayed a full-page advertisement instead of displaying flight offers.

It was noticed that Opodo was triggering bot detection when scraped headless, however when scraping in headful mode, bot detection disappeared. Therefore, even though more resources are required, it was decided to create a headful bot that could run in a windowless Linux container with the help of a virtual frame buffer. [10]

Smartphone Website Scraping

Desktop web browsers can access websites as a mobile device. For example, Chrome Devtools[11] offers 'device mode', that can simulate a mobile viewport, limit network (e.g., 3G) and CPU speed, put a custom Geo-location and set screen orientation. Chrome 'Devtools' gives an approximation to a real mobile device; it is still recommended to test a website on a real mobile device, since for example, CPU architecture is different between desktops and smartphones. Therefore, in this study, mobile website outlets are scraped from within a smartphone.

[8]https://www.npmjs.com/package/puppeteer-extra-plugin-stealth
[9]https://github.com/antoinevastel/fpscanner
[10]https://medium.com/dot-debug/running-chrome-in-a-docker-container-a55e7f4da4a8
[11]https://developer.chrome.com/docs/devtools/device-mode/

3.4 Equality of products

Because product offers displayed on one outlet may be different from those displayed on another outlet, a definition of offer equivalency is needed to help in correctly attributing price discrimination. To be clear, the framework is only collecting data; grouping offers that are equal is the responsibility of the analyst.

Matching products to each other that have been found by web scraping is not a trivial task, since products found on the web may not have unique identifiers (such as an International Article Number[12]).

We have to accept that exact matching of products is not possible. Therefore, per product category, an approximation of equivalency will be defined.

In this study, two product categories are under consideration: flights and hotel rooms.

For a flight to be equal to another flight, the following properties have to be equal: departure time, departure airport , arrival time, arrival airport, airline company, and flight number. All other properties of a flight offer will be ignored.

Definition 1 (Equivalence of flights) *Two flights, F_a and F_b are considered equivalent, notation $F_a \approx F_b$, if and only if all of the following hold:*

- $F_a.departure.time = F_b.departure.time$
- $F_a.departure.airport = F_b.departure.airport$
- $F_a.arrival.time = F_b.arrival.time$
- $F_a.arrival.airport = F_b.arrival.airport$
- $F_a.flightclass = F_b.flightclass$

For a hotel room to be equal to another hotel room, the following properties must be equal: location, hotel name, check-in time, and checkout time.

Definition 2 (Equivalence of two hotel room reservations) *Two hotel room reservations, H_a and H_b are considered equivalent, notation $H_a \approx H_b$, if and only if all of the following hold:*

- $H_a.location = H_b.location$
- $H_a.name = H_b.name$
- $H_a.check\_in.time = H_b.check\_in.time$
- $H_a.check\_out.time = H_b.check\_out.time$

For both flights and hotel rooms, the location has to be as exact as possible; for example, the city center and suburbs are not considered equivalent.

These definitions of equality could be extended with more properties to increase accuracy, however, the amount of work to build a web bot needs to be taken into account; more properties means a higher workload to construct and maintain a suitable web bot that extracts these extra properties.

In any case, the price differences that are found with these equality definitions, will always need to be verified manually to be certain that the price difference is not originating from the fact that different products are being compared.

[12]https://en.wikipedia.org/wiki/International_Article_Number.

Chapter 4

Design

In this chapter, the design is discussed of a distributed software architecture that can compare multiple outlets simultaneously.

First, a section is devoted to the requirements that such a software architecture needs to satisfy. The requirements are tailored to compare multiple outlets in the context of price differentiation.

Already early on in the research, the need for automatic comparison of outlets became clear, and an initial version of a program was developed to assist in simultaneous comparison of mobile phones and desktop websites. A small price differentiation study was done that helped with the discovery of several shortcomings in the initial design.

The initial design also made clear that a distributed software was needed, for example to compare a mobile application on one phone and a mobile website on another phone. Different approaches to distributing data extraction are considered.

Once the solution for distribution is chosen, synchronizing data extraction from multiple outlets in a distributed environment is discussed.

The final synchronized and distributed software architecture is illustrated by means of a Colored Petri Net.

4.1 Framework Requirements

The purpose of this study's framework is the comparison of products and/or services from multiple outlets, simultaneously. To this end, there are several requirements for the system listed in Table 4.1

Requirement #	Description
1	The system should keep running at all times.
2	Bots can be distributed over multiple machines.
3	Bots can extract data from any type of outlet.
4	Comparison of outlets should be synchronous.
5	Extracted data must be verifiable.

Table 4.1: Framework requirements

The first requirement is resilience. Because scraping bots depend on many external factors to run properly, crashes are guaranteed to occur. For example, frameworks for testing mobile phone applications are not designed with web scraping in mind; scraping may crash when run for many days on a smartphone or smartphone emulator. A requirement is that when a scraping bot is crashing, it will be restarted automatically for a limited number of times. Moreover, a crash of one scraping bot cannot influence the other running scrapers. And a crash of one bot is certainly not allowed to crash the whole system. Network connections in a distributed cluster are not guaranteed to be stable; for example, a bot that was deployed on a laptop connected to the internet with a 4G connection, crashed repeatedly. Upon further inspection of the 4G connection, very short but frequent drops in the connection were analyzed, making the bot lose connection with another machine that was sending scraping bot commands. A requirement is that in such a case, the bot automatically reconnects.

It should be possible to distribute several bots over several machines. The reasons for distribution include standard benefits of distribution such as fail-over, but also the possibility to allow simultaneous comparisons between more outlets by adding more machines to the cluster. Adding a new machine to the cluster should be of minimal effort, and a new machine should start processing scraping jobs as soon as it is added. Moreover, scraping bots distributed over multiple machines should report to a centralized log system to efficiently keep track of errors.

The framework should support website scraping and mobile application scraping, however, it should be possible to add any other type of outlet (for example, affiliate data feeds[1]).

Scraping of different types of products and/or services should be supported, including but not limited to flight tickets and hotel room reservations.

Comparison among different outlets should be synchronous. This is especially important for flight tickets and hotel room bookings. When two or more outlets are compared to each other, extraction of offers should happen simultaneously; that is: there can be a maximum delay of an analyst's defined number of seconds between the moment offers are extracted from one outlet and the moment offers are extracted from another outlet.

All data that has been scraped, must be verifiable, for example, by taking a screenshot of a view of a given outlet where the offer was found. It is always possible that scraping bots return wrong data, therefore, manual verification of previously collected prices should be possible.

4.2 Architecture Design

In this section, the design of the software architecture will be discussed. The final software architecture is the result of an evolutionary process, starting from a single-node design and ending in an architecture that supports an unlimited number of scraping bots that can be distributed over multiple nodes and that can synchronize to each other.

4.2.1 Initial Design

At the time of the initial design, there was no requirement to distribute the workload over several nodes.

The initial design came forth from a small experiment, where the requirement was to compare prices for five companies from its mobile application outlet and its website outlet. After manual inspection of all offerings and after noticing that prices can change frequently, the idea emerged that synchronization between outlets was needed. A logical point to do synchronization was 'synchronization on search', which means that two bots

[1]https://www.postaffiliatepro.com/affiliate-marketing-glossary/affiliate-data-feed/

progress individually and then wait until search parameters are entered on both outlets. An example of this can be seen in Figure 4.1.

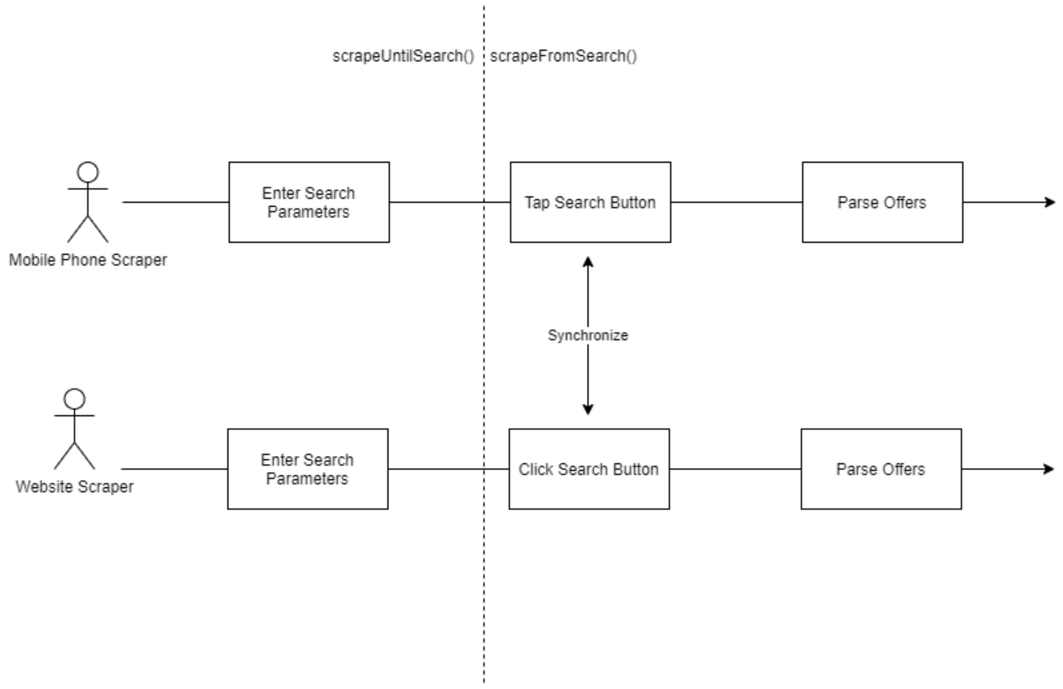


Figure 4.1: Initial Design

As can be seen in Figure 4.1, the scraping process is split into two parts. Each bot implements a common interface with two methods: *scrapeUntilSearch()* and *scrapeFromSearch()*.

To execute a comparison between a mobile application and a website, first *scrapeUntilSearch()* is called on both website scraper and application scraper. During this phase, search parameters are entered (e.g., for flights: origin, destination, flight date). The timing of this phase can be different among outlets: a website bot is already finished, while the corresponding application bot is still scrolling a calendar pixel by pixel, to select the correct flight departure date. Both bots have to reach the ‘search’ barrier, even when one bot takes triple the amount of time of the other bot.

To launch two processes at the same time and wait for both processes to finish, the following statement is executed:

```
await Promise.all([appScraper.scrapeTillSearch(input_params),
webScraper.scrapeTillSearch(input_params)])
```

scrapeAppTillSearch and *scrapeWebTillSearch* both return promises that resolve when filling out input data (such as departure airport) finishes without errors, or reject when an exception occurs.

When both scraping bots reach the search button barrier, *scrapeFromSearch()* starts by ‘clicking the search button’. For each outlet, the first price displayed is extracted and returned:

```
const [priceApp, priceWeb] = await Promise.all([
appScraper.scrapeFromSearch(), webScraper.scrapeFromSearch()]);
```

It was assumed that the first price displayed on an offer list is the cheapest and that the first offer in every outlet is the same.

4.2.2 Shortcomings of Initial Design

Running the implementation of this design for one week, comparing website versus mobile applications for five airline companies, produced remarkable results. For example, for Air France, mobile application prices were consistently higher than website prices as can be seen in Figure 4.2.

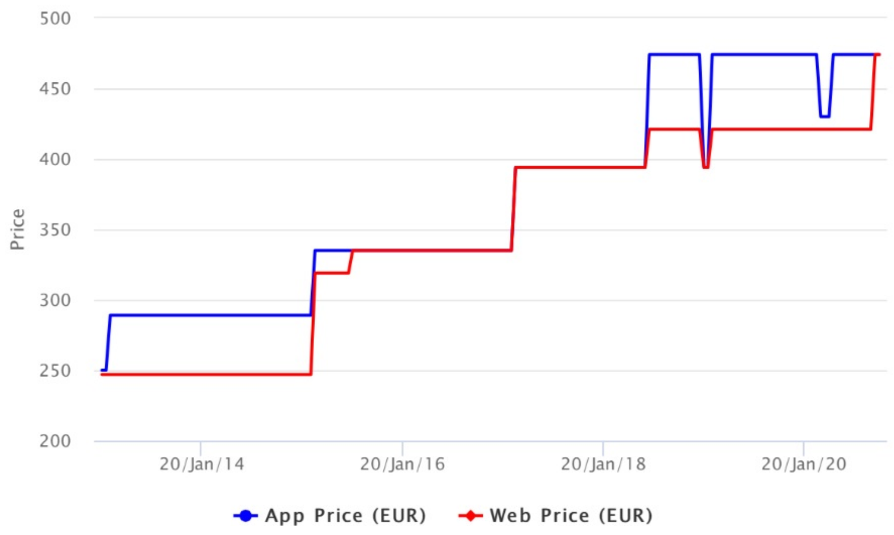


Figure 4.2: Mobile application price versus Website price for Air France

However, manual inspection of the Air France's search results (screenshots were automatically taken by the framework), showed that on the mobile application, different product offers were displayed than on the website. Scraping bots only returned the first offer found and assumed that the website and mobile application would return the same offer list with the first offer in the list to be the cheapest. These assumptions were wrong and explained the price 'difference'. Another shortcoming of the initial design was that all bots were running on the same local machine sharing the machine's resources. Ideally, bots can be deployed in the cloud, except for bots that need to be deployed on premises, such as a bot connected to a physical Android device.

In the initial design's synchronization setup, another problem is that there is no limit on how long a web bot can run. When a bot is in deadlock (e.g., scrolling through a calendar infinitely), the comparison will never end.

Thus, only requirements #4 and #5 from Table 4.1 were satisfied in the initial design.

4.2.3 Distributed Scraping

To create a *distributed* bot comparison architecture (Requirement #2 from Table 4.1), several options are possible.

The question asked was: how can web bots be distributed over several machines?

The idea of one central 'controller' emerged, where comparisons can be launched by a scheduler, or manually by an analyst through a CLI. A comparison consists of two or more jobs. Jobs can be processed by any scraping bot on any machine. A scraping bot will start processing a job and return the extracted information to the controller as can be seen in Figure 4.3.

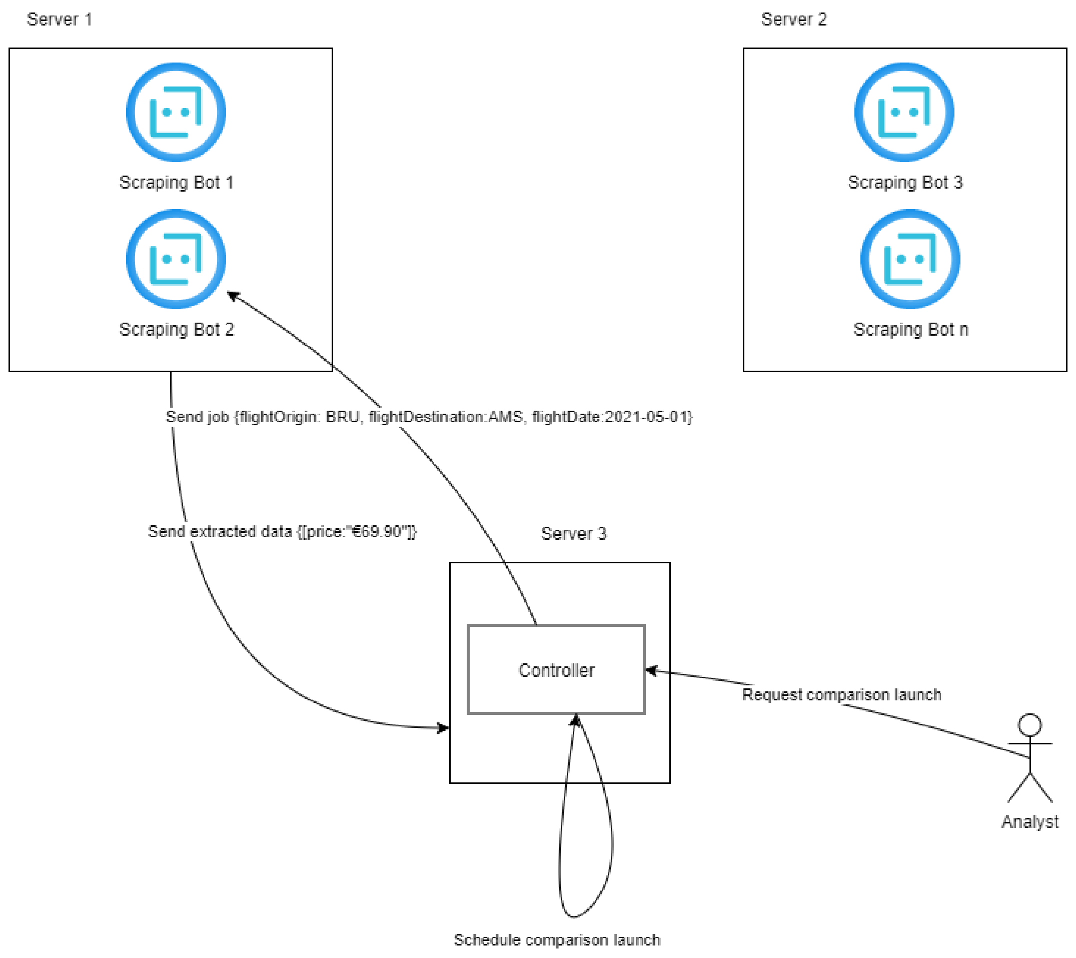

Server 1
Scraping Bot 1
Scraping Bot 2
Server 2
Scraping Bot 3
Scraping Bot n
Send job {flightOrigin: BRU, flightDestination:AMS, flightDate:2021-05-01}
Send extracted data {[price:"€69.90"]}
Server 3
Controller
Request comparison launch
Analyst
Schedule comparison launch

Figure 4.3: Controller communicating with Scraping Bots

In Figure 4.3, there are three different servers. On server 1 and server 2 together, four bots are deployed. On server 3, a controller is deployed through which an analyst can launch new instances of a comparison and where new instances of a comparison are launched by the controller's scheduler. A comparison consists of at least two jobs that are sent to the correct bots. In Figure 4.3, one of the jobs in a comparison requests a bot to search for flight offers from BRU to AMS on 2021-05-01.

Scraping jobs are long running operations; when a scraping bot receives a new job, multiple operations will be executed sequentially (e.g. navigating to a given URL and waiting, then extracting a certain element with a timeout of 10,000ms). Therefore, the launch of a new scraping job and the collection of prices is an asynchronous operation.

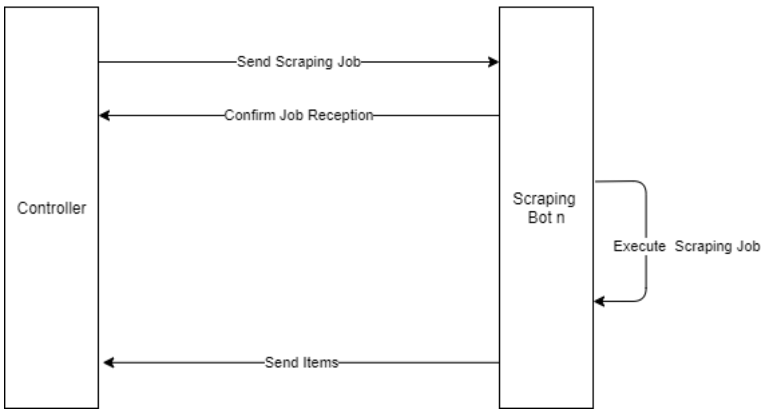


Figure 4.4: Interaction between Controller and Bot

In Figure 4.4, a controller sends a job to a bot. Since it takes time for a bot to collect items, the bot will return a confirmation that a job has been received and that the job will be processed. The controller can continue with other work. When the bot finishes the scraping process, it sends all the collected items back to the controller.

There are different methods of launching long running operations and collecting items produced by these operations.

Remote Procedure Calls [BN84] allow a procedure to execute remotely on another machine, while the semantics for the calling program are the same as when the procedure is executed locally. With RPC, correlation of calls and returns is handled by the RPC implementation. Different variations of RPC exist, for example RPC-WebSockets[2] is an implementation of RPC over the WebSocket communication protocol. A disadvantage with RPC is that when a new machine is added, jobs from the controller will not be sent automatically to web bots on the new machine; the machine needs to be registered first. In addition, when all machines are down, jobs are lost.

A more recent solution to executing remote operations is the cloud computing concept of 'Function As A service'(FAAS)[MPdF18]. The promise of FAAS is that a developer does not need to be concerned about the deployment of hardware resources; more web bots will be deployed and more hardware resources will be assigned when there are many web scraping jobs. Conversely, when there are no web scraping jobs available, resources will be released, which is suitable for this study's use case since web scraping jobs are only created a few times per day. In addition, web bots also fit with the stateless design of FAAS; our web bots do not need to resume from a previous state. FAAS is well supported by commercial cloud providers, e.g. AWS Lambda[3]. However, a requirement in this study is that web bots also run on privately maintained machines, for example, a personal computer at home that has a physical smartphone attached to it.

Yet another solution to distribute work among bots, is a message-oriented architecture [vST16]. In a message-oriented architecture, a job is sent to a specific queue. Scraping bots can subscribe to one or more queues and start processing these jobs. This is the architecture that has finally been chosen. Figure 4.5 depicts how work is divided:

[2] https://www.npmjs.com/package/rpc-websockets
[3] https://aws.amazon.com/lambda/

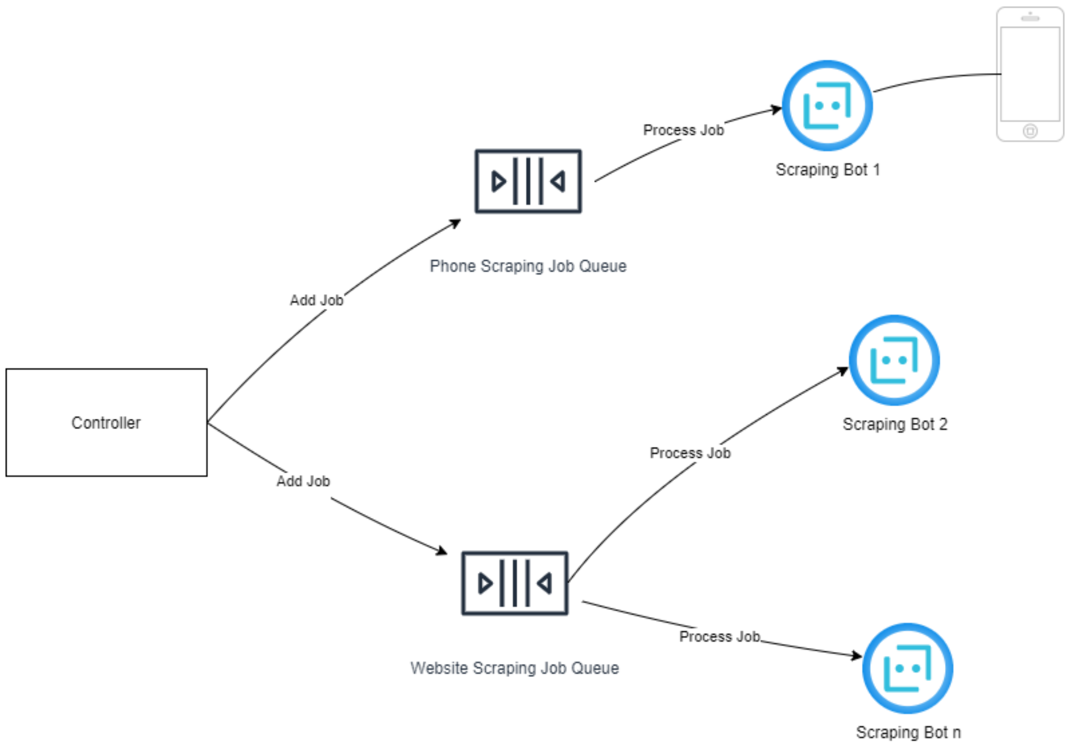


Figure 4.5: Message Driven Architecture

As can be seen in Figure 4.5, there are two queues (although there can be an unlimited number of queues). Each scraping bot has certain capabilities, for example, one scraper bot can scrape smartphones, while another scraper bot can only scrape websites. This is reflected in the queues that a scraper bot is subscribed to.

The controller takes comparison definitions as input from an analyst, and can run this comparison once or schedule the comparison to run continuously multiple times per day. An example of a comparison definition:

```
 {
    'scrapers': [
        {
            'params': {
                'useRealDevice': 'true'
            },
            'scraperClass': 'ExpediaAppScraper'
        },
        {
            'params': {},
            'scraperClass': 'ExpediaWebScraper'
        }
    ],
    'inputData': {
        'origin': 'BRU',
        'destination': 'AMS',
        'departureDate': '2021-07-01'
    }
}
```

In this comparison, the analyst is asking the system to compare a flight from Brussels to Amsterdam on two outlets: the mobile application and the desktop website of Expedia.

In addition, the analyst can supply parameters that depend on the type of outlet. For mobile applications, the parameter ‘useRealDevice’ means that scraping is to be executed on a real mobile device and not on an emulator.

For this comparison, the controller will split the work into two scraping jobs. One job will be sent to the Website Scraping Job Queue, while another job will be sent to the Phone Scraping Job Queue. An example of a job that is put on the Phone Scraping Job Queue:

```
{
  'comparisonRunId': 2623,
  'comparisonSize': 2,
  'comparisonId': 69,
  'params': {
    'proxy': '1.2.3.4'
  },
  'scraperClass': 'ExpediaAppScraper',
  'inputData': {
    'origin': 'BRU',
    'destination': 'AMS',
    'departureDate': '2021-07-01'
  }
}
```

As can be seen in this sample, the controller adds extra information about the comparison the job is part of. To understand this extra information, please refer to Appendix A.1.

Scraping jobs are taken off the queue by one of the scraping bots subscribed to the queue. A bot will start scraping; when the bot finishes, the original job together with the items collected by the bot, will be added to a queue collecting finished jobs. The finished job queue is continuously polled by the controller; the controller will store extracted information from bots in a relational database (see Appendix A.1).

A sample of a finished job:

```
{
  'comparisonRunId': 2623,
   ...
   'items': [
    {
      'price': '151 Euro',
      'departureTime': '10:25',
      'arrivalTime': '11:25',
      'origin': 'BRU',
      'destination': 'AMS',
      'airline': 'KLM',
      'screenshot': 'https://scraperbox.be/screenshots/
      ExpediaAppScraper-1616834552865.png'
    },
    ...
  ],
}
```

4.2.4 Synchronization of Distributed Scraping

When a comparison is launched, the requirement is that all jobs within a comparison are synchronized to each other. This means that offers have to be extracted from each outlet at the same time.

In the initial design, there were only two scraping bots: a mobile device scraping bot and a website scraping bot, executed in two threads on the same machine.

In a distributed design, a comparison can consist of an unlimited number of scraping jobs over multiple types of outlets. An unlimited number of scraping bots can be deployed on an unlimited number of machines.

A requirement of a distributed comparison framework was that the 'synchronization on search' from the initial design would be supported.

'Synchronization on search' is an instance of a 'barrier'[Tro94], where multiple processes all wait for a shared condition and then continue together after that barrier. In a comparison with only two scraping bots that run on the same machine, this could be done with one line of code:

Listing 4.1: Syncing two bots

```
await Promise.all([webScraper.scrapeUntilSearch(),
appScraper.scrapeUntilSearch()])
```

However, when there are multiple scraper bots distributed over multiple machines, how can scraper bots synchronize to each other?

In John A. Trono's article 'a new exercise in concurrency' [Tro94], processes that need to synchronize to each other, increment a centralized counter up to a fixed number (the fixed number is the barrier). To prevent multiple processes to increment the counter simultaneously, there is a semaphore on the counter.

In the case of a comparison: when the comparison consists of n scraping jobs, the barrier is reaching the end of *scrapeUntilSearch()* n times.

Barcelona-Pons et. al. [BPSAP+19] ported synchronization concepts from *java.util.concurrent* to a distributed environment, where distributed processes can synchronize using a central low-latency in-memory data store. However, existing open source solutions such as Redis in-memory database can also be used.

Our approach to synchronization is displayed in Figure 4.6.

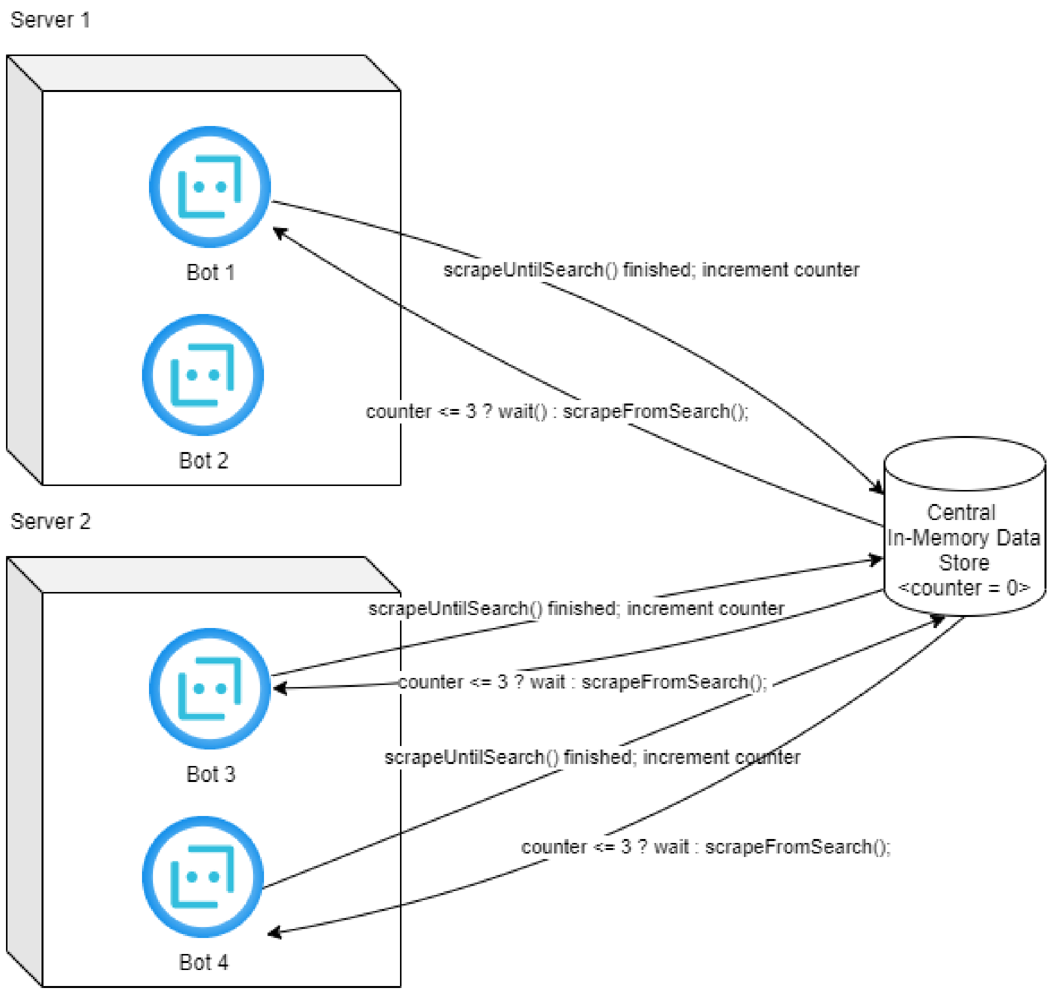

Server 1
Bot 1
Bot 2
scrapeUntilSearch() finished; increment counter
counter <= 3 ? wait() : scrapeFromSearch();
Central In-Memory Data Store <counter = 0>
Server 2
scrapeUntilSearch() finished; increment counter
counter <= 3 ? wait : scrapeFromSearch();
Bot 3
scrapeUntilSearch() finished; increment counter
counter <= 3 ? wait : scrapeFromSearch();
Bot 4

Figure 4.6: Synchronizing on Search

In Figure 4.6 , the use case is a comparison that consists of three scraping jobs, distributed over three scraping bots. The indent is that these three scraping jobs will synchronize.

To do synchronization, first a new counting variable will be created in a central in-memory data store (Redis[4]) and set to zero. Then for each job in the comparison, a scraping bot will start processing the job.

A participating bot will first execute *scrapeUntilSearch()*. When *scrapeUntilSearch()* is finished, the bot will request to increase the central data store counter associated with the comparison it is part of. Then, the bot will wait. Other bots in the comparison will do the same; when the slowest bot finishes scrapeUntilSearch(), then $counter == 3$ and all bots will continue and start executing *scrapeFromSearch()*.

An assumption with this approach is that all scraper bots in a comparison start processing a job at the same time. However, a guarantee is needed that this assumption will hold. For example, Figure 4.7 shows a situation where a mobile phone crashed.

[4]https://redis.io/

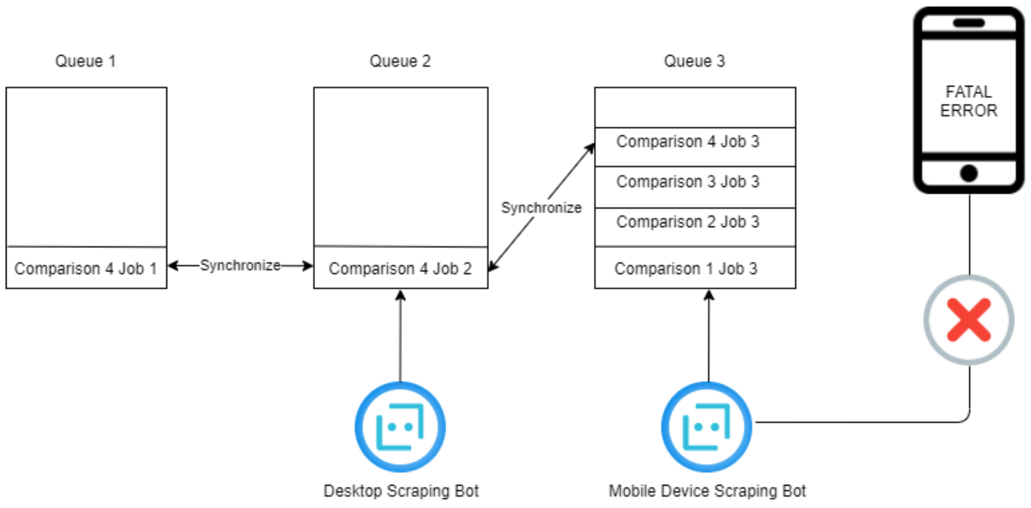


Figure 4.7: Queue 3 is Blocked

In Figure 4.7, Comparison 4 wants to synchronize among jobs 1, 2 and 3 that were pushed by the Controller to three different queues. However, since the mobile phone device has crashed, jobs from Queue 3 are not processed anymore; job 1 and job 2 will wait for job 3 to finish. In the worst case, this results in the current comparison and all subsequent comparisons utilising a mobile phone, to never finish. Therefore, some safety mechanisms are needed to prevent de-synchronization and keep the system alive at all times. This will be explained in the following section.

4.2.5 A Distributed and Synchronized Architecture

Synchronized scraping, while preventing deadlocks, can be accomplished in different ways.

The solution proposed here, makes use of synchronization on start, synchronization on search, and timeouts.

To explain the proposed solution, synchronization has been modeled in a Colored Petri Net [Geh19].

Colored Petri Nets extend Petri Nets. With Colored Petri Nets, it is possible to use a high level programming language and to create timed models.

As in a Petri Net, a Colored Petri Net consists of ‘places’ and ‘transitions’. A restriction is that two places cannot directly connect to each other; they always have to pass through a transition. Below is an example of a petri net.

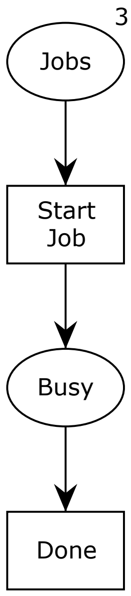


Figure 4.8: A Petri Net with two Places and two Transitions

As can be seen in Figure 4.8, there is a place called 'Jobs' that is initialized with three tokens (as indicated by the superscript). A token can represent anything; in this case, a token is a job represented by an integer.

Initially, only the transition 'Start Job' will be enabled. After 'Start job' is fired, there will be only two tokens left in 'Jobs' and there will be one token in place 'Busy'. The final state of this CPN will be three tokens in the place 'Done' and zero tokens in 'Jobs'.

The term 'Colored' in CPN refers to the built-in programming language of CPN that allows to declare types (called COLSETS). For example, a scraping job can be declared as a COLSET, where tokens of type scraping job are flowing through a petri net until they are processed. For a detailed guide about Colored Petri Nets, please see [Geh19].

In the next few pages, synchronization between scraping bots will be explained by the CPN model depicted in Figure 4.9.

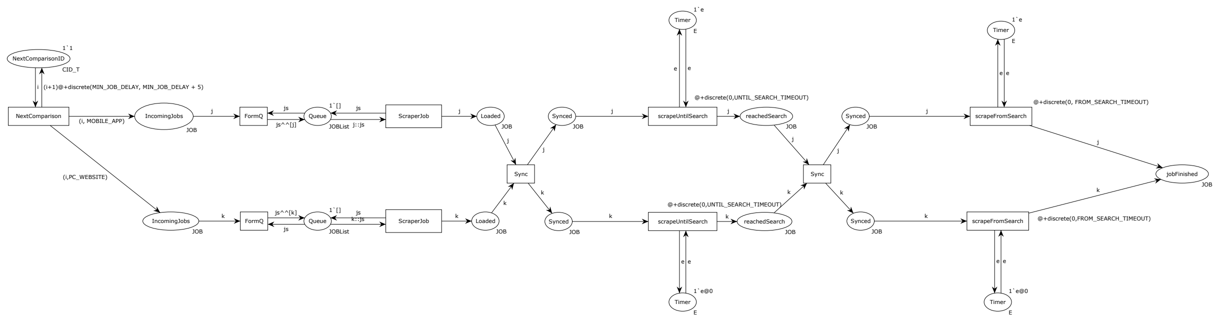


Figure 4.9: Synchronization of a Comparison with two Scraper Runs

In Figure 4.9, the synchronization of a comparison with two outlets is displayed. Comparisons are continuously launched by means of a *generator*. The generator creates a new comparison containing two scraping jobs, in this case one on a smartphone application and one on a desktop browser. Synchronization is done in two places: scraper runs from a comparison synchronize on start and synchronize on search. Note that this CPN is a simplification of the actual implementation; in the actual implementation, more than two simultaneous scraper runs are possible.

In the following figures, we will zoom in on each part of the CPN and explain the CPN in more detail.

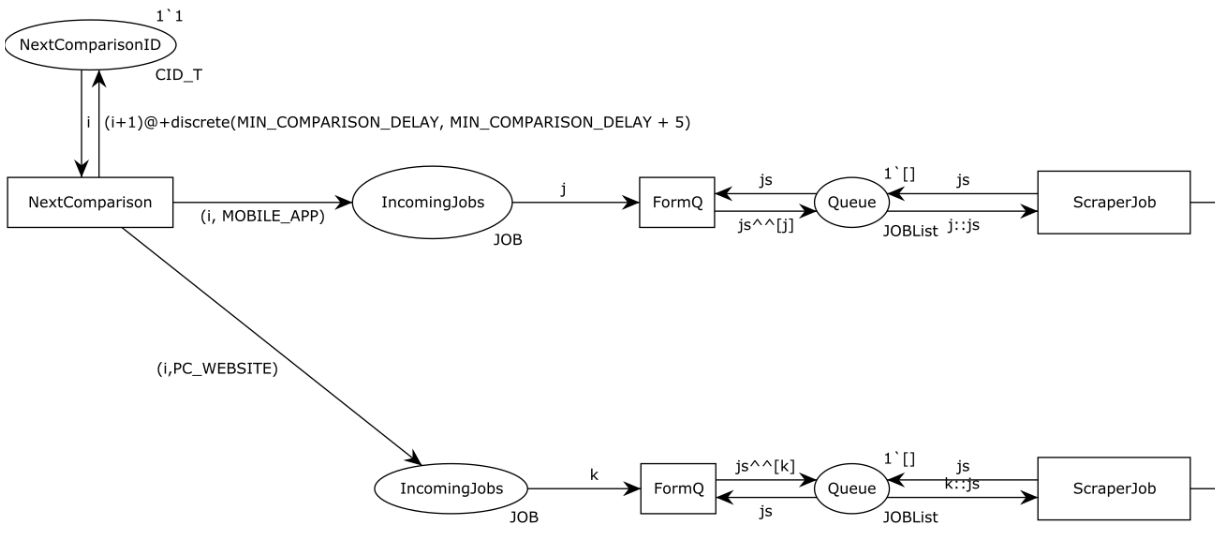


Figure 4.10: Synchronization of two Scraping Bots - Part 1

Part 1 of the synchronization process starts with the generation of a new comparison. Initially, the place 'NextComparisonID' contains one token of type CID_T (CID_T is a timed integer) with initial value 1 (at time zero). This token i is forwarded to the 'NextComparison' transition.

In addition, after a random time (between MIN_COMPARISON_DELAY and

MIN_COMPARISON_DELAY+5), a next comparison token is created. There is a pause between the launch of each comparison, to prevent scraping jobs from different comparisons to interfere with each other. The length of the delay after a new comparison launch depends on how many scraping bots are available for every outlet type, how many and what type of outlets a comparison can consist of, the maximum time scraping runs are allowed to synchronize between each other, and the maximum time each phase of a scraping run is allowed to run. Assuming at least one bot is available for every outlet in a comparison, the delay between comparisons is calculated with the following formula, where *comparison.length* is the number of jobs in a comparison; the timeout constants are explained later in this section:

$MIN\_COMPARISON\_DELAY = (beforeSearchTimeout + afterSearchTimeout + synchronizationTimeout \cdot 2) \cdot comparison.length$

Back to the transition 'NextComparison', the scraping work is split between two scraping bots. One job represented by a tuple (i, MOBILE_APP) requesting price extraction from a mobile application and one tuple (i, PC_WEBSITE) requesting extraction from a desktop website are sent for processing to a smartphone and a desktop scraping bot, respectively.

Now consider the PC_WEBSITE scraping run. From the place 'IncomingJobs', a token k of type JOB (where JOB is a tuple (*comparisonId*, OUTLET_TYPE)) is sent to a Queue. Initially, *Queue* is initialized to one token with a value of '[]'; then with the CPN operator '^^', a new job is added to the queue when k arrives. From 'IncomingJobs' a token k, e.g. (1, PC_WEBSITE) will arrive; k will be added to js after which the token in Queue has a value of [previous_jobs, (2, MOBILE_APP)]. The reverse happens when a job token is taken off the queue (with the CPN operator '::') to be sent for processing.

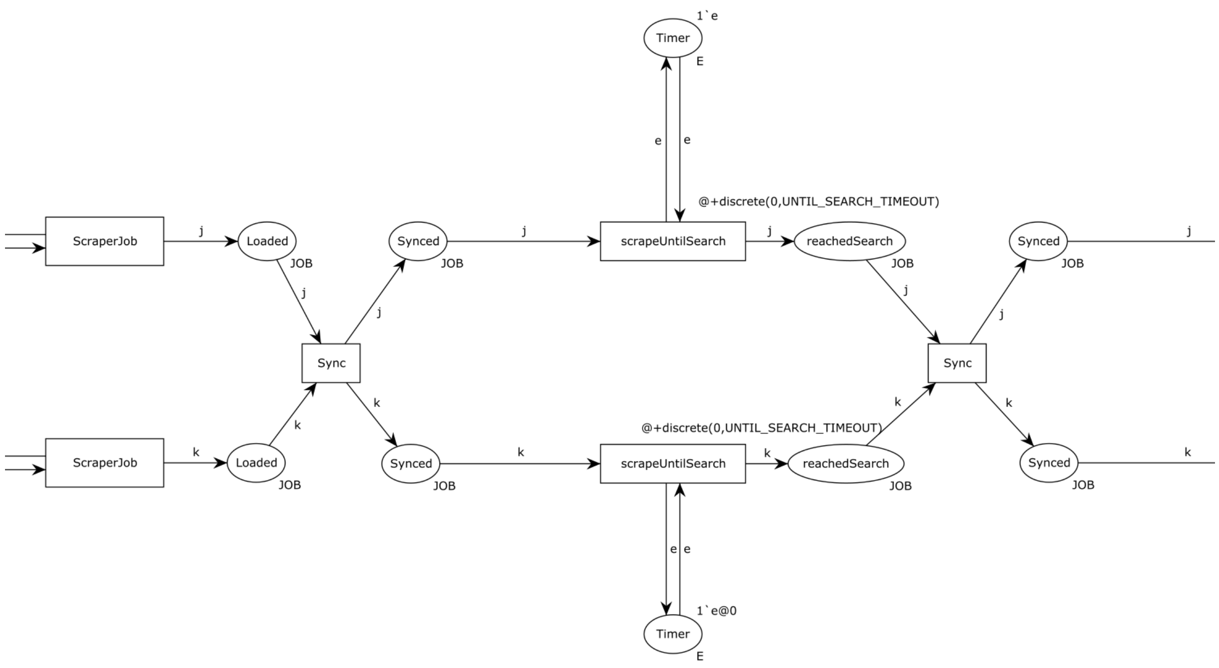


Figure 4.11: Synchronization of two Scraping Bots - Part 2

In Figure 4.11, a job may arrive sooner at the upper mobile application scraping process than the other job from the containing comparison will arrive for processing to the desktop bot. To make the two scraper runs start at the same time, synchronization is done by sending the two scraper runs through a common 'Sync' transition. Next, scraper runs are entering the input data in the place 'scrapeUntilSearch' which lasts a random amount of time (generated by the 'discrete' function) with a maximum allowed time of UNTIL_SEARCH_TIMEOUT. This is followed again by a synchronization, this time 'on search', where both scraper runs wait for each other to 'click the search button'.

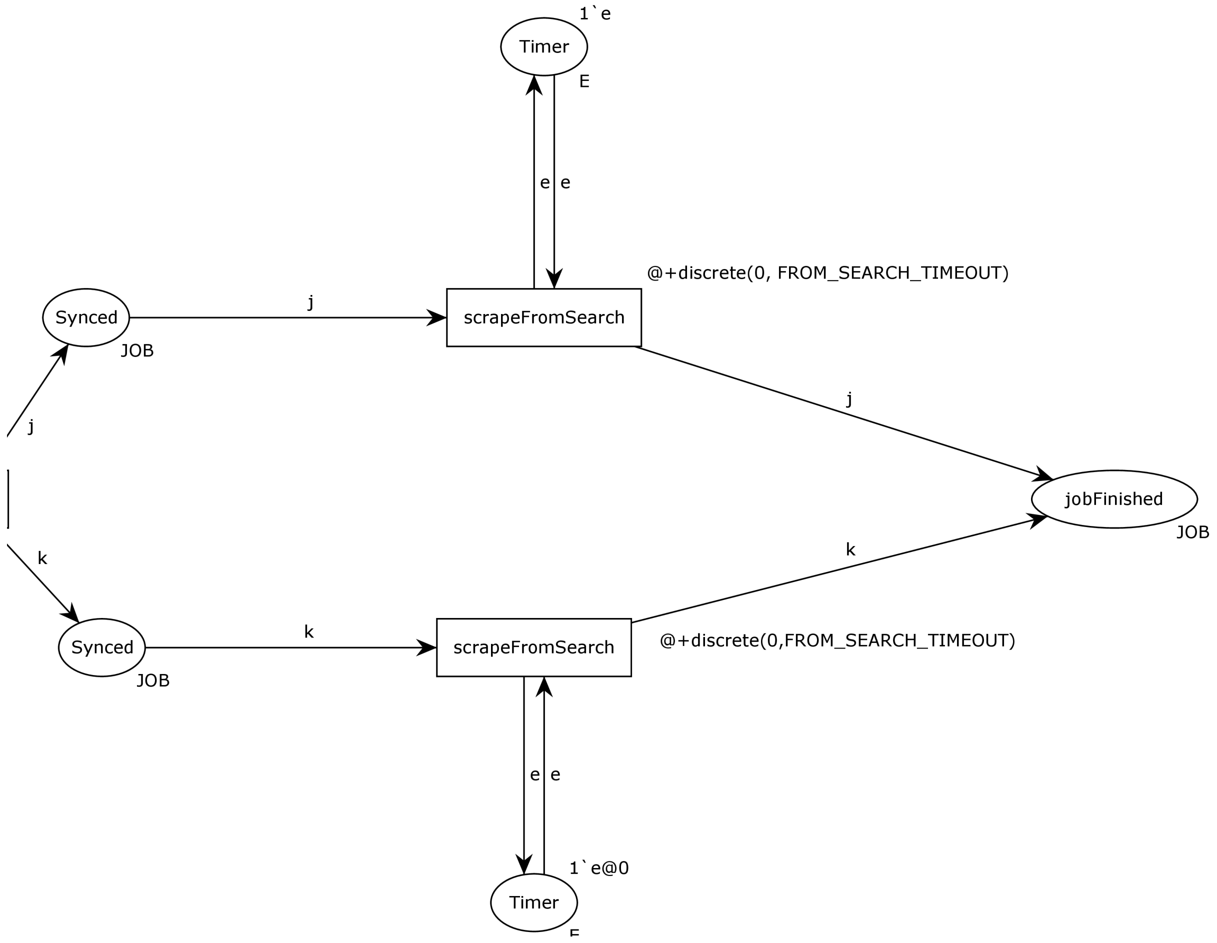


Figure 4.12: Synchronization of two Scraping Bots - Part 3

Finally, in Figure 4.12, the scraping process is continued with the last phase of the scraping process. In this phase, all product or service offers are collected, which takes a maximum of FROM_SEARCH_TIMEOUT. If this phase takes longer, the execution will be cut off and the scraper run will be marked as 'errored'.

The policy to handle errors is an all-or-nothing approach. When an error in one scraper run happens, the synchronization between all scraper runs in a comparison will stop (in fact, all scraper runs in a comparison will stop).

In the CPN diagram, only two scraper runs are displayed, however, synchronization between more than two scraper runs is possible; in Chapter 5 where experiments are conducted, up to three outlets are compared simultaneously.

4.3 From Architecture to Implementation

The architecture defined in the last section consists of many parts, and all those parts have to work together.

The controller is a centralized place where new jobs are generated and where jobs processed by scraping bots are returned. Jobs are put by the controller on a queue[5] based on a central Redis database. One or more bots poll queues for jobs; a bot will start

[5] https://github.com/OptimalBits/bull

processing a job if it is capable of processing this kind of job (for example, to process a mobile application job, the bot has to be connected to a smartphone).

All scraping bots are represented by the same Docker image that contains all dependencies for website scraping (including a Chrome web browser). If mobile application scraping support is needed, a scraping bot has to be connected to an Appium server with a smartphone outside of the container.

Because scraping bots are based on the same Docker image, adding a new scraping bot is a matter of creating a new Docker container. Once a new Docker container is created, the bot will connect to Redis and start polling for jobs. For our experiment, scraping bots were deployed by means of a Kubernetes ReplicaSet[6]. Bots specialized on mobile application scraping that needed a connection to a smartphone, were deployed as Docker containers on dedicated machines.

Synchronization of scraping bots that participate in a comparison was also done through a central Redis server; every comparison launched resulted in the creation of a new synchronization Redis variable around which all participating bots synchronized.

More details on the implementation can be found in Appendix A. In addition, the complete source code is open sourced[7].

To finalize this chapter, a short recap is given on how the final implementation satisfies our requirements expected from a synchronized distributed scraping system:

Requirement #1: The system should keep running at all times

When running the implementation in a price comparison study (Chapter 5) for more than two months with 12 different bots, while individual bots regularly crashed, comparisons with other bots were not affected and the system never completely halted. While the CPN model in Figure 4.8 serves as an illustration of how the comparison between outlets is synchronized, we do not provide a formal proof of any system properties (for example, there is no guarantee that a deadlock will never happen). System verification and model checking were outside of the scope of this study. We did, however, use the simulation functionality of CPN Tools on the model in Figure 4.8, with no deadlocks detected after running the simulation from start to finish for 10 iterations.

Requirement #2: Bots can be distributed over many machines

Bots can be deployed anywhere on a Kubernetes cluster or on a dedicated machine, and will start processing jobs as soon as they are started.

Requirement #3: Bots can extract data from different outlet types

All bots are based on a common blueprint; the implementation is only limited to bots that can extract from mobile applications, mobile browsers and desktop browser outlets; however bots for other outlets can be added as long as the outlet is accessible online.

Requirement #4: Comparison of outlets should be synchronous

See the discussion in Section 4.2.5 on how de-synchronization is prevented. Moreover, after a price differentiation study (Chapter 5) of more than two months, by verifying the timestamp of screenshots taken from each offer, the time difference between the extraction from multiple outlets was always smaller than the maximum allowed synchronization time.

[6] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/
[7] https://github.com/godfriedmeesters/diffscraper

Requirement #5: Extracted data must be verifiable

For every single item extracted, screenshots are taken. If item extractions are based on REST requests instead of capturing screen information, JSON files containing items are stored for verification.

Chapter 5

Validation by means of a price comparison study

To verify that the synchronized distributed web scraping system can be useful in practise, we performed a price differentiation study.

5.1 Experiment Design

Anecdotal reports and one research article [BMW17], reported that there are price differences when buying flight tickets. More specifically, the same flight ticket was found to be more expensive on mobile applications than on websites.

We decided to verify this and created a list of the following companies for which to compare prices:

- Air France
- Booking.com
- EuroWings
- Expedia
- Kayak
- Opodo

For each of these companies, the latest version of its mobile application is tested versus its desktop website. In addition, for each company, its German (.de) and French (.fr) website is compared. For the company Kayak, three outlets are compared simultaneously: its mobile application versus its mobile website versus its desktop website.

Because this study was done during the corona crisis, it was more difficult to find flight tickets. There were fewer flights or flights were suddenly canceled. Therefore, only flights sufficiently far in the future were chosen, that is: scraping was started on April 24, 2021, looking for one-way flights departing in July-August 2021. A list of comparisons was defined (see Table 5.1 and Table 5.2), with a varying selection of departure and arrival airports.

All comparisons were launched three times per day (that is, every eight hours). We could collect data more times per day. However, the chance of triggering bot detection would also increase since our pool of IP addresses was limited.

Initially on April 24, for website scraping, the bots picked an IP address randomly from four data center IP addresses. We also experimented with routing all website traffic through residential IP addresses from 2021-05-27 until 2021-06-05. Mobile application scraping was always conducted from the same Belgian residential IP address; mobile browser scraping was done from a Germany residential IP address.

Among the scraping bots for each company, there were differences in the type and amount of data that was returned. For Air France, Expedia, EuroWings, Kayak and Opodo, a list of all non-stop flights was returned. On Booking.com, all hotels within 5 kilometers of the city center were returned.

Website scraping was done on a Kubernetes cluster with two nodes (total cluster capacity: 4vCPUs with 8 GB RAM). One bot was deployed on each node. All website scraping was 'headful', inside a Linux container with Google Chrome ('google-chrome-stable' on Ubuntu 20.04) rendered to a virtual frame buffer. Before every new bot extraction, all browser history was deleted. Moreover, to appear as a real website visitor (for example, to create realistic user agents), Puppeteer Stealth[1] was put in place.

For mobile application scraping, a home laptop connected to a Xiaomi smartphone with Android 10.0 was used. The home laptop was running 24 hours/day, connected to a residential 4G connection in Belgium. For every application, the latest APK on `https://apkpure.com` was installed. All mobile application scraping was done from the Belgian residential IP address.

For mobile browser scraping, a dedicated server connected to a Motorola smartphone (with mobile Chrome 90.x) was used. The dedicated server was located in a university data center in Germany; scraping passed through a German residential proxy server.

5.2 Experiment Results

Scraping was started on April 24, 2021.

In total, more 5280 comparisons (measured on June 18, 2021) were executed (from the comparison definitions defined in table 5.1 and 5.2). In total, scraping bots were started 10406 times. Of the 10406 times that scraping bots started, 6440 times scraping bots finished successfully and 3966 times scraping bots stopped with a fatal error. The scraping bots that finished successfully, collected a total of 55667 offers.

As can be seen in Figure 5.1, the number of collected offers per day started out small and increased when we added more comparisons.

[1]`https://www.npmjs.com/package/puppeteer-extra-plugin-stealth`

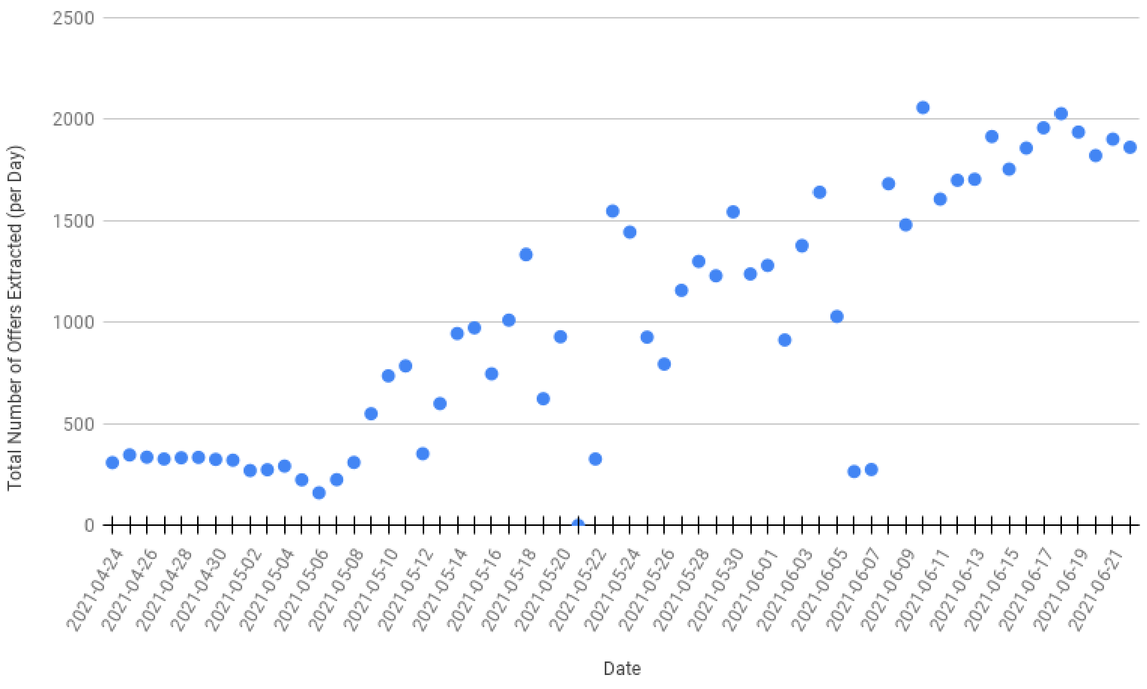


Figure 5.1: Number of Collected Offers per Day

In Figure 5.2, the number of started comparisons per day can be seen:

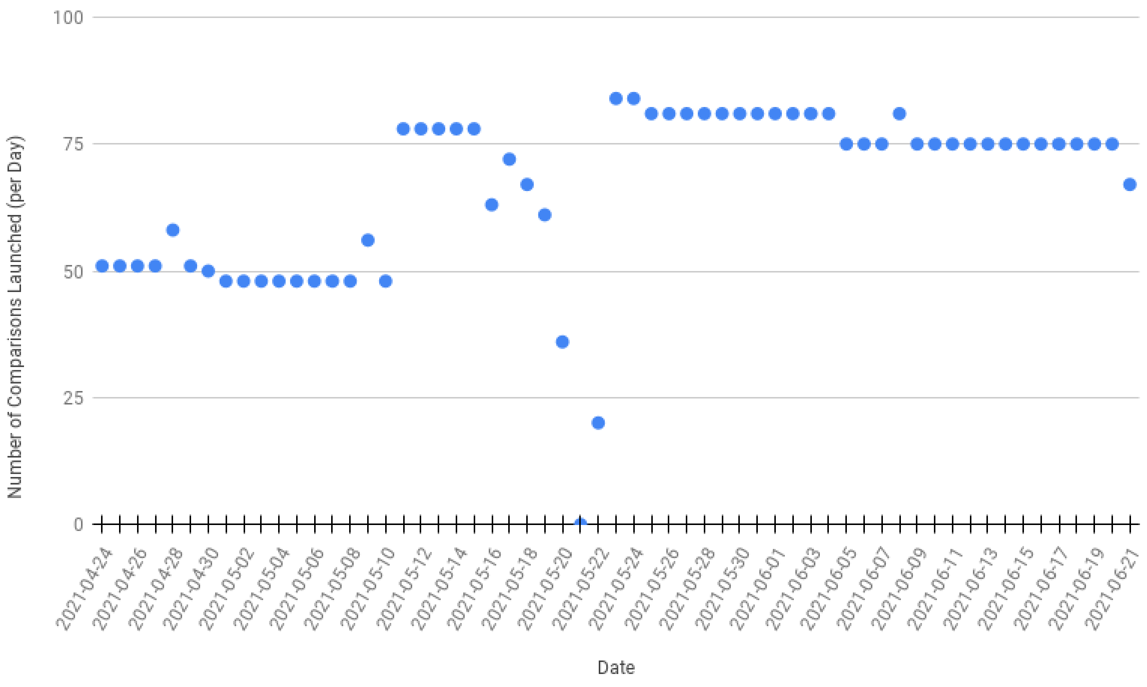


Figure 5.2: Number of Daily Comparisons Launched per Day

A launch of a comparison means that the controller takes a comparison definition and puts two or more jobs that represent the outlets being compared in the correct queue. Scraping bots are then responsible for taking these jobs from the queue.

There are several reasons for the volatility in Figure 5.2. For example, on May 10 2021, several new comparisons were defined, resulting in an increase in comparison runs.

Around May 18, it was discovered that an unusually high number of jobs requesting to scrape the mobile website of Kayak were never processed. To prevent building up more unprocessed jobs in the queue, the launch of new comparisons was stopped until the bug was resolved. Other variations in Figure 5.2 can also be explained by the addition and removal of comparison definitions, resulting in an increase or decrease in comparison runs.

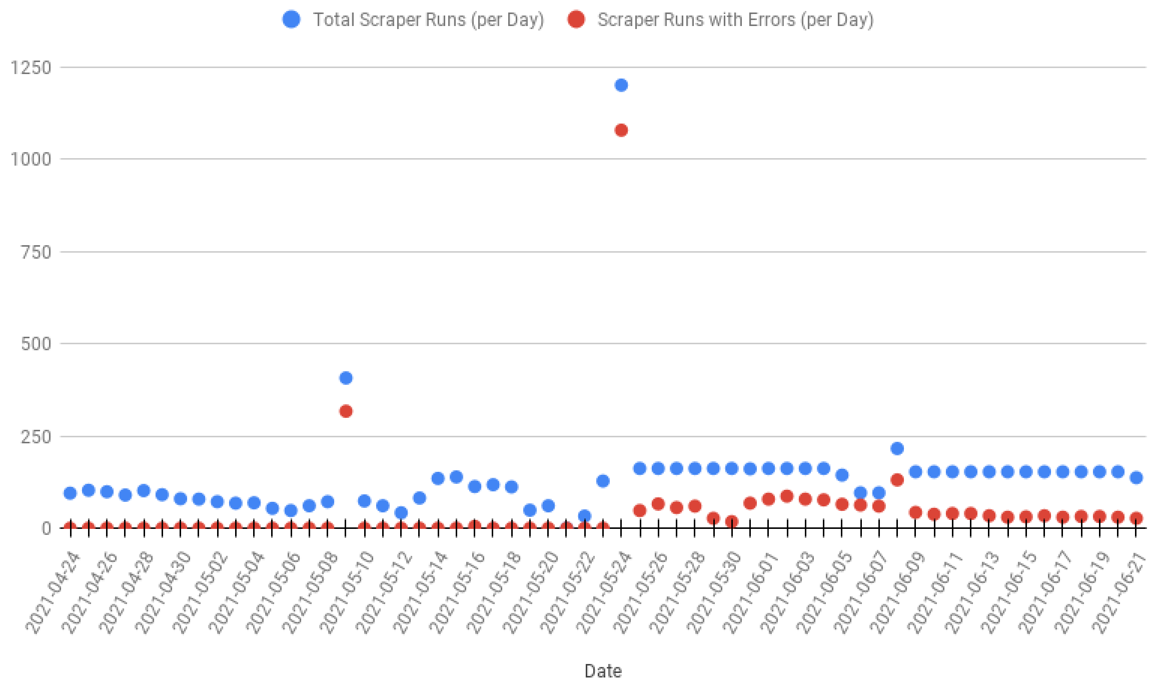


Figure 5.3: Number of Daily Bot Extractions Launched per Day

In Figure 5.3, the number of scraper runs succeeded versus the number of scraper runs that did not finish successfully are shown. To be clear, a scraper run represents the launch of a bot. A scraper run can either finish successfully, return one or more offers, or it can fail.

When one scraper bot fails, the other bots in the same comparison will also fail. When a bot is synchronized with other bots in the comparison, and another bot fails or takes too much time to complete a certain phase, then the bot will throw an exception FATAL ERROR: $synchronizationOnSearchSeconds$ > MAX_SYNCHRONIZATION_TIME, resulting in a 'snowball effect' in the number of failed scraper runs.

What can be seen is that there are no errors from April 24, then around May 9 there is a spike in errors. Upon inspection, it was found that there was a bug in the bot responsible for website scraping (in the production environment setting the locale resulted in different behavior than in development). There is also a spike in the total number of scraper runs, since many solutions (which were tested by launching more comparisons) to resolve the bug were tried.

The spike around May 24 was related to the resolution of several application scraper bugs, halting of application scraper bots, and the consecutive congestion of unprocessed scraping jobs in the queue that resulted in a large number of scraper runs to fail (see also Figure 4.7 for an illustration of such a situation).

In general, during the entire experiment, the web scraping bots continuously encountered errors. The most popular error was ‘element not found error’; it was impossible to avoid this error type all-together, even though a best effort was made to update bots regularly to reflect changing websites. Another category of errors was ‘connection failed’, which was caused by an inactive residential proxy.

5.3 Analysis

What follows is an analysis of the data that has been collected. Tables 5.1 and Table 5.2 give an overview all comparisons executed, grouped per company.

		Data set		Data collection			Price Difference
Company	Comparison	dept.	orig–dest	start	end	#	price difference range
Expedia	app/web	07-01	bru-ams	05-17	06-23	273	–
		08-10	ams-arn	05-17	06-23	962	–
		08-18	opo-bru	05-17	06-23	1045	–
	FR/DE	07-01	bru-ams	05-17	06-23	495	€13
		08-01	bru-ams	05-17	06-23	560	–
AirFrance	app/web	07-01	fra-cdg	04-24	06-21	448	€13
		08-01	fra-cdg	04-24	06-21	500	–
		08-09	vie-ams	04-24	06-21	513	–
	FR/DE	07-01	fra-cdg	05-25	06-21	500	€1-€5
Opodo	app/web	07-01	fra-cdg	05-25	06-21	878	€11.99-€24.01
		08-01	fra-cdg	05-25	06-21	966	€8.99-€21.01
		08-23	cgn-prg	05-25	06-21	152	–
		08-18	opo-bru	05-25	06-21	860	€10.99-€45.01
	FR/DE	07-01	fra-cdg	05-25	06-21	882	€4.27-€53.96
Kayak	app/web/mobile	08-18	opo-bru	06-15	06-30	5036	–
	app/web	08-07	mad-fco	06-15	06-30	4919	–
		08-13	ber-bcn	06-15	06-30	9623	–
EuroWings	app/web	07-11	ams-ham	04-08	05-31	161	–
	FR/DE	07-11	ams-ham	04-08	05-31	189	–

Table 5.1: Overview of Flight Price Comparisons

		Data set		Data collection			Price Difference
Company	Comparison	check-in	location	start	end	#	
Booking.com	app/web	07-01	Geel	06-08	06-29	1101	–
		07-01	Herentals	06-08	06-29	654	–
		08-01	Herentals	06-08	06-29	799	–
	FR/DE	07-01	Geel	06-08	06-29	1720	–
		08-01	Geel	06-08	06-29	2104	–

Table 5.2: Overview of Hotel Price Comparisons

In Table 5.1 and Table 5.2, the first column displays the company name. The second column contains the outlets that are compared, for example FR/DE means that the French desktop website (e.g., eurowings.fr) is compared to the German desktop website (e.g., eurowings.de); app/web/mobile means that the mobile application vs the desktop website vs the mobile browser are compared. The third and the fourth columns (grouped under 'Data set') explain which search parameters have been used; for flight tickets, this is departure date and airport origin-airport destination; for hotel bookings, this is check-in date and hotel location. Under 'Data collection', three columns are grouped that display the start and end date of offer extraction, and the number of offers that have been collected in this period from *all* outlets involved in the comparison. Finally, when a comparison is finished, either a price difference is found or no price difference is found. When a price difference is found, multiple outcomes are possible. For example, €13 specifies that if a price difference is found, the difference is always the same: €13. A price difference of €8.99-€45.01 means that the difference is between €8.99-€45.01, without saying which outlet is more expensive (one outlet could be €8.99 cheaper or €8.99 more expensive than another outlet). When a price difference is detected, it does *not* mean that the price of *every* offer is different; rather it means that at least one price difference was detected between two equivalent offers from different outlets.

5.3.1 Air France

Android Application versus Desktop Website

For Air France, its mobile application was compared with its desktop website (airfrance.de).

Flights were searched departing from FRA flying to CDG on 2021-07-01. Measurements were performed from April 24 to June 21.

For flight AF1019 no price differences were found.

For flight AF1619, one price difference was found; see Figure 5.4.

On 2021-06-21 01:00, the price found on the website was 95 EUR, the price found on the application was 108 EUR.

Flights were also searched departing from FRA flying to CDG on 2021-08-01. No price differences were found.

In addition, flights were searched departing from VIE flying to AMS on 2021-08-09. No price differences were found for flights KL1840, KL1838, and KL1844.

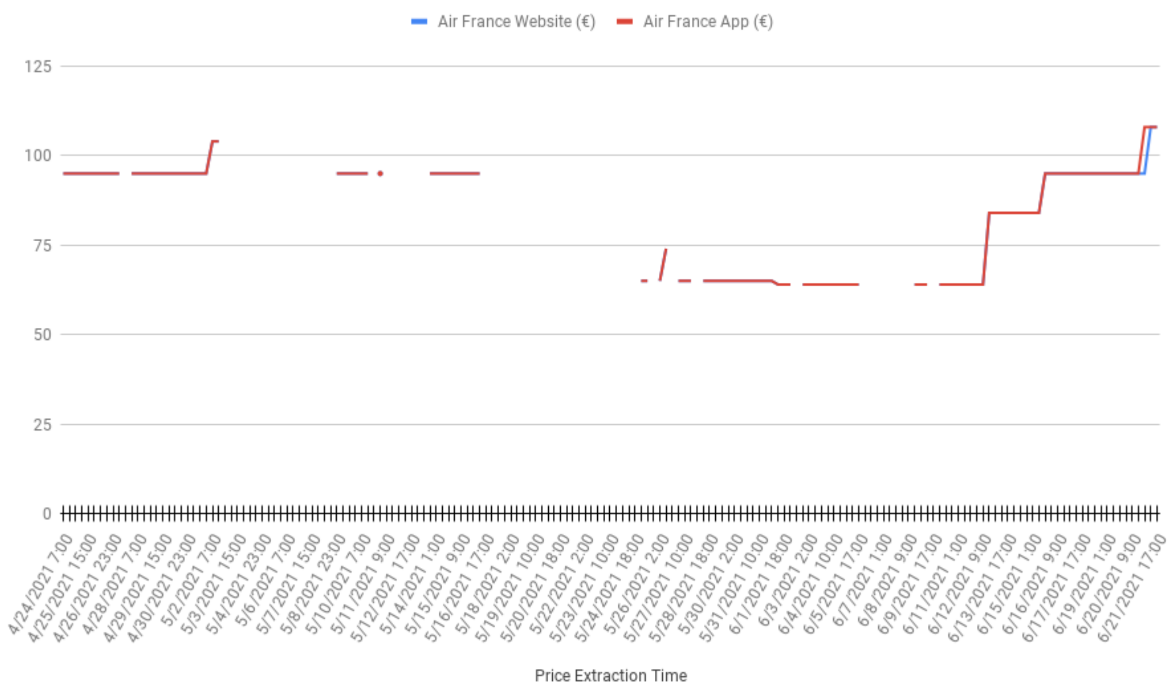


Figure 5.4: Flight AF1619, price on Air France mobile application vs website.

In Figure 5.4, as in all figures below, note that only pairs of data from two outlets are shown. When there is only one data point for one outlet, nothing is shown in the graph.

German Desktop Website versus French Desktop Website

To compare the German website of Air France (airfrance.de) versus the French website of Air France (airfrance.fr), search queries were launched again for flights departing from FRA and flying to CDG on 2021-07-01. Measurements were executed from May 25 to June 21.

Again, the prices for flights AF1019 and AF1619 are displayed:

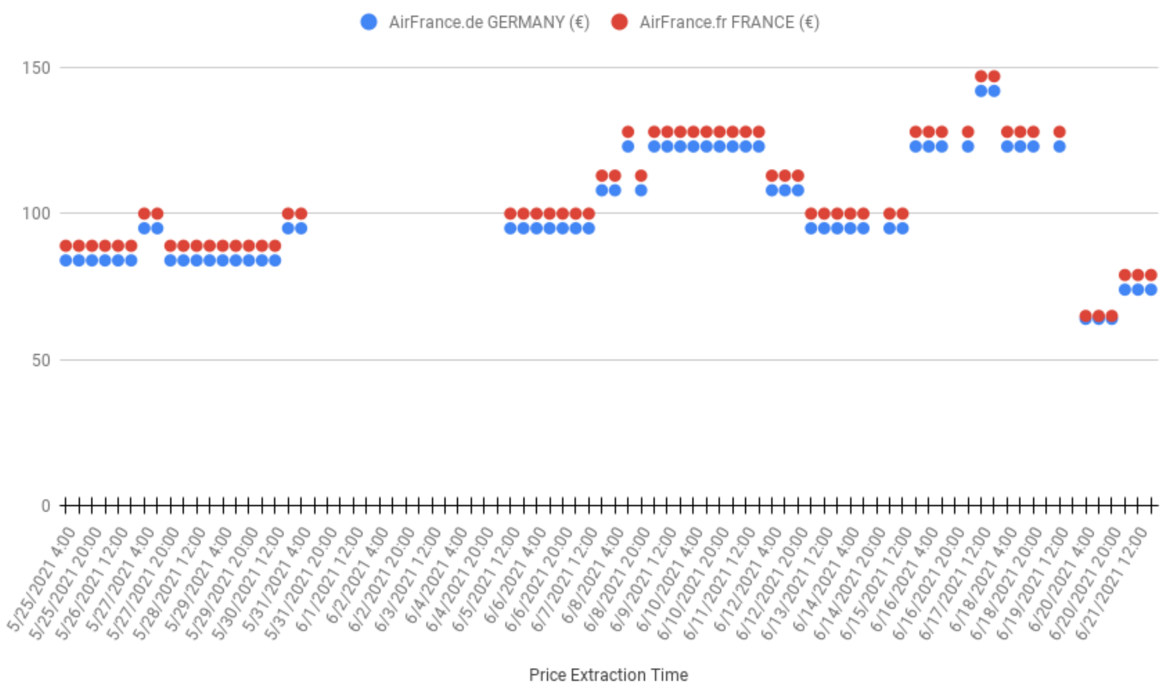


Figure 5.5: Flight AF1019, price on airfrance.de vs. airfrance.fr website

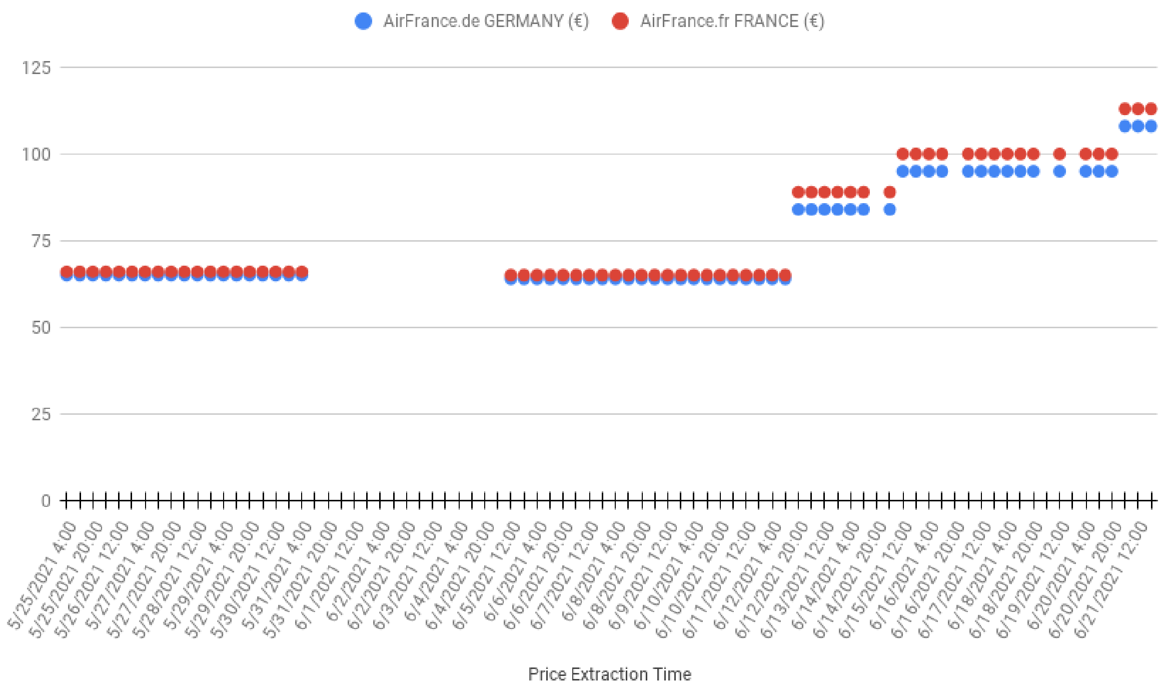


Figure 5.6: Flight AF1619, price on airfrance.de vs. airfrance.fr website.

As can be seen, for both flight AF1019 and flight AF1619, there are price differences between airfrance.fr and airfrance.de.

For flight AF1019, the difference is almost always constant, except around June 20 the difference is smaller. We could not find an explanation for the price differences; the departure airport and the arrival airport are the same, as is the departure time. The difference is more than the different in VAT rate, which is 19% in Germany and 20% in France.

Another indication showing that price difference is not caused by tax difference, is flight AF1619. First, there is almost no price difference (only one Euro) between the two websites; the price difference increases to five Euro when the price extraction time is getting closer to the departure flight date.

5.3.2 EuroWings

Price measurements were executed from April 8 until May 31.

Android Application versus Desktop Website

For the airline company EuroWings, the Android application of EuroWings was compared with the website of EuroWings (eurowings.de). One-way flights from AMS to HAM leaving on 2021-07-11 were queried. No price differences were found for any flight offers displayed.

In addition, flights were searched from CGN to LON leaving on 2021-08-12. However, because of a bug in the EuroWings Mobile Application scraping bot where flights were returned for September 12 instead of August 12, the results from this comparison were ignored.

German Desktop Website versus French Desktop Website

The EuroWings German desktop website (eurowings.de) was also compared with the French website (eurowings.fr). One-way flights were searched from AMS to HAM departing on 2021-07-11. No price differences were found.

5.3.3 Opodo

For all comparisons, measurements started on May 25 and ended on June 21.

Android Application versus Desktop Website

The Android application of Opodo and the website (opodo.de) were compared for one-way flights from FRA to CDG leaving on 2021-07-01. In Figure 5.7, most of the time the price of the application is higher. This is confirmed by taking averages. The average price of a flight ticket for flight LH1052 on the application is €137.4 and the average price on the website is €132.9.

Price differences were also found when searching flights from FRA to CDG leaving on 2021-08-01.

In addition, flights were compared matching origin airport CGN, destination airport PRG and departure date 2021-08-23. There was only one flight available on each outlet: flight EuroWings EW9772. No price differences were found between the application and website for flight EW9772.

Flights leaving from OPO flying to BRU on 2021-08-18 were also compared. From Figure 5.8, it may not be clear whether the price is higher on the application or on the website. The average price on the mobile application is EUR 134.64, while the average price on the desktop website is EUR 131.85.

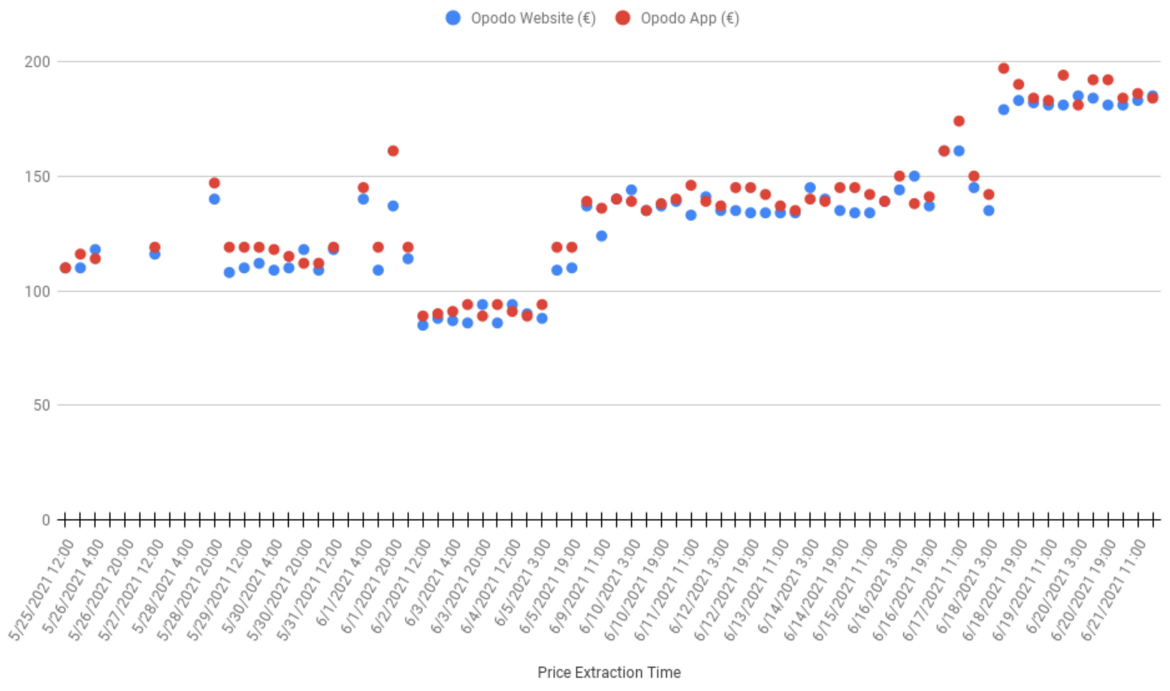


Figure 5.7: Flight LH1052, price on Opodo mobile application vs website

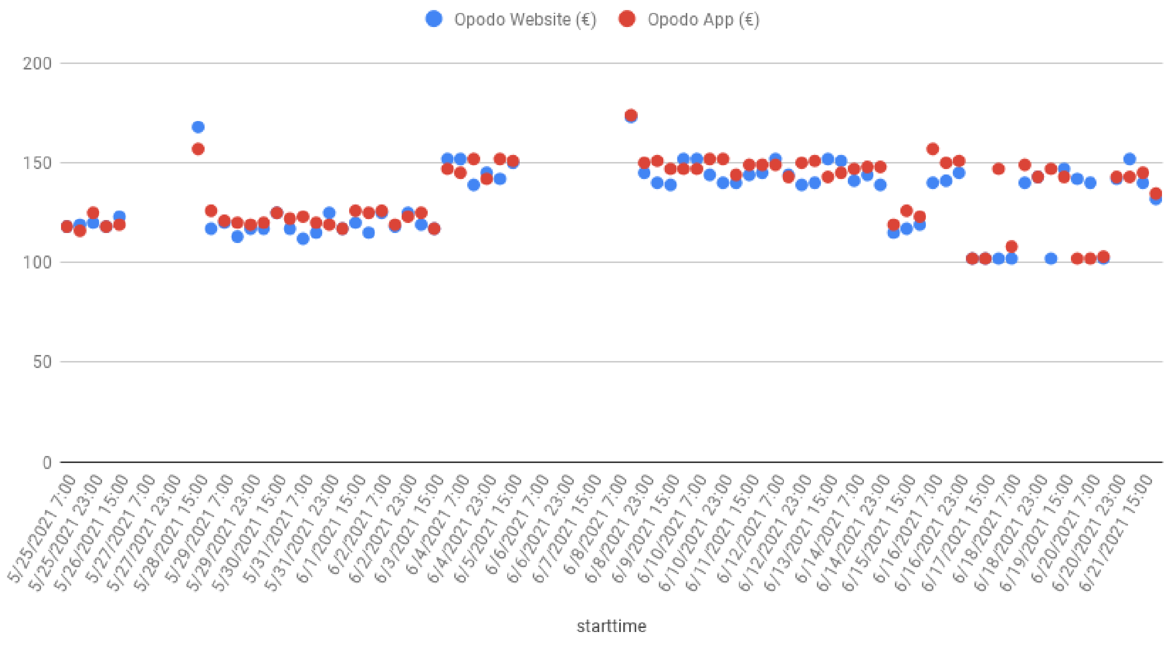


Figure 5.8: Flight SN3810, price on Opodo mobile application vs website

German Desktop Website versus French Desktop Website

The German website of Opodo (opodo.de on locale de_DE) was compared against the French website (opodo.fr on locale fr_FR). One-way flights were searched from FRA to

CDG departing on 2021-07-01. In Figure 5.9, the price evolution of flight LH1052 is shown. It can be seen that during almost the entire measurement time (from May 25 to June 21), the price on the opodo.fr website is higher than on the opodo.de website.

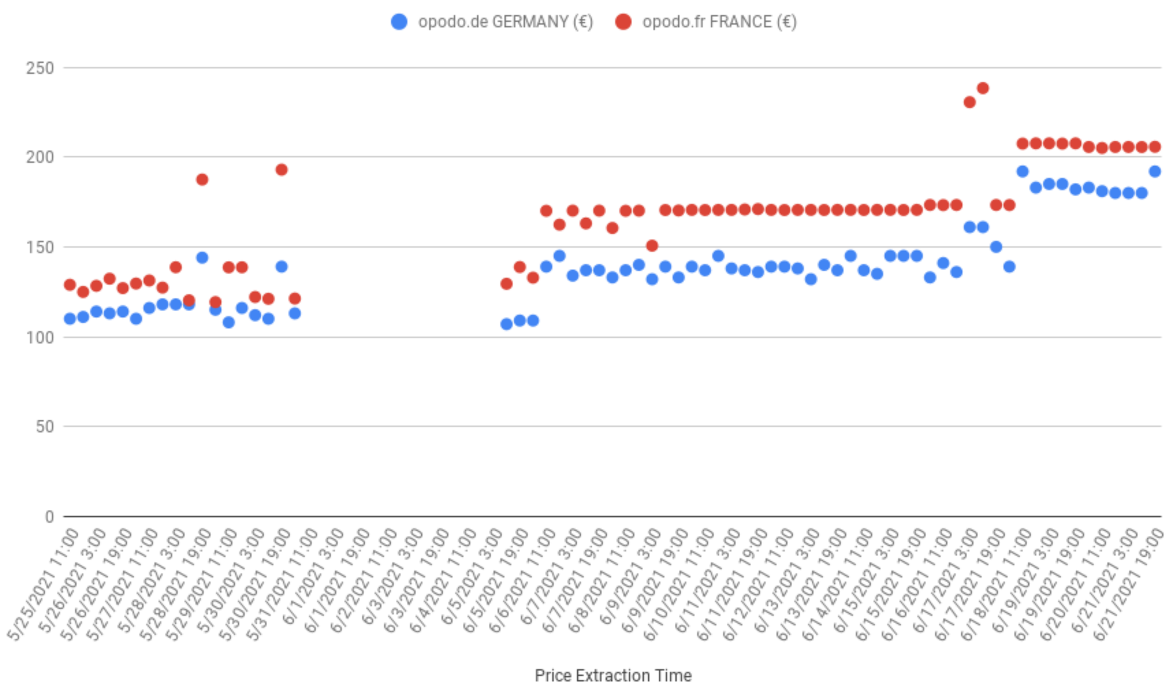


Figure 5.9: Flight LH1052, price on opodo.de vs. opodo.fr website

5.3.4 Expedia

For Expedia measurements were executed from May 17 until June 23.

Android Application versus Desktop Website

Expedia's mobile Android application versus its website (expedia.de) was compared for flights from BRU to AMS leaving on 2021-07-01. No price differences were detected.

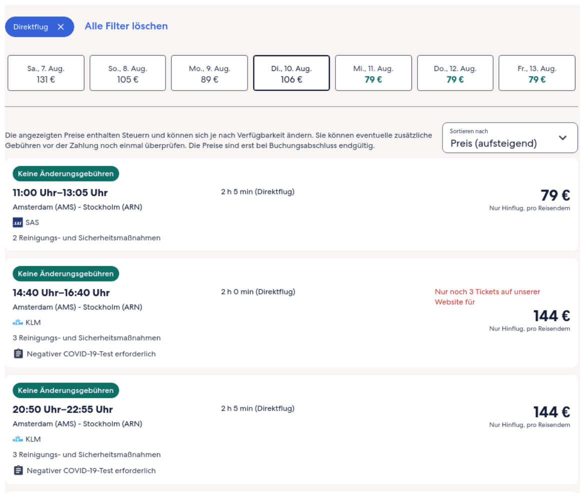
Comparison was also done for flights from AMS to ARN leaving on 2021-08-10. No price differences were detected.

As a side note, no price differences between individual flight tickets were found. However, sporadic differences in the list of flight offers were found. In Figures 5.10 and 5.11, the offer lists are shown for the website and mobile application of Expedia. Offers were extracted on 2021-06-23. On the website, there were a total of five offers, while on the mobile application, there were only a total of four offers.

Prices were also compared for flights from OPO to BRU leaving 2021-08-18. No price differences were found.

French Desktop website versus German Desktop website

Expedia's French website (expedia.fr) versus its German website (expedia.de) was compared for flights from BRU to AMS leaving on 2021-07-01.


Direktflug ✕ Alle Filter löschen

Sa., 7. Aug.	So., 8. Aug.	Mo., 9. Aug.	Di., 10. Aug.	Mi., 11. Aug.	Do., 12. Aug.	Fr., 13. Aug.
131 €	105 €	89 €	106 €	79 €	79 €	79 €

Die angezeigten Preise enthalten Steuern und können sich je nach Verfügbarkeit ändern. Sie können eventuelle zusätzliche Gebühren vor der Zahlung noch einmal überprüfen. Die Preise sind erst bei Buchungsabschluss endgültig.

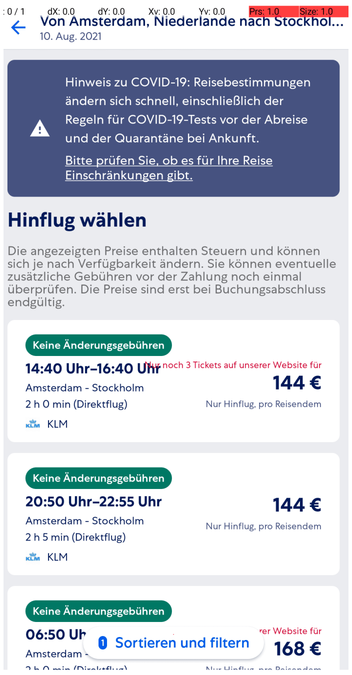
Sortieren nach: Preis (aufsteigend)

Keine Änderungsgebühren
11:00 Uhr–13:05 Uhr 2 h 5 min (Direktflug) **79 €**
Amsterdam (AMS) - Stockholm (ARN) Nur Hinflug, pro Reisendem
SAS
2 Reinigungs- und Sicherheitsmaßnahmen

Keine Änderungsgebühren
14:40 Uhr–16:40 Uhr 2 h 0 min (Direktflug) Nur noch 3 Tickets auf unserer Website für **144 €**
Amsterdam (AMS) - Stockholm (ARN) Nur Hinflug, pro Reisendem
KLM
3 Reinigungs- und Sicherheitsmaßnahmen
Negativer COVID-19-Test erforderlich

Keine Änderungsgebühren
20:50 Uhr–22:55 Uhr 2 h 5 min (Direktflug) **144 €**
Amsterdam (AMS) - Stockholm (ARN) Nur Hinflug, pro Reisendem
KLM
3 Reinigungs- und Sicherheitsmaßnahmen
Negativer COVID-19-Test erforderlich

Figure 5.10: SAS flight shown on Expedia website (extraction: 2021-06-23)


: 0 / 1 dX: 0.0 dY: 0.0 Xv: 0.0 Yv: 0.0 Prs: 1.0 Size: 1.0

Von Amsterdam, Niederlande nach Stockhol...
10. Aug. 2021

Hinweis zu COVID-19: Reisebestimmungen ändern sich schnell, einschließlich der Regeln für COVID-19-Tests vor der Abreise und der Quarantäne bei Ankunft.
Bitte prüfen Sie, ob es für Ihre Reise Einschränkungen gibt.

Hinflug wählen

Die angezeigten Preise enthalten Steuern und können sich je nach Verfügbarkeit ändern. Sie können eventuelle zusätzliche Gebühren vor der Zahlung noch einmal überprüfen. Die Preise sind erst bei Buchungsabschluss endgültig.

Keine Änderungsgebühren
14:40 Uhr–16:40 Uhr Nur noch 3 Tickets auf unserer Website für
Amsterdam - Stockholm **144 €**
2 h 0 min (Direktflug) Nur Hinflug, pro Reisendem
KLM

Keine Änderungsgebühren
20:50 Uhr–22:55 Uhr **144 €**
Amsterdam - Stockholm Nur Hinflug, pro Reisendem
2 h 5 min (Direktflug)
KLM

Keine Änderungsgebühren
06:50 Uh rer Website für
Amsterdam **168 €**

Figure 5.11: SAS flight not shown on Expedia mobile application (extraction: 2021-06-23)

The price evolution of two matching flights (KLM1720 and KLM1732) is displayed in Figures 5.12 and 5.13.

For flight KLM1720, there are no price differences until measurement date June 17, after which price differences start to appear.

For flight KLM1732, there are also no price differences until measurement date June 17, after which price differences appear.

For flights KLM1720 and KLM1732, it can be observed that price differences start appearing when measurement time is getting closer to the flight departure time.

Flights were also compared going from BRU to AMS on 2021-08-01. No price differences were found for any matching flights.

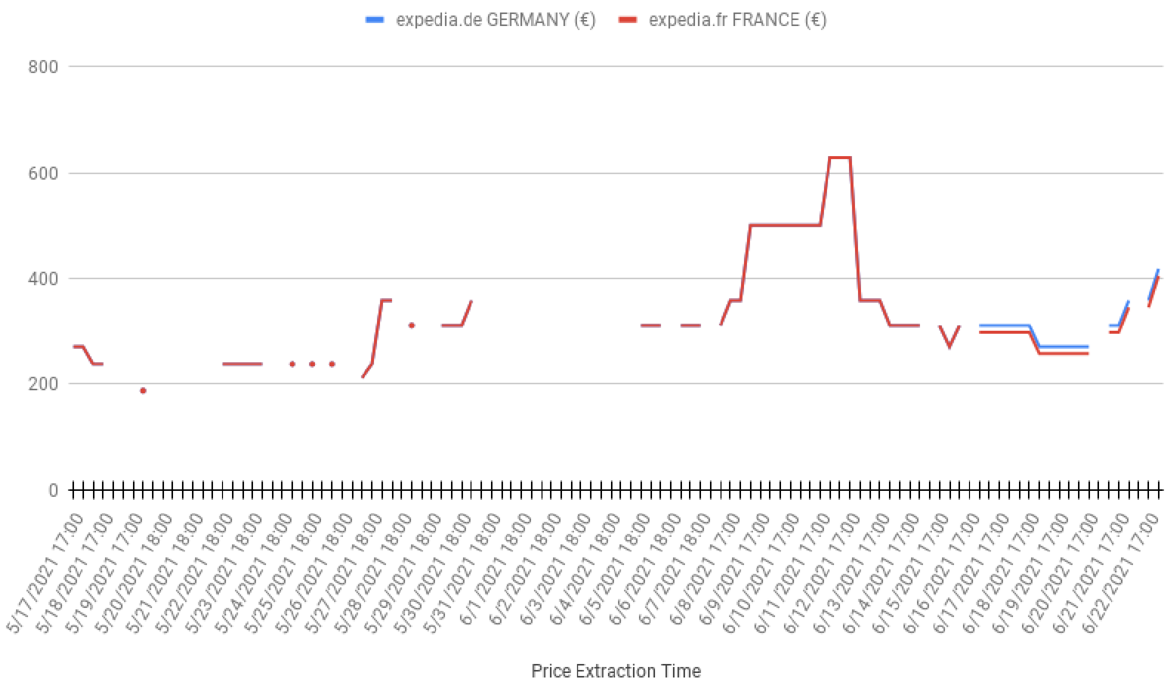


Figure 5.12: Flight KLM1720, price on expedia.de vs. expedia.fr website

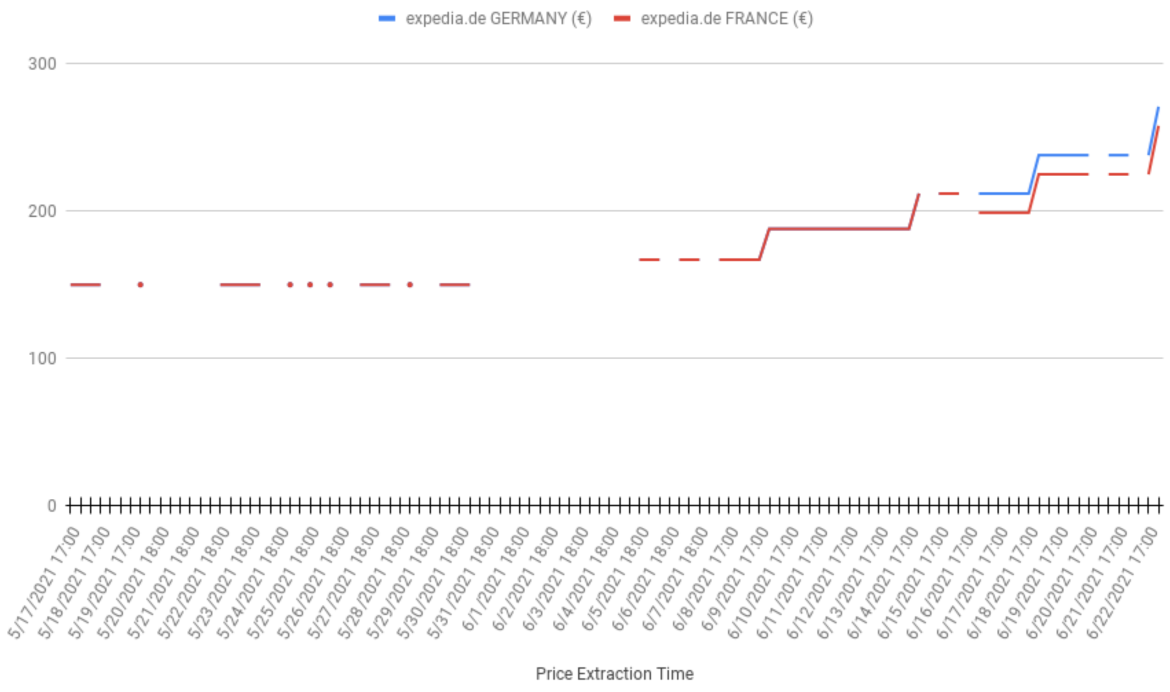


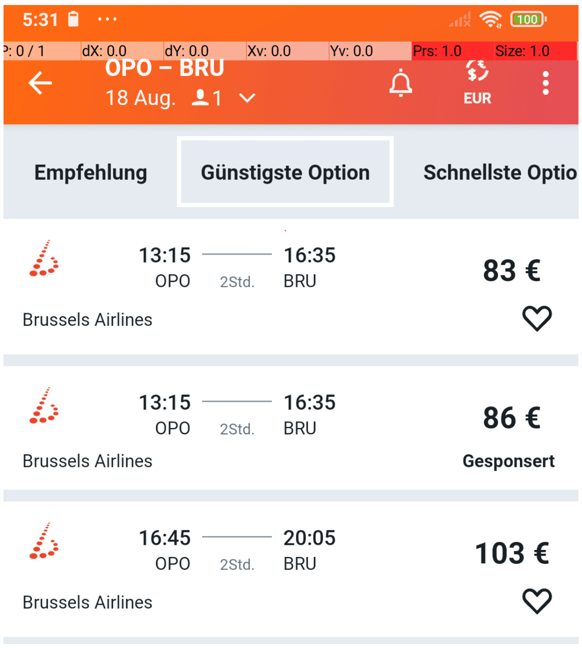
Figure 5.13: Flight KLM1732, price on expedia.de vs expedia.fr website

5.3.5 Kayak

Measurements were done from June 15 to June 30.

Android Application versus Desktop Website versus Mobile Website

Three different outlets are compared: the mobile website (kayak.de), the mobile application, and the desktop website (kayak.de). Flights are searched leaving from OPO and going to BRU on 2021-08-18. No price differences were found. Note that for flight SN3812, at first the scraping bots found price differences. However, upon inspection of the screenshots taken by the bots, it became clear that these were false positives. See Figure 5.14, where each outlet displays flight SN3812 two times. This 'problem' is not found on the desktop website.



5:31 ···

P: 0 / 1 dX: 0.0 dY: 0.0 Xv: 0.0 Yv: 0.0 Prs: 1.0 Size: 1.0

OPO – BRU
18 Aug. 1

EUR

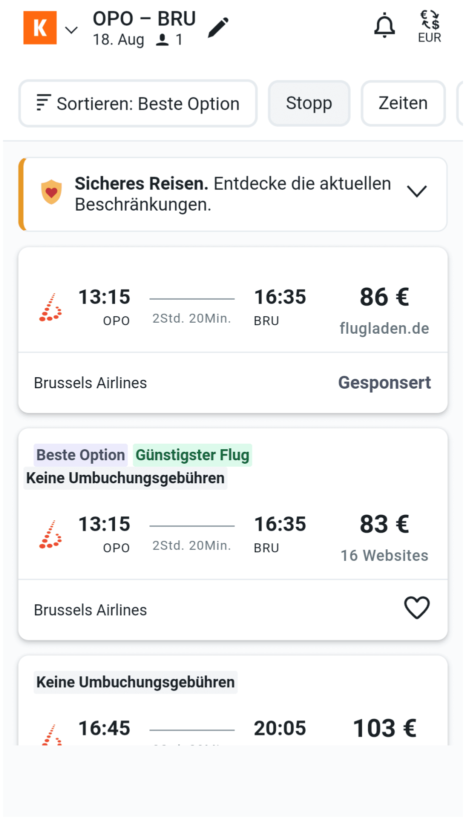
Empfehlung | Günstigste Option | Schnellste Optio

13:15 OPO — 2Std. — 16:35 BRU — 83 €
Brussels Airlines

13:15 OPO — 2Std. — 16:35 BRU — 86 €
Brussels Airlines — Gesponsert

16:45 OPO — 2Std. — 20:05 BRU — 103 €
Brussels Airlines

(a)



OPO – BRU
18. Aug 1

EUR

Sortieren: Beste Option | Stopp | Zeiten

Sicheres Reisen. Entdecke die aktuellen Beschränkungen.

13:15 OPO — 2Std. 20Min. — 16:35 BRU — 86 €
flugladen.de
Brussels Airlines — Gesponsert

Beste Option Günstigster Flug
Keine Umbuchungsgebühren
13:15 OPO — 2Std. 20Min. — 16:35 BRU — 83 €
16 Websites
Brussels Airlines

Keine Umbuchungsgebühren
16:45 — 20:05 — 103 €

(b)

Figure 5.14: Flight SN3812 appearing twice with different prices on Kayak's (a) mobile application and (b) mobile website.

Android Application versus Desktop Website

The mobile application of Kayak was compared with the Kayak desktop website (kayak.de). Flight departure airport was MAD, destination airport was FCO, and the departure date was 2021-08-07. No price differences were found.

The mobile application versus the desktop website (kayak.de) was also compared for flights from BER to BCN departing on 2021-08-13. No price differences were found.

5.3.6 Booking.com

For Booking.com, hotel room prices were compared. Measurements were done from June 8 to June 29.

Android Application versus Desktop website

Hotel rooms located in the city of Geel were queried with a check-in date of 2021-07-01 and with a checkout date of 2021-07-02.

No price differences between equivalent hotel rooms were found.

Hotel rooms located in the city of Herentals were also queried with a check-in date of 2021-07-01 and with a checkout date of 2021-07-02. No price differences were found.

In addition, hotel rooms located in the city of Herentals were queried with a check-in date of 2021-08-01 and with a checkout date of 2021-08-02. Again, no price differences were found.

German Desktop Website vs French Desktop Website

The German desktop website (booking.de) was compared with the French desktop website (booking.fr). Hotel rooms were searched in the city of Geel with check-in date of 2021-07-01 and checkout-date of 2021-07-02. No price differences were found.

The same comparison was done for check-in date 2021-08-01 and checkout-date of 2021-08-02. No price differences were found.

Chapter 6

Conclusion

The motivation to start this thesis were rumors about differences in flight ticket prices between mobile applications and websites. These rumors, however, were based on a limited amount of manual observations that may be prone to noise, such as price fluctuations.

Therefore, we wanted to create a system to assist in price difference verification. In this thesis the problem of scraping data using bots from multiple outlets simultaneously was addressed.

We focused on creating a system that can extract offers from any type of outlet, including but not limited to mobile applications, desktop websites, and mobile websites. A design is proposed that can run on a cluster of heterogeneous machines and can host an unlimited number of bots. Each scraping bot that is part of a comparison, can synchronize to other bots in the comparison at several phases of the scraping process. The end result of a comparison is a collection of offers matching a given query, extracted from each outlet simultaneously.

Using an implementation of our design, a price comparison study was conducted. The study compared prices from six companies in the travel industry. Flight ticket and hotel room prices on mobile applications were compared with prices on desktop and mobile websites.

After two months of running our system with a varied set of comparison definitions, more than 5,000 comparisons were executed and more than 55,000 offers were collected.

Price differentiation was discovered for the majority of comparisons on Opodo.com. Flight ticket prices on the mobile application were higher than the prices of the same flight tickets found on the desktop website, as was its French website compared to its German website. For Air France, we found that the flight ticket price on its French desktop website was sometimes more expensive than on its German website; this difference increased when prices were queried closer to the departure date of the flight in question. For Expedia, price differences were also found between the French and German website when the measurement date approached the departure date.

6.1 Future work

Our price comparison study was only limited to a small number of comparison definitions, one reason being that offers had to be matched manually by an analyst instead of the system. The determination of equivalency of items from different outlets did not make it as a feature of the system; automatic matching based on equivalency definitions and/or fuzzy matching, could be added as future work. Another useful extension to the system would be dynamic browser and mobile application fingerprints, which would allow the study of the effects of IP address, browser state, and other properties on pricing.

Future work could involve querying for a wider range of data, for example by running the system for a period of three months while always looking for flights departing two days after the measurement date. This was not possible in our study, because it had been conducted during the corona crisis and flights were randomly canceled. In addition, our system is not limited to flight tickets and hotel reservations; any type of offer could be retrieved simultaneously on any type of outlet available online.

Bibliography

[ATSV06] Jeannie R. Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In Atul Adya and Erich M. Nahum, editors, *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 301–314. USENIX, 2006.

[BMW17] Timo Bertsch, Joram Markert, and Sebastian Wiesendahl. Research project: Price Discrimination, 2017. Technische Hochschule Köln.

[BN84] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[BPSAP$^+$19] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[CMW15] Le Chen, Alan Mislove, and Christo Wilson. Peeking beneath the hood of uber. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 495–508, 2015.

[DCAT12] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Trans. Internet Technol.*, 12(1), July 2012.

[Geh19] Vijay Gehlot. From petri NETS to colored petri NETS: A tutorial introduction to NETS based formalism for modeling and simulation. In *2019 Winter Simulation Conference, WSC 2019, National Harbor, MD, USA, December 8-11, 2019*, pages 1519–1533. IEEE, 2019.

[HSL$^+$14] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 305–318, 2014.

[HTWH18] Thomas Hupperich, Dennis Tatang, Nicolai Wilkop, and Thorsten Holz. An empirical study on online price differentiation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, pages 76–83, 2018.

[JKV19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. In Kazue Sako, Steve Schneider, and

Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 586–605, Cham, 2019. Springer International Publishing.

[KHT08] Milly Kc, Markus Hagenbuchner, and Ah Chung Tsoi. A scalable lightweight distributed crawler for crawling with limited resources. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and International Conference on Intelligent Agent Technology - Workshops, 9-12 December 2008, Sydney, NSW, Australia*, pages 663–666. IEEE Computer Society, 2008.

[MGEL13] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Crowd-assisted search for price discrimination in e-commerce: first results. In *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, pages 1–6, 2013.

[MPdF18] Sunil Kumar Mohanty, Gopika Premsankar, and Mario di Francesco. An evaluation of open source serverless computing frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 115–120, 2018.

[ON19] Bogdan Oancea and Marian Necula. Web scraping techniques for price statistics – the romanian experience. *Statistical Journal of the IAOS*, 35:1–10, 10 2019.

[SR15] Felipe Sierra and Anthony Ramirez. Defending your android app. In *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, RIIT '15, page 29–34, New York, NY, USA, 2015. Association for Computing Machinery.

[TEBS12] Jakob G. Thomsen, Erik Ernst, Claus Brabrand, and Michael I. Schwartzbach. Webself: A web scraping framework. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering - 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, volume 7387 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2012.

[Tro94] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, September 1994.

[Uzu20] Erdinç Uzun. A novel web scraping approach using the additional information obtained from web pages. *IEEE Access*, 8:61726–61740, 2020.

[VNBJ14] Thomas Vissers, Nick Nikiforakis, Nataliia Bielova, and Wouter Joosen. Crying Wolf? On the Price Discrimination of Online Airline Tickets. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, Amsterdam, Netherlands, July 2014.

[vST16] Maarten van Steen and Andrew S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, 2016.

[XWGY19] Yuantao Xie, Wen Wang, Yabo Guo, and Juan Yang. Study on the country risk rating with distributed crawling system. *J. Supercomput.*, 75(10):6159–6177, 2019.

[YJHC18] Feng Ye, Zongfei Jing, Qian Huang, and Yong Chen. The research of a lightweight distributed crawling system. In Shaowen Yao, Zhi Jin, Xiaohui Cui, Bing Luo, Junfeng Wang, and Zhengtao Yu, editors, *16th IEEE International Conference on Software Engineering Research, Management and Applications, SERA 2018, Kunming, China, June 13-15, 2018*, pages 200–204. IEEE Computer Society, 2018.

[YY20] Zhiju Yang and Chuan Yue. A comparative measurement study of web tracking on mobile and desktop environments. *Proceedings on Privacy Enhancing Technologies*, 2020(16):24 – 44, 2020.

Appendices

Appendix A

Implementation

In this chapter, the final system as it was developed is explained.

A.1 Analyst's Guide

First, the system will be explained from the viewpoint of an analyst. Then, a more detailed explanation will be given for the persons maintaining the system.

Of interest to the analyst is how she can run comparisons and collect pricing data. To understand how she can use the system, she must understand the data model which is organized as follows:

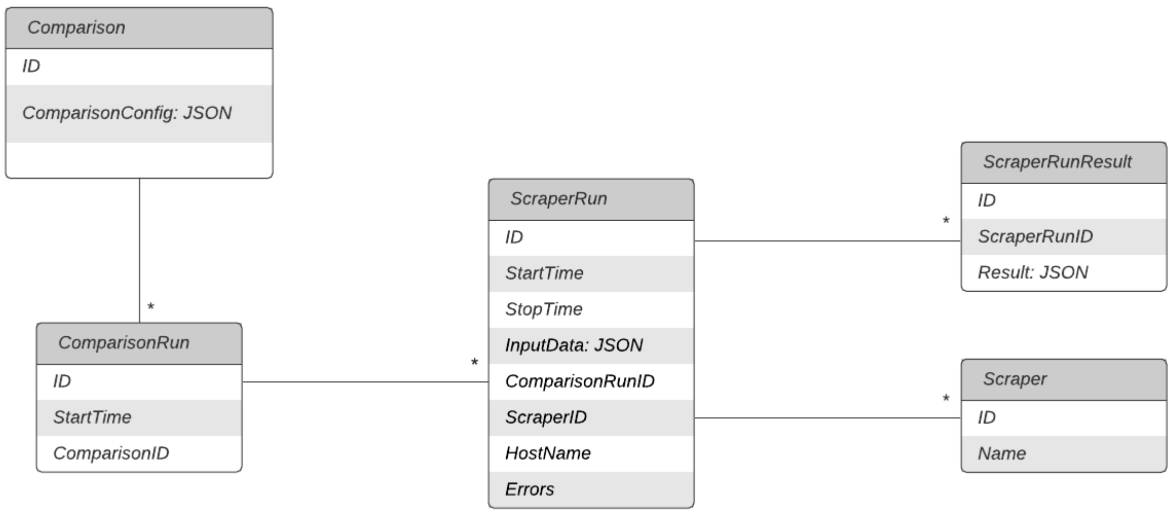


Figure A.1: Entity Relationship Diagram

To understand A.1, know that the basis from which scraping bots start, is a comparison. The table 'comparisons' defines all comparisons that are automatically run by the system at regular intervals (e.g., three times per day - every eight hours). An example of a comparison:

Listing A.1: Example of a Comparison

```
{
    "scrapers": [
        {
            "params": {
                "useRealDevice": "true"
            },
            "scraperClass": "KayakAppScraper"
        },
        {
            "params": {},
            "scraperClass": "KayakWebScraper"
        },
        {
            "params": {},
            "scraperClass": "KayakMobileBrowserScraper"
        }
    ],
    "inputData": {
        "origin": "BRU",
        "destination": "AMS",
        "departureDate": "2021-07-02"
    }
}
```

In the comparison definition in listing A.1, three outlets of the flight comparison company 'Kayak' are compared:

- Its mobile application
- Its desktop website
- Its mobile website

For each outlet, the same input data is entered, namely, 'search for all flights from BRU to AMS that depart on July 2, 2021'. Note that IATA codes are preferred instead of more ambiguous city names.

"An IATA airport code, also known as an IATA location identifier, IATA station code, or simply a location identifier, is a three-letter geocode designating many airports and metropolitan areas around the world, defined by the International Air Transport Association (IATA)." [1]

For each scraper definition, note the 'params' property. The following 'params' are available:

- useRealDevice:
  - applies to mobile application bots;
  - requests a bot to use a real smartphone instead of an emulator.
- lang:
  - applies to desktop browser bots;
  - 'fr' or 'de';

[1]https://en.wikipedia.org/wiki/IATA_airport_code

  - sets a browsers' locale to fr_FR or de_DE and switches to the .fr or .de version of a website.

- proxy:
  - applies to desktop browser bots;
  - forces a browser to use a proxy.
- useRandomProxy:
  - applies to desktop browser bots;
  - when set, a scraper bot will randomly choose a proxy from its proxies.json file.

After adding a new entry in the comparison table, the comparison will be picked up immediately by the system on the next scheduled scraping.

When the scheduler fires, all entries in the comparison table will be processed sequentially. A 'comparison run' represents the running of a comparison at a given start time.

A comparison run will be split in two or more scraper runs, which are instances of a scraper (which can be chosen from the *Scraper* table).

A scraper run has a start time, which depends on the time that a bot picks up the scraping job from the queue; it contains the input data that will be given to the bot; the host name which is the machine hosting the bot; and the possible errors that a bot encountered.

When a scraper run is finished, every offer found on a given outlet will be stored in "scraperRunResult". The "result" column is a JSON object that represents one offer could look like:

Listing A.2: Example of a scraperRunResult

```
{"price":"151 EUR","departureTime":"10:25","arrivalTime":"11:25",
"origin":"BRU","destination":"AMS","airline":"KLM","screenshot":
"https://scraperbox.be/screenshots/ExpediaAppScraper-1616455728862.png"}
```

In this offer, a flight offer from Brussels to Amsterdam is displayed. Note the inclusion of a screenshot in every offer for verification.

Moreover, note that the system is only fetching data; the matching of data has to be done manually. When all scraper runs are finished, the analyst can query the data and look for price differentiation. It is up to the analyst to match product offers among outlets. The definitions in section 3.4 could be used for this.

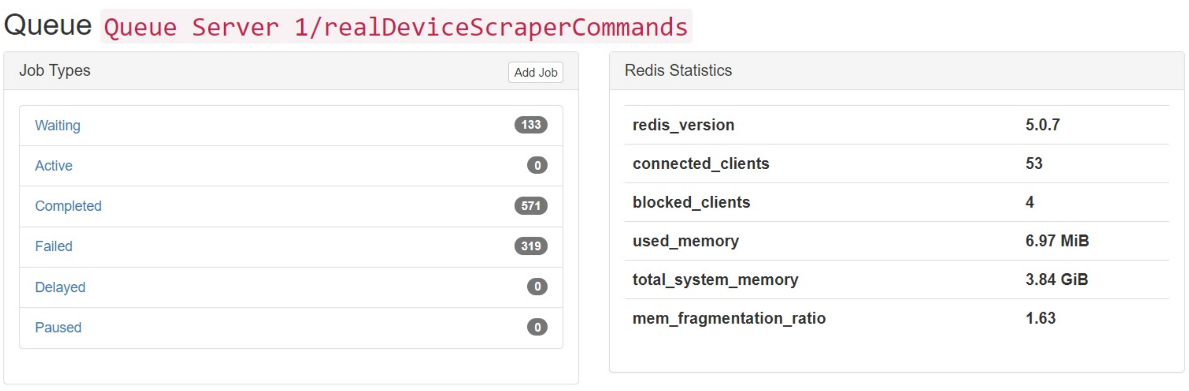
A.2 Developer's Guide

Here, the system will be described from a developer's point of view.

As mentioned in the design chapter, the system consists of one controller and one or more scraping bots.

A.2.1 Interaction via Queues

The controller interacts with bots via centralized queues.


Queue Queue Server 1/realDeviceScraperCommands

Job Types

Job Types	
Waiting	133
Active	0
Completed	571
Failed	319
Delayed	0
Paused	0

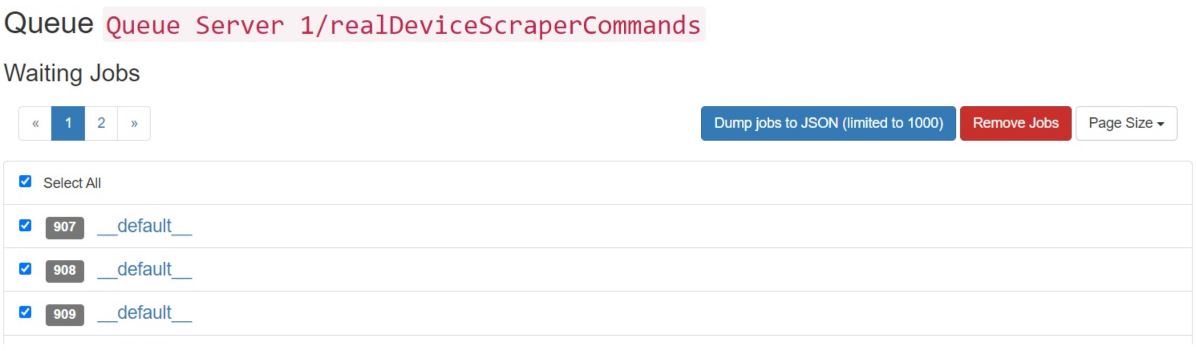
Redis Statistics

redis_version	5.0.7
connected_clients	53
blocked_clients	4
used_memory	6.97 MiB
total_system_memory	3.84 GiB
mem_fragmentation_ratio	1.63

Figure A.2: Queue for processing smartphone scraping jobs

In Figure A.2, an overview is displayed of the queue (implemented by BullMQ[2]) which is responsible for the processing of jobs destined for smartphone scraping bots. As can be seen, there are 133 jobs sent by the controller and waiting in the queue, which probably means that no bot is pulling jobs from this queue, probably because a bot has crashed.

On the right side of Figure A.2, we can see that the queue is based on one Redis[3] in-memory database, which is shared by all scraping bots and the controller.


Queue Queue Server 1/realDeviceScraperCommands

Waiting Jobs

Select All

907 __default__

908 __default__

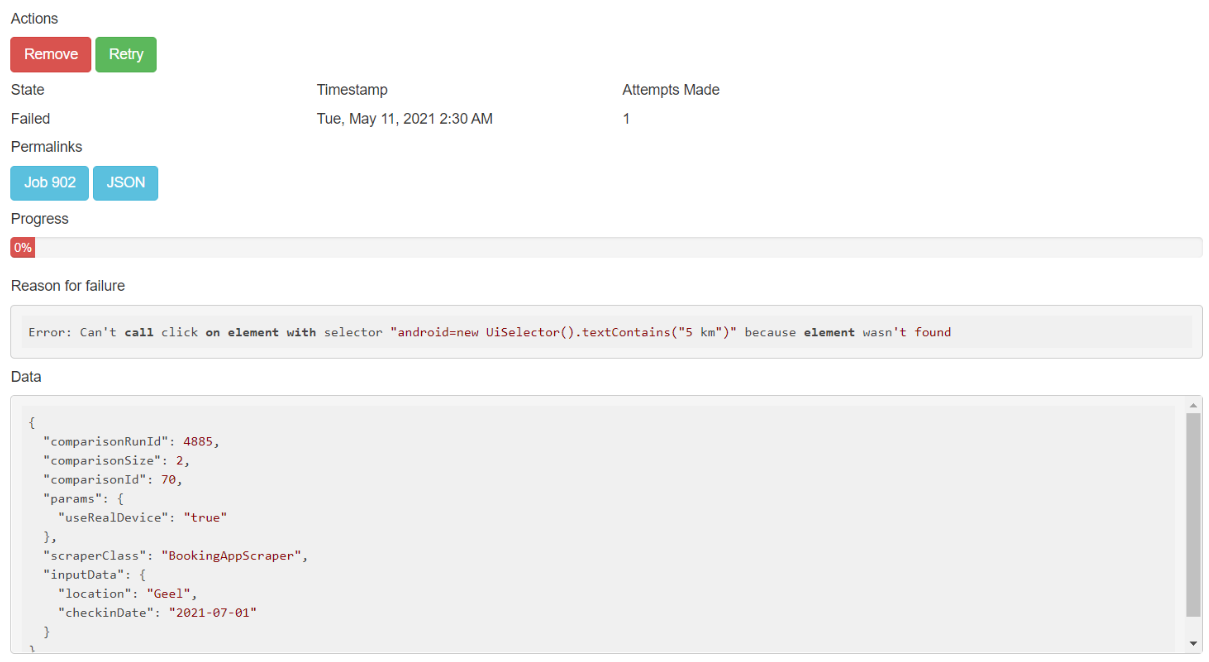
909 __default__

Figure A.3: Jobs waiting to be processed

In Figure A.3, when it has been found out that a bot has crashed, it may be decided to remove all outstanding jobs. Since the jobs in a comparison are synchronized and should run simultaneously, the solution is to repair any crashed bots and remove any outstanding jobs from the queue before starting these bots again.

It is also possible to 'retry' jobs. When during the processing of a job an exception has been encountered, the job will be marked as 'failed':

[2]https://github.com/OptimalBits/bull
[3]https://redis.io/


Actions

Remove Retry

State	Timestamp	Attempts Made
Failed	Tue, May 11, 2021 2:30 AM	1

Permalinks

Job 902 JSON

Progress

0%

Reason for failure

```
Error: Can't call click on element with selector "android=new UiSelector().textContains("5 km")" because element wasn't found
```

Data

```
{
  "comparisonRunId": 4885,
  "comparisonSize": 2,
  "comparisonId": 70,
  "params": {
    "useRealDevice": "true"
  },
  "scraperClass": "BookingAppScraper",
  "inputData": {
    "location": "Geel",
    "checkinDate": "2021-07-01"
  }
}
```

Figure A.4: Failed job

In Figure A.4, a bot marked a job as failed, because it could not find a certain element in an Android Application. It is up to the developer to fix this error by patching the typescript file containing the 'BookingAppScraper' class. After a fix, this job can be readded to the queue by clicking the 'Retry' button.

A.2.2 Bot Implementation

Since outlets are prone to change, it is important to know how scraping bots can be repaired.

To understand the structure of a scraping bot, consider the following class diagram:

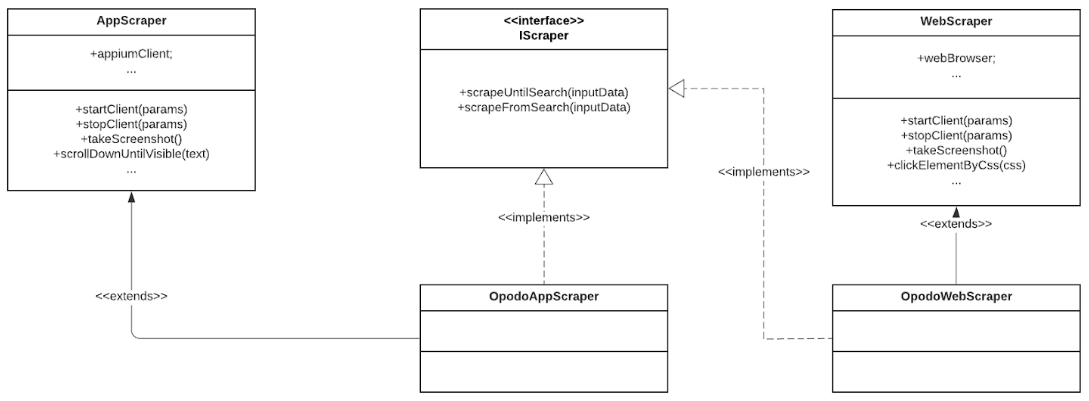


Figure A.5: Class diagram for a bot

Figure A.5 shows a class diagram that includes the mobile application scraping bot and website scraping bot for the company Opodo.

All bots implement an interface *IScraper* with two methods, *scrapeUntilSearch*() and *scrapeFromSearch*() (as discussed in 4.6; the scraping process is split into two phases).

For every type of outlet, there is a class that groups common operations. *AppScraper* groups operations common to smartphone scraping. For example, it has a property

appiumClient that is used to interact with a smartphone. It has its own *startClient*() method which instantiates *appiumClient*, and a method *stopClient*() that releases the client when scraping is finished. It has methods specific to smartphone applications. For example, the method *scrollDownUntilVisible*(*elem*) will scroll down by simulating a finger press, scrolling down by a certain amount of pixels until *elem* is visible.

WebScraper groups operations common to desktop website scraping. It has a property that represents a browser. *startClient*() will create a new instance of a browser, while *stopClient*() will close the browser. *WebScraper* contains various methods related to website scraping, for example *clickElementByCss*(*cssSelector*).

Bots can be launched in two modes: batch mode and interactive mode.

In batch mode, bots run as background processes and continuously wait for new jobs in the queues they are subscribed to.

In interactive mode, an analyst can bypass the queue system and launch bot operations directly with a built-in CLI. For example, with this command, scraping will start immediately from the Opodo mobile application:

```
ts-node cli.ts scrape OpodoAppScraper inputData.json
```

Concerning the deployment of bots: each bot runs as a Docker container. A bot can run anywhere, as long as Docker is supported and as long as there is a stable communication channel to the job queues.

For example, in this study, there were two website scraping bots deployed in Kubernetes[4]. These Docker containers include a headful browser; no extra configurations or dependencies are needed for website scraping.

For smartphone scraping bots, on-premises servers were required that allow bots to connect to a real smartphone. Our setup included one Windows laptop connected to a physical smartphone that runs continuously at home, and one Windows Server in a university data center connected to a physical smartphone.

Each bot can be configured with the following environmental variables:

Listing A.3: Environmental variables in a bot

```
DB_HOST=scraperbox.be                                  // Redis host
DB_PORT=6379                                           // Redis port
DB_PASS=*                                                   // Redis password
DEFAULT_PUPPETEER_TIMEOUT=55000         // Timeout looking for a DOM element
DEFAULT_APPIUM_TIMEOUT=25000          //  Timeout looking for and Android element
FTP_USER=fteepee                        //  FTP username to upload screenshots
FTP_PASS=*
FTP_HOST=scrapebox.be
LOG_LEVEL=debug
PULL_EMULATOR_QUEUE=false
PULL_REAL_DEVICE_QUEUE=true         // only pull jobs from smartphone queue
PULL_WEB_BROWSER_QUEUE=false
PULL_MOBILE_BROWSER_QUEUE=false
```

[4] https://kubernetes.io/

```
APPIUM_HOST=127.0.0.1                              // Appium server runs locally
DEVICE_NAME=device-5554                   // Name of Android device ('adb devices')
MAX_SYNCHRONIZATION_TIME=90                        // max sync time
TIMEOUT_BEFORE_SEARCH=90                           // timeout for phase 1
TIMEOUT_AFTER_SEARCH=180                           // timeout for phase 2
```

To be noted in listing A.3 is that a connection to Redis is configured to access the queues, and also to synchronize with other bots in a comparison.

For verification purposes, a bot is obliged to take screenshots and upload them to a central FTP server. A bot can be subscribed to one or more queues. In this case, it is only subscribed to a queue containing jobs that are meant to be processed by a bot connected to a real smartphone. The interaction with a smartphone happens via an Appium server, which is installed on the same machine. Apps will run on 'device-5554' (the names of all smartphones connected to a machine can be retrieved with the command 'adb devices'). Next, it is specified to synchronize for a maximum of 90 time units, after which scraping will continue (see section 4.6 for more information about synchronization). The first phase of scraping can take a maximum of 90 time units, and the second phase an take a maximum of 180 time units. When the first or second phase exceeds this time, the scraping will stop and a *Timeout* error will be thrown. In pseudo-code, this is a simplified version of how a bot processes jobs and synchronizes:

Listing A.4: Job processing by a Bot

```
realDeviceQueue.process(job)  //new job arrived
{
const scraper:IScraper = null
        ...
if(job.scraperClass =  'OpodoAppScraper')
                scraper = new OpodoAppScraper();
        ...

scraper.startClient();  //start mobile application
redis.jobsStartedFor(job.comparisonRunId)++;
while(redis.jobsStartedFor(job.comparisonRunId) <
job.TotalJobsInComparison and not MAX_SYNCHRONIZATION_TIME reached);
//block

scraper.scrapeUntilSearch(job.inputData); // if slower than
//TIMEOUT_BEFORE_SEARCH, throw Timeout Exception

while(redis.jobsReachedSearchFor(job.comparisonRunId) <
job.TotalJobsInComparison and not MAX_SYNCHRONIZATION_TIME reached);
//block

const offers = scraper.scrapeFromSearch(job.inputData); // if slower than
//TIMEOUT_AFTER_SEARCH throw Timeout Exception

scraper.stopClient();  //stop mobile application

finishedJobsQueue.put({...job, offers});
}
```

As can be seen in listing A.4, a bot will wait for new jobs. In this case, the job requested to scrape the mobile application of Opodo, so a new instance of OpodoAppScraper is created.

Synchronization on start is done with other bots in the same comparison. This synchronization is done for a maximum of MAX_SYNCHRONIZATION_TIME time units, after which the scraping process will continue.

The first phase of scraping takes a maximum time of TIMEOUT_BEFORE_SEARCH, after which the execution will be cut off and the job will be marked as FAILED. The same holds for the second phase.

Then, synchronization on search is done with other bots in the same comparison. This synchronization is done, again for a maximum of MAX_SYNCHRONIZATION_TIME time units, after which the scraping process will continue.

In the second phase, offers are collected and passed back to the controller via the $finishedJobsQueue$. Offers are collected for a maximum period of TIMEOUT_AFTER_SEARCH time units.

What is not visible in this code is that when one job fails, all others jobs in the same comparison will also stop. This all-or-nothing design decision means either all jobs succeed, or all jobs fail. This is a design decision; a more relaxed approach would allow scraper runs to continue when other runs in the comparison failed.

A.2.3 Controller Implementation

The controller is a centralized place where new jobs are generated and where jobs processed by bots are returned.

As with bots, the controller can be run in batch mode or interactive mode.

In batch mode, when a controller is started, a scheduler is also started that will launch new comparisons according to a CRON specification. For example, the default setting is $0 * /8 * **$ which means that the controller will launch new comparisons every 8 hours. Comparisons will be taken from the 'comparison' table and will be launched in the order of the table.

In interactive mode, a comparison can be launched as follows:

```
ts-node cli.ts launchComparison 45  //launch
//comp. 45 from database table 'comparisons'
```

Both batch mode and interactive mode will generate scraping jobs from a comparison and put these in the correct queues.

The controller offers a configuration by means of environmental variables as follows:

Listing A.5: Controller environmental variables

```
REDIS_HOST=scraperbox.be
REDIS_PORT=6379
REDIS_PASS=*
PG_HOST=scraperbox.be                //POSTGRESQL host
PG_PORT=5432
PG_USER=godfried
PG_PASS=*
PG_DATABASE=diffscraper
LOG_LEVEL=debug
CRON=0 */8 * * *          //launch comparisons every 8 hours
```

```
MAX_SYNCHRONIZATION_TIME=90    //max sync time
TIMEOUT_BEFORE_SEARCH=90             //timeout for phase 1
TIMEOUT_AFTER_SEARCH=180             //timeout for phase 2
```

The environmental variables are similar to a bot, with the addition of a relational database from which the controller reads predefined comparisons and stores offers returned by bots.

The following simplified pseudocode explains how comparisons are launched by a controller:

Listing A.6: Controller pseudo code

```
new CronJob(process.env.CRON, function{
        foreach(comparison in db.comparisons)
        {
        launchComparison();

            const sleepTime = (timeoutBeforeSearch + timeoutAfterSearch
            + synSeconds * 2) * comparison.scrapers.length * 1000;
        sleep(sleepTime);


}
});

function  launchComparison(comparison)
{
        const comparisonRunId = database.add(comparison);
        foreach(scraper in comparison.scrapers)
        {
                addToJobQueue({scraper, comparison.inputData,
                comparisonRunId, comparisonSize, ...});
        }
}
```

In listing A.2.3, first a new CRON job is created that will fire every 8 hours. Then, the code iterates through all entries in the comparison table to launch a new instance of a comparison (called a 'comparison run'). For each outlet in a comparison, a new scraper run is launched by adding a job to the correct queue that will be picked up for processing by a bot.

Lastly, after a scraper run is successfully processed by a bot, the extracted offers will be put on the 'finishedScrapes' queue, where they will be pulled by the controller and stored in a relational database according to the schema given in section A.1.

For those interested, the full source code can be found on the author's GitHub repository[5].

[5]https://github.com/godfriedmeesters/diffscraper