Recommendation-Aware Mobile Prefetching Strategies for Video Sharing Sites

Nutzung von Videoempfehlung auf Videoverteilseiten zur Unterstützung von Vorausladestrategien auf mobilen Endgeräten

Master-Thesis von Dominik Schreiber aus Frankfurt am Main

April 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Distributed Media Systems

Recommendation-Aware Mobile Prefetching Strategies for Video Sharing Sites
Nutzung von Videoempfehlung auf Videoverteilseiten zur Unterstützung von Vorausladestrategien auf mobilen Endgeräten

Vorgelegte Master-Thesis von Dominik Schreiber aus Frankfurt am Main

1. Gutachten: Prof. Dr. Wolfgang Effelsberg
2. Gutachten:

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den April 29, 2015

_____

(Dominik Schreiber)

Acknowledgement

First of all, I want to thank *God the almighty* for his love. He blessed me with the opportunity to write this thesis, and he gave me the strength to carry through.

I express my deep thanks to *Prof. Dr.-Ing. Wolfgang Effelsberg*, my supervisor. His support and encouragement made this thesis possible.

Many thanks go to *Denny Stohr* and *Stefan Wilk*, my supervisors at DMS. Their endless help, permanent open ears, and constant support were way beyond their supervisor duties. Thank you.

Furthermore, I'd like to thank the testers that participated in the evaluation of this thesis and spent a good month with my application.

Finally, I express my deep gratitude to my family and friends, especially my lovely wife *Hanna*. Without their mental support in every situation, this thesis would never have been finished.

Abstract

The masters thesis *"Recommendation-Aware Mobile Prefetching Strategies for Video Sharing Sites"* from Dominik Schreiber, submitted April 30, 2015 at Technische Universität Darmstadt, aims to evaluate *prefetching strategies* on *mobile devices* and suggests a prefetching strategy based on *personalized recommendations*.

In a digital world that more and more moves from desktop PCs to mobile devices, with their specific accounterments like limited computing power and only partially available internet connectivity, prefetching strategies need to adapt. Previously, they have been designed for full peer-to-peer networks, where a controlling peer manages and redirects requests. Now, they need to perform on a single mobile device, and cannot rely on a peer-to-peer network.

This thesis motivates the use of prefetching strategies on those devices, and suggests personalized recommendations as a source for prefetching decisions. The main part is the design and realization of an Android application that uses a man-in-the-middle approach to read the traffic between the YouTube application and the corresponding servers. It simulates both well-known prefetching strategies and newly proposed ones in parallel. An evaluation shows that a naïve caching approach performs an order of magnitude better than sophisticated prefetching strategies. Furthermore, it shows that the personalized-recommendation-based prefetching strategy performs about equal to NetTube and SocialTube, two well-known prefetching strategies.

It is therefore concluded that a naïve but fast algorithm should be the go-to approach for mobile devices. If a sophisticated approach is focused, personalized recommendations can be factored in as a valid source for prefetching decisions.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Goal . . . 1
- 1.3. Overview . . . 2
  - 1.3.1. Code listings . . . 2
  - 1.3.2. Next chapters . . . 4

2. Background **5**
- 2.1. Video sharing services . . . 5
  - 2.1.1. Types of video sharing services . . . 5
  - 2.1.2. Improving user experience . . . 5
- 2.2. Recommendation systems . . . 6
  - 2.2.1. Association rule mining . . . 7
  - 2.2.2. Collaborative filtering . . . 7
- 2.3. Mobile devices . . . 8
- 2.4. Conclusion . . . 9

3. Related Work **10**
- 3.1. NetTube: Exploring Social Networks for Peer-to-Peer Short Video Sharing . . . 10
- 3.2. SocialTube: P2P-assisted Video Sharing in Online Social Networks . . . 11
- 3.3. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model . . . 14
- 3.4. The YouTube Video Recommendation System . . . 15
- 3.5. One of a Kind: User Profiling by Social Curation . . . 16
- 3.6. Conclusion . . . 17

4. Design **18**
- 4.1. Host operating system . . . 18
  - 4.1.1. Android . . . 18
  - 4.1.2. iOS . . . 19
  - 4.1.3. Windows Phone . . . 19
- 4.2. Transparent application . . . 19
  - 4.2.1. Using a proxy . . . 20
  - 4.2.2. Using a VpnService . . . 21
  - 4.2.3. Using iptables . . . 21
- 4.3. YouTube in a WebView . . . 22
  - 4.3.1. Chrome-based WebView . . . 22
  - 4.3.2. Chromium-based WebView . . . 22
  - 4.3.3. Mozilla GeckoView . . . 23
  - 4.3.4. Patched versions of WebView . . . 23
- 4.4. Standalone application . . . 24
- 4.5. Conclusion . . . 24

5. Realization **25**
5.1. Main goals . . . 25
5.1.1. Evaluate prefetching strategies . . . 25
5.1.2. Reliable and extensible . . . 25
5.1.3. Useable in everyday life . . . 26
5.2. Architecture overview . . . 27
5.2.1. A birds-eye view . . . 27
5.2.2. Prefetching . . . 32
5.2.3. File-system management . . . 33
5.2.4. Prefetching strategy evaluation . . . 33
5.3. Realized prefetching strategies . . . 34
5.3.1. Baseline: cache watched videos . . . 34
5.3.2. NetTube: prefetch three related videos . . . 35
5.3.3. SocialTube: prefetch followers and interest-clusters . . . 36
5.3.4. NewTube: prefetch YouTube personalized recommendations . . . 38
5.4. Evaluation infrastructure . . . 39
5.4.1. Data acquisition . . . 39
5.4.2. Aggregating data on the server . . . 41
5.5. Noteworthy side products . . . 43
5.5.1. underscore.java . . . 43
5.5.2. express-persistent-resource . . . 44
5.6. Conclusion . . . 45
5.6.1. Key challenges and their solutions . . . 45
5.6.2. Outlook: further steps . . . 45

6. Evaluation **47**
6.1. Setup . . . 47
6.2. Characteristics of the dataset . . . 48
6.3. Evaluated metrics . . . 50
6.3.1. Evaluated strategies . . . 50
6.3.2. Key metrics . . . 50
6.3.3. Evaluation dimensions . . . 51
6.4. Results . . . 52
6.4.1. Cache hitrate . . . 52
6.4.2. Prefetching accuracy . . . 54
6.5. Conclusion . . . 56

7. Conclusion **57**
7.1. Retrospect . . . 57
7.2. Outlook . . . 57
7.3. Final words . . . 59

Appendix A. Tables **I**

Appendix B. Bibliography **II**

List of Figures

2.1. Network hierarchy, caches can be applied at every level; from [Wan99, p.4] 6

3.1. Network topology of NetTube: (lower-layer) *swarms* for videos $v1$ to $v5$, (upper-layer) neighborhood for $P1$; from [CL09, p.4] 11
3.2. Network topology of SocialTube: a source user S with two followers in interest-clusters; from [LSW+12, p.2888] 12
3.3. Evaluation results of SocialTube versus NetTube and PA-VoD. (a) client population → prefetching accuracy, (b) number of *prefetched* videos → prefetching accuracy, (c) number of *watched* videos → prefetching accuracy, (d) client population → percent of server contribution; from [LSW+12, p.2889] 13
3.4. Normalized click through rates per day, gathered over 21 days; from [DLL+10, p.296] . . 16
3.5. Images of the same category spread randomly before refinement (a) and clustered afterwards (b); from [GZS+14, p.572] 17

4.1. Process to set a proxy in Android: “Settings” → “Wi-Fi” → active connection → “Modify network” → “Advanced options” → “Proxy” = Manual → host and port → “SAVE” 20
4.2. a) permanent warning from Android that a root certificate is installed, b) permanent warning from Android that a VpnService is running 21

5.1. The user onboarding process. a) linking a YouTube account, b) setting the Android proxy to `localhost:9008`, c) adding the custon root certificate, d) using the application 27
5.2. Overview over the Android application structure 28
5.3. MainActivity of the realized Android application, explained 29
5.4. The internal flow inside the SandroProxy library 30
5.5. The evaluation server dashboard for the history view, captured at 2015-04-16 42

6.1. Number of captured requests a) per day, b) per hour of day 49
6.2. Number of captured requests to specific byte ranges 49
6.3. Cache hitrates of the evaluated prefetching strategies 53
6.4. Prefetching accuracy of the evaluated prefetching strategies a) based on all requests b) based on requests to 10s prefixes c) based on requests without re-requests 55

List of Listings

1.1. Lambda expressions: a) as “closure”, b) as lambda expression 2
1.2. Generics in lambda expressions 2
1.3. How 1.2 would occur in this thesis 3
1.4. Example usage of underscore.java (5.5.1: a) static method execution and b) chained calls) 3
1.5. Relative package names in the Android manifest 3

5.1. Sample video chunk request 30
5.2. HitRate performance evaluator (in class `.evaluation.strategy.HitRate`) 33
5.3. Example call of the YouTube API wrapper 34
5.4. Realization of cache watched strategy (in class `.prefetching.strategy.CacheWatched`) . . . 34
5.5. Realization of the NetTube strategy (in class `.prefetching.strategies.NetTube`) 35
5.6. Realization of SocialTube (in class `.prefetching.strategy.SocialTube`) 36
5.7. Realization of the NewTube strategy (in class `.prefetching.strategy.NewTube`) 38
5.8. the Range datatype 39
5.9. the Mimetype datatype 39
5.10. the Video datatype 40
5.11. the Strategy datatype 40
5.12. the HistoryEvent datatype 40
5.13. the Recommendation datatype 40
5.14. serializing a `ChunkHistory` to JSON (in class `.statistics.renderer.HistoryJsonRenderer`) . 40
5.15. the evaluation server REST API (in `lib/api.js`) 41
5.16. Example usage of _ 43
5.17. Realization of `_.each()` with and without Java 8 features 44
5.18. The `fields` string in *express-persistent-resource* 44

6.1. Data structure for the evaluation script 47

1 Introduction

Whoever owns a smartphone knows the pain of staring at the loading indicator waiting impatiently for it to finish. Be it a blog post about working "in the zone", a video one of the friends shared, or a gif of a cat – when we want to see it we want to see it. Now. Not in half a minute.

What if science could solve this? What if my smartphone *knows me* and knows what I am going to look for – *and downloads it in advance*? This would not only serve the impatient me, but also optimize the resources that I have: it could download at home, in my fast and free WiFi, and I could watch on the train, where I'm happy if there is enough satellite connection to make a phone call.

Now, enough of the dreaming. This thesis will show what science *can do* and how a *prefetching strategy* aware of the mobile-device related problems could help fulfilling this dream.

1.1 Motivation

Prefetching, the act of downloading content before the user actually accesses it, has been an active research field during the past years. Most work focused on peer-to-peer networks, where a controlling unit routes requests for content to the best peers. Approaches include content- as well as social-network-based prefetching.

The market of *mobile devices* grew since the introduction of the iPhone in 2007 from serving phonecall-ready cell phones to smartphones that are almost as powerfull as desktop PCs. They even outnumbered desktop devices in 2014 [LL14]. This growing market opens a new field of research: *prefetching on mobile devices*. Peer-to-peer approaches may not work, as they look at a whole network in its entirety, while the mobile device stands on its own. Therefore, new approaches have to be considered.

Finally, speaking of video sharing services, *recommendation systems* have been developed recently, that promise user-specific recommendations with high precision. Where previous prefetching strategies could only rely on content-specific recommendations, e.g. a list of "similar" videos, *personalized* recommendations may be an entry point to higher prefetching accuracy.

Combining those three topics – *prefetching* on single *mobile devices* based on *recommendation systems* – is the big picture of this thesis.

1.2 Goal

As stated above, this thesis aims to combine the topics of prefetching strategies, mobile devices and recommendation systems. This is, to the best knowledge, the first attempt to do so.

The main goal is to evaluate prefetching strategies on a single mobile device. In addition to that, a new prefetching strategy leveraging personalized recommendations shall be proposed.

The result is an Android application that simulates multiple prefetching strategies, realized for the video sharing service YouTube, simultaneously. Furthermore, an evaluation server with a RESTful API is created, that collects reported pseudonymized data from application instances to allow centralized analysis.

1.3 Overview

The next chapters offer in-depth discussion of all parts involved in the research done for this thesis. To remove theoretical barriers up front, it is started with necessary general explanations – especially regarding code listings (1.3.1) – as well as an outlook to the following chapters.

1.3.1 Code listings

Throughout this thesis, there will be several code listings. To make sure they are understandable, this is an overview of the conventions and abbreviations used:

Lambda expressions

Java 8 introduced the concept of *Lambda Expressions* to the Java language [GJS+15, p.601-612]. As "anonymous functions" they can drastically improve readability. Generally speaking, an anonymous class that has only one method and no other members can be refactored to a lambda expression without thinking. Listing 1.1 gives an example.

Listing 1.1: Lambda expressions: a) as "closure", b) as lambda expression

```
// a) a "closure"
new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
}
// b) the same, as lambda expression
() -> { System.out.println("Hello World!"); }
// or even
() -> System.out.println("Hello World!")
```

That is why e.g. the Android IDE, AndroidStudio, offers so-called "closure folding" that displays anonymous classes of that kind as lambda expressions, even though Java 7, which is used by Android, does not support the language feature.

Lambda Expressions make heavy use of *generics* and almost never declare an explicit type – they simply rely on the Java compiler to determine the right type. Please note that still everything is statically typed, the language has not lost any strength. It just makes use of the already existing feature of generics and type inference to aid the reader/developer. Listing 1.2 shows how generics and type inference are used to dramatically trim the amount of code needed to perform an operation.

Listing 1.2: Generics in lambda expressions

```
Map<String, String> languages = new HashMap<>(); // <> is the "diamond" operator
                                                 // from Java 7, it infers its
                                                 // type from the static declaration
languages.put("Haskell", "great");
languages.put("Lisp", "awesome");
languages.put("Java", "improving");
languages.put("C++", "...");

// printing languages + comments using lambda expressions
// (and java.util.Map.forEach, which came with Java 8 as well)
languages.forEach((lang, comment) -> System.out.println(lang + " is " + comment));

// the same, with underscore.java without lambda expressions
_.each(languages.entrySet(), new BiConsumer<String,String>() {
```

```
    public void accept(String lang, String comment) {
        System.out.println(lang + " is " + comment);
    }
});
```

So, for readability, code listings in this thesis make use of lambda expressions even though in the code there is actually an anonymous class with only one method. There is confidence in the reader that he will be able to infer the actual type from the context – a task that is normally done by the java compiler. The example from listing 1.2 would therefore occur as shown in listing 1.3.

Listing 1.3: How 1.2 would occur in this thesis

```
_.each(languages.entrySet(), (lang, comment) ->
    System.out.println(lang + " is " + comment));
```

underscore.java

The helper library _ already appeared in previous listings (1.2, 1.3). It was realized during this thesis to simplify dealing with collections of data and brings functional programming approaches to (pre version 8) Java. Refer to section 5.5.1 for more in-depth information.

As almost every class/method realized makes use of _, it is suggested to get a little familiar with the concept – at least to not get afraid once an _ occurs in a listing. An example is given in 1.4.

Listing 1.4: Example usage of underscore.java (5.5.1: a) static method execution and b) chained calls)

```
// a) static methods of _
List<Integer> fibonaccis = Arrays.asList(new Integer[] {1, 1, 2, 3, 5, 8});
List<Integer> squares = _.map(fibonaccis, (i) -> i * i);
// => squares = [1, 1, 4, 9, 25, 64]

// b) chained calls
List<Integer> oneToHundred = _.range(1, 101);
int sumOfSquaresOfEvenNumbers = new _<>(oneToHundred)
                                        .filter((i) -> i % 2 == 0)
                                        .map((i) -> i * i)
                                        .reduce((a, b) -> a + b, 0);
// => sumOfSquaresOfEvenNumbers = 171700
```

Relative package names

Another way to increase readability of Java code is to define a common *package prefix* and then name packages only relative to this package prefix. This is used for example in Android manifest files, where a **package** attribute is set for the whole file, and subsequent Java classes/packages are referenced relatively. Listing 1.5 gives an example.

Listing 1.5: Relative package names in the Android manifest

```
<manifest package="com.dominikschreiber.ytproxy" ...> ...
    <application ...>
        <!-- a shorthand form of com.dominikschreiber.ytproxy.ui.MainActivity -->
        <activity android:name=".ui.MainActivity" ...>...</activity>
    </application>
</manifest>
```

Throughout this thesis, whenever a package name begins with a dot (e.g. `.fs.Cache`), resolve it relative to `com.dominikschreiber.ytproxy` (e.g. to `com.dominikschreiber.ytproxy.fs.Cache`).

Usage of the Android Code Style Guidelines

While it is a purely esthetical consideration, one might wonder why variable names in the code listings of Java classes have an "unnecessary" prefix (`m` and `s`). This is due to the *Android Code Style Guidelines for Contributors* [All15], which evolved to the generic go-to styleguide reference in the Android environment.

They state, among other aspects, that *instance members* should be `mPrefixed`, *static members* should be `sPrefixed`, and *constants* should be `ALL_CAPS`.

1.3.2 Next chapters

The rest of this thesis is organized as follows:

Chapter 2 introduces necessary vocabulary as well as *background* information regarding video sharing services (2.1), recommendation systems (2.2) and mobile devices (2.3).

Related work is discussed in chapter 3. The focus is on prefetching strategies (3.1, 3.2) and personalized recommendation systems (3.3, 3.4, 3.5).

Then, the *design* process that led to the actual Android application is presented in chapter 4. This involves selecting a host operating system (4.1), considering a transparent application (4.2) as well as a patched website in a WebView (4.3) or a completely self-contained application (4.4).

A transparent application for an Android host operating system was created, and the *realization* process is described in chapter 5. The main goals are laid out (5.1), the application architecture is shown (5.2), realized prefetching strategies are explained (5.3), the evaluation infrastructure (5.4) and arose side projects (5.5) are described, and the key issues are commented in a conclusion (5.6).

To back the decisions up with hard facts, an evaluation of four prefetching strategies is performed (6). Cache hitrate and prefetching accuracy (6.3.2) are evaluated for all data, 10s-prefixes and filtered out re-requests (6.3.3). It is found that a naïve caching approach performs an order of magnitude better than sophisticated strategies in any metric (6.4). A conclusion is that focus could be laid on improving naïve strategies rather than sophisticated ones (6.5).

The thesis is ended with an overall conclusion (7). A summary of the whole work is given (7.1), and an outlook to further work is made (7.2). Final words end the thesis (7.3).

2 Background

To get into the topic of "Recommendation-Aware Mobile Prefetching Strategies for Video Sharing Sites", the three parts of this thesis are examined in this chapter, rolling it up from the behind. First, in section 2.1, video sharing services are described and categorized. Then, in section 2.2, recommendation systems and their methodologies are explained. Finally, in section 2.3, the special properties of mobile devices are illustrated.

2.1 Video sharing services

Video consumption is one of the key applications in today's internet usage. The platform YouTube alone serves multiple hundred million hours of video content per day[1], and over 25 percent of western students spend more than 15 minutes per day on YouTube [SVBW11, p.4]. All video-on-demand services combined are responsible for over 60 percent of world wide internet traffic [Cis14, p.10]. This makes video-on-demand a huge topic of research, to better understand current usage patterns and to improve the future experience.

2.1.1 Types of video sharing services

The Cisco Visual Networking Index divides internet video services into *short-form*, *video-calling*, *long-form*, *live internet TV*, *internet PVR* (personal video recording), *ambient video* and *mobile video* [Cis14, p.12], from which short-form and long-form can be considered video-on-demand services – the subject of this thesis. Mobile video is listed as an own category, but while the other categories are based on the *content* of video traffic, this category is solely based on the *infrastructure used* – namely 2G, 3G and 4G mobile network. A short-form video watched over 3G would fall into both categories.

However, the split into *short-term* and *long-term* videos is crucial: short-form videos are usually user-generated, uploaded and spread over social networks and video platforms, what gives them a strong social component and high capacity for personalized approaches. Long-form videos, on the other hand, are more "traditional": full-featured movies served to set-top-boxes of cable providers or tv series produced for single video-on-demand services like NetFlix[2], Amazon Prime Instant Video[3] or hulu[4]. Here, the focus lies on the content itself, and the social component, if any, is less important. The topic of this thesis are user-generated content videos, usually *short-form* – and possible improvements for their user experience.

2.1.2 Improving user experience

While video-on-demand is such a large topic, the user experience in video sharing services is often rather questionable. As the video has to be downloaded before it can be shown, the user has to wait a significant amount of time, before enjoying the content. To solve this issue, several key approaches from both classical network theory as well as from research on video consumption have been developed.

1 youtube.com/yt/press/statistics.html, checked on 2015-04-28
2 netflix.com, the Netflix video sharing service, checked on 2015-04-28
3 amazon.com/piv, the Amazon Prime Instant Video video sharing services, checked on 2015-04-28
4 hulu.com, the Hulu video sharing services, checked on 2015-04-28

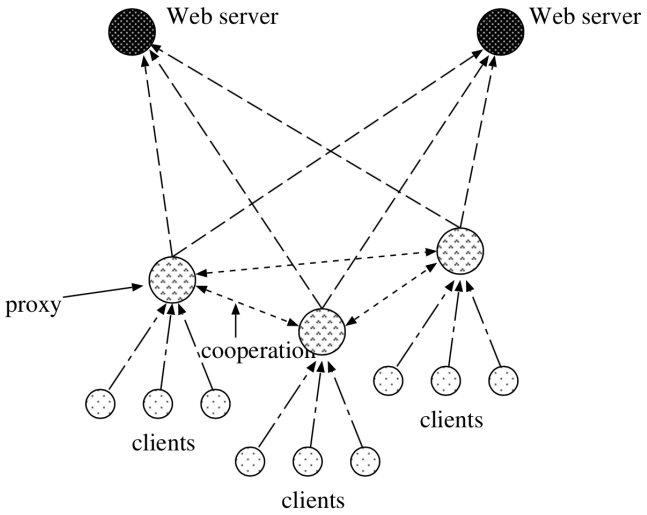

Figure 2.1.: Network hierarchy, caches can be applied at every level; from [Wan99, p.4]

A first approach is to *buffer* a video and start the playback without the video being fully downloaded [DDM+95, p.3,5]. Speaking of a video as a set of chunks – each chunk a set of single images (so called "frames"), or audio parts, delivered in a single TCP/UDP package, one needs only the first few chunks with a length greater than the estimated time needed to download the rest of chunks to start video playback. This provides a significant speed boost compared to a download-first-watch-later approach.

To further improve the video startup time, *caching* of videos can be applied. This has been widely researched for internet traffic in general, and hierarchical caches at different network levels have been proposed [Wan99, p.6-10]. Figure 2.1 displays a common network hierarchy, where caches can be installed at every stage: the *web server* itself can cache requests in its working memory instead of performing filesystem operations, *proxy servers* at local gateways can cache requests for their subnet leading to less requests to the server, and the *client* itself can cache videos for later re-play. This means, the video is kept in memory instead of being removed immediately, so that future requests do not have to be propagated all the way through. With this mechanism, watching videos can again be significantly sped up.

Finally, instead of caching videos that have already been watched, one can *prefetch* videos that have not been watched yet – but are likely to be watched in the future. Details on multiple strategies on deciding which videos to prefetch are given in 3, and a new prefetching strategy is proposed in 5.3.4.

2.2 Recommendation systems

In *data mining* and *machine learning*, recommendation systems are an active topic of research. They generally aim to predict user preferences in order to suggest items with the highest estimated "rating". Their appearance is widely known, it ranges from "users who bought x_1 also bought x_2" recommendations at online shopping platforms to listening recommendations from iTunes Genius[5].

On the theoretical level, recommendation systems can be categorized as either *content- or preference-based* [JSS12, p.1]. Content-based recommendation systems make recommendations based on the user's browsing/purchasing history – and most often use *association rule mining* [AS94] (2.2.1) for this. Preference-based recommendation systems, on the other hand, use *collaborative filtering* [ERK11] (2.2.2) to infer recommendations from item or user similarities.

[5] apple.com/itunes/music, the iTunes features – including Genius – explained, checked on 2015-04-28

2.2.1 Association rule mining

Association rule mining infers rules from a set of transaction data (where "transactions" can be purchases, views, likes or anything else) that hold true for the majority of the dataset. Such a rule could be "98% of users that bought a coffee mug also bought coffee beans". [AIS93, p.208] gives a great formalization:

> *"Let $\mathscr{I} = I_1, I_2, \ldots, I_m$ be a set of binary attributes, called items. Let T be a database of transactions. Each transaction t is represented as a binary vector, with $t[k] = 1$ if t bought the item I_k, and $t[k] = 0$ otherwise. There is one tuple in the database for each transaction. Let X be a set of some items in $\mathscr{I}$. We say that a transaction t* satisfies *X if for all items I_k in X, $t[k] = 1$.*
>
> *By an* association rule*, we mean an implication of the form $X \Longrightarrow I_j$, where X is a set of some items in $\mathscr{I}$, and I_j is a single item in $\mathscr{I}$ that is not present in X. The rule $X \Longrightarrow I_j$ is satisfied in the set of transactions T with the confidence factor $0 \leq c \leq 1$ iff at least c% of transactions in T that satisfy X also satisfy I_j."*

Applied to a huge dataset – e.g. the total four billion video views on YouTube *per day* (in 2012) [Spo12] – this can lead to a large amount of association rules (but can also be a very time/resource-consuming task).

An association rule mining algorithm would do multiple breadth-first (the *Apriori* algorithm, [AS94]) or depth-first (the *Eklat* algorithm, [Zak00]) searches through the transaction dataset to find association rules, e.g. exploiting the *downward-closure property* [LHM99, p.338] of rules to scale performant.

2.2.2 Collaborative filtering

Unlike association rule mining that focuses on the users transaction history, collaborative filtering focuses on *similarities*. They focus on user or item *neighborhoods*, translate both users and items to a *latent factor space* or combine both approaches to a *hybrid* model (which, as often, usually performs best).

Neighborhood models

A *user* neighborhood uses transactions of *similar users* to recommend new items. For example: Andy and Ben are both software developers in their late twenties living in San Francisco, CA, owning a cat and drinking insane amounts of coffee. Andy watches "Silicon Valley"[6]. A user neighborhood-based recommendation system would suggest Ben to also watch this series.

An *item* neighborhood, on the other hand, uses transactions of a single user with *similar items* to recommend new items. An example would be: Carl watched the "The Matrix"[7] and "The Matrix Reloaded"[8]. Now, the movie "The Matrix Revolutions"[9] shares a lot of features – like a large subset of the cast, the directors, even parts of the title – with those watched movies; so an item neighborhood-based recommendation system would suggest watching this movie as well.

The important task in designing any neighborhood model is defining a *similarity measure*. It will be used to find the most similar users/items which are the base for recommendations. For item neighborhoods, the *Pearson Correlation Coefficient* [FRS01] ($\rho_{x,y} = \frac{cov(x,y)}{\sigma_x \sigma_y}$) came up promising: it is used to measure the probability of an item-tuple (i, j) to get the same ratings from multiple users. So, starting with a seed item i, a user would be suggested that item j that has the highest Pearson Correlation Coefficient across the dataset.

[6] imdb.com/title/tt2575988, "Silicon Valley" in the IMDb, checked on 2015-04-28
[7] imdb.com/title/tt0133093, "The Matrix" in the IMDb, checked on 2015-04-28
[8] imdb.com/title/tt0234215, "The Matrix Reloaded" in the IMDb, checked on 2015-04-28
[9] imdb.com/title/tt0242653, "The Matrix Revolutions" in the IMDb, checked on 2015-04-28

Neighborhood models perform well for "obvious" recommendations. Leaving out the vast majority of data and concentrating only on the top k nearest neighbors (either user- or item-wise), they struggle with finding more hidden correlations.

Latent factor models

Where neighborhood models try to create recommendations from user↔user or item↔item relationships, latent factor models *compare users and items directly*. Those factors are usually automatically inferred from user interactions. An example could be the genre of a music title, where the the item (a song) could be transformed to something like `is_rock()` and the user could be transformed to something like `listens_to_rock()`. Again, they are usually automatically inferred and may not at all open up to the human reader (but work well on the dataset). As one single feature would be pretty one-dimensional, both users and items are represented as feature vectors.

The techniques to obtain latent factors reach from *Latent Dirichlet Allocation* [BNJ03] over *Singular Value Decomposition* [Pat07] to *Neuronal Networks* [SMH07].

Latent factor models have hard times finding associations in a small dataset – where neighborhood models shine –, but perform well in understanding the overall structure of a large dataset. Therefore they are able to provide recommendations on a completely other level.

As latent factor models struggle where neighborhood models are good at, and vice versa, a combined *hybrid* approach looks like the obvious next step. In fact, such a combination performs pretty well, and SVD++, an algorithm that does this, is described in detail in section 3.3.

2.3 Mobile devices

After introducing *video sharing services* (2.1) and *recommendation systems* (2.2), the final purpose of this chapter is to define and explain *mobile devices*.

With more iterations of Moore's law [Sch97] and therefore smaller chips with greater computing power, cell phones evolved to smartphones with almost desktop-PC-like computing power over the past several years. While likeminded mobile devices existed since the early 1990s (e.g. IBM Simon [Kob09]), their huge uplift started with Apple Inc. unveiling the iPhone in 2007[10]. Multitouch input as well as an application ecosystem offering possibilities far beyond phone calls or short messages were key features to this success.

Measured since 2007, the global number of users of mobile devices exceeded the number of desktop device users in 2014 [LL14], it even exceeded the human population [Cis15, p.3].

Mobile devices are subcategorized as "smartphones" and "tablets" based roughly on their screen size – smartphones around 5 inches [Bar14], tablets around 10 inches. Depending on their price segment, they are equipped to hardly perform anything else than phone calls and messaging (entry segment, 100-200$) or are full-featured desktop PCs in small cases (top-level segment, around 1000$). As they almost all depend on *reduced instruction set computer* (RISC) architecture chips instead of the desktop-leading *complex instruction set computer* (CISC) architecture [Bha97], they are way less energy consuming and sometimes plain faster, but applications have to be developed and compiled solely for this processor architecture.

A common bottleneck to all mobile devices is their connectivity: at places without public/known WiFi, their connectivity comes from a satellite plus base-station union using mobile connectivity standards 2G (GSM) to 4G (LTE). Depending on the users current location and cell phone plan, a modern 4G connection – which offers speeds even above wire-based DSL – might not be available and the slower 3G and 2G standards need to be used. This is a very noticeable slowdown, as 2G GPRS has a maximum connection speed of 40 kBit/s (compare that to 300 mBit/s of 4G) [HRM04].

[10] macworld.com/article/1054769, "Apple unveils iPhone" by Mathew Honan, PCWorld, checked on 2015-04-28

2.4 Conclusion

It has been shown that

- *video sharing services* are responsible for the majority of internet traffic and that prefetching can improve the there-perceived user experience;
- *recommendation systems* are an active topic of research with both content- and preference-based approaches that result in association-rule-mining or collaborative filtering;
- *mobile devices* had a huge rise since 2007 – now even exceeding desktop PCs – and that internet connectivity is a main bottleneck.

These findings combined, following the discussion of related work (3) should be feasible.

3 Related Work

To get a better understanding of the current state of the art, research related to prefetching and recommendation systems is reviewed in this chapter. The presented literature is the most relevant for the topic of this thesis.

In the recent decade there has been plenty of work regarding prefetching strategies, especially targeting peer-to-peer-assisted algorithms. Those algorithms create network overlays of clients directly connected together and sharing information, in order to unburden the server. The research presented in this chapter focuses on *NetTube* (3.1), where every client prefetches the first 10 seconds of 3 videos of the "related videos" list on YouTube, and *SocialTube* (3.2), where clients are split into "interest-clusters" and prefetch published videos in a *push* way (rather than pulling, what is used nearly everywhere else).

Recommendation systems, which become more and more personalized – from a global "top k" list to recommendations for a single user – as more and more data on single users is available, are the other topic of great interest in this chapter. The prefetching strategy proposed in this thesis (5.3.4) leverages personalized recommendations, and thus sets itself apart from previous strategies. In this chapter, the recommendation systems of Netflix (3.3) and YouTube (3.4) are reviewed. They are both collaborative filtering algorithms that use an item neighborhood to recommend new videos, but Netflix also takes a latent factor model into account. Further, the use of social curation services (in practice: Pinterest) for better recommendations is discussed by Geng *et al.* (3.5). They generate item ontologies from "pins" (images) in "boards" (manually assigned tags) and observe heavy gains in recommendation accuracy using this approach.

Finally, section 3.6 draws a conclusion on the presented work.

3.1 NetTube: Exploring Social Networks for Peer-to-Peer Short Video Sharing

Cheng and Liu proposed a strategy for peer-to-peer sharing of short videos – e.g. from YouTube – in [CL09]. They observed that prior research results, which were merely obtained from traditional video sharing services, are not applicable for short video sharing services like YouTube, because the videos served are heavily different, especially in video length. More than that, they found that prior peer-to-peer approaches relied on those traditional-service videos (of 1-2 hours of length) and did therefore create new peer-to-peer overlays for every video, which does simply not work for short videos as the playback time is shorter than the overlay creation time.

Therefore, they proposed a two-layered peer-to-peer overlay. In the lower layer, there are *swarms* of *peers* for videos: every client is called a "peer" and caches all videos he watches, and all peers that have watched a single video form the "swarm" for that video – they serve the video to clients that want to see it. In the upper layer, neighborhoods of clients/peers are modeled: a neighborhood for a single client is the set of swarms that contain either the client itself or clients that are in a swarm the client is in as well. Figure 3.1 illustrates this. The lower layer serves as sources for the current video being watched, while the upper layer helps to find swarms for the next video as fast as possible – this is crucial as the videos are generally short and many videos may be played in total before the right swarm is found (in which case the server would have served it alone). To further speed up the swarm-finding process, a search index based on a *Bloom filter* [Blo70], which had a false-positive rate of 0.0014 on the test set, is used.

Most interestingly, NetTube uses a prefetching algorithm to lower the transition time between two videos even further. Having observed that smooth video playback is possible when 59% of a video are downloaded, they conclude that prefetching the first ten seconds of a 200 second video (the measured average across all YouTube videos) is enough to ensure smooth playback – assuming that downloads are

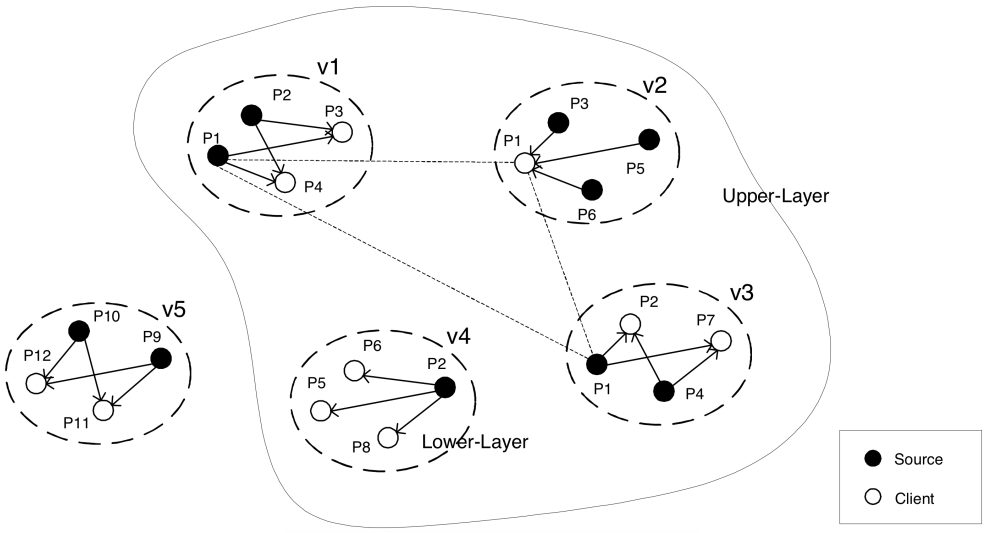

Figure 3.1.: Network topology of NetTube: (lower-layer) *swarms* for videos $v1$ to $v5$, (upper-layer) neighborhood for $P1$; from [CL09, p.4]

twice as fast as playback, the next 20 seconds of the video can be downloaded while the first 10 are played, from then on it grows exponentional. Using the un-evaluated assumption that users select the next video solely based on the top *n* related videos list, Cheng and Liu propose the following prefetching strategy:

> *When watching a video, fetch prefixes (of 10 seconds) for 3 videos in the related videos list that are not in the cache yet, and cache them.*

This leads to a prefetching accuracy of 90% after playing 5 videos in their simulation. In an evaluation with a self-created dataset (with the same probability distribution as the YouTube dataset), the prefetching had an accuracy of up to 55% with 10.000 clients when using the two-layered peer-to-peer overlay (nearly 0 with only the lower layer). A prototype experiment with 200 clients showed that 95% had an average startup delay below 4 seconds using NetTube, while only 70% had that using Peer-Assisted VoD [HLR07] – a reference implementation.

While this research is almost canonical and facilitated a lot of further research (e.g. [LSW+12,KZGZ12, THI+10]), the results might have aged. They are based on the assumption that users select the next video based on the related videos list. But as YouTube evolved, the ways to discover new videos evolved as well (see 3.4). Therefore the assumption is no longer true. Also, the video length is no longer restricted to 10 minutes, so the average of 200 seconds per video might be wrong – or at least have a high variance if still near right. Therefore the prefetched prefix length of 10 seconds has to be questioned.

As this thesis focuses on a single mobile device, the peer-to-peer prefetching scheme can not be used directly. However, as a baseline, the NetTube prefetching scheme for a single peer is of high interest and is evaluated in 6.4.

3.2 SocialTube: P2P-assisted Video Sharing in Online Social Networks

Inspired by [CL09] (see 3.1), Li *et al.* developed a peer-to-peer-assisted prefetching strategy that takes social relations between users into account [LSW+12]. They studied video sharing in online social networks (namely: Facebook[1]) and found that viewers can be divided into *followers* that watch nearly all videos a user publishes and *non-followers* who watch some but not all videos. The conclusion is that

[1] facebook.com, checked on 2015-04-28

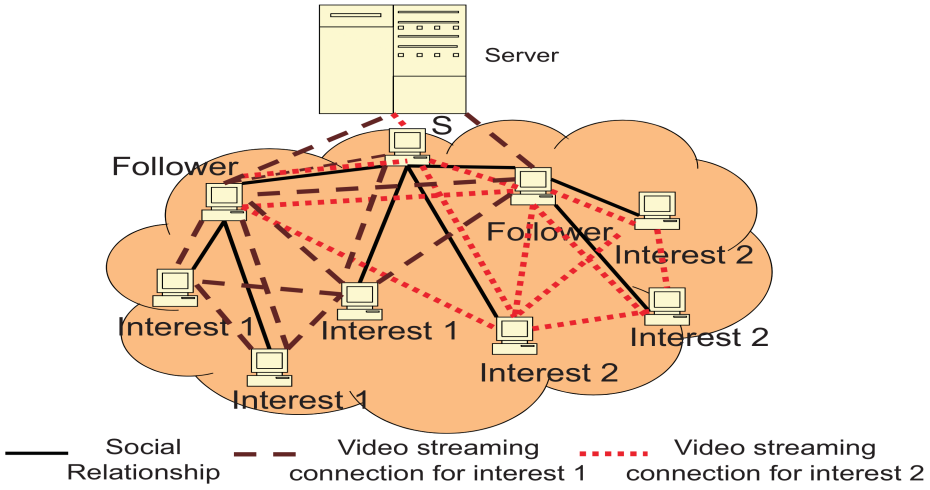

Figure 3.2.: Network topology of SocialTube: a source user S with two followers in interest-clusters; from [LSW+12, p.2888]

followers watch those videos because they were published by the user, whereas non-followers watch them because they are interested in the content. Another observed concept is that users share videos for different *interests* (e.g. "music", "news", "cats") and that non-followers watch videos of matching interests (e.g. all "cats", no "music").

Based on their observations, they proposed a *per-user* peer-to-peer overlay. Users are in general only connected to users within two hops in their friends graph (e.g. to the friend of a friend, but not to the friend of a friend of a friend). They are organized in *interest clusters*, so two users are in the same interest cluster if they happen to be interested in the same category of videos (e.g. "cats"). *Followers* of a user are placed in all interest clusters the user is in, even if they are not interested in the topic (they are interested in the user). A user with his followers is therefore called *swarm*. In figure 3.2 interest clusters and swarms are displayed. Follower/non-follower roles are maintained and assigned by a central server.

The proposed prefetching algorithm relies heavily on follower/non-follower roles. Interestingly, prefetching is done in a "push" way (rather than "pulling", what is used almost anywhere else): when a user publishes a new video, he actively sends the first chunk of the video to all users he shares an interest cluster with (note that this includes his followers – they are in every interest cluster). It is derived that this prefetching strategy has a high accuracy as followers and interest-cluster-peers are assumed to want to watch the video.

Evaluation versus implementations of NetTube [CL09] and PA-VoD [HLR07] shows the advantage of this prefetching algorithm: while PA-VoD had a prefetching accuracy of almost 0 regardless of the number of clients, and NetTube dropped its accuracy from 75% at 1000 users to 40% at 5000 users, SocialTube had a constant prefecthing accuracy of almost 95% at all simulated numbers of clients when prefetching prefixes of 5 videos. Even when prefetching only one prefix at a time, SocialTube retained 70% accuracy. Figure 3.3 shows all evaluation results.

SocialTube has therefore proven to be a well-performing peer-to-peer-assisted prefetching strategy. However, the focus in this thesis is on a single user with his device, not a whole network of devices. Therefore the SocialTube strategy needs to be adapted in order to be used here. But friends – and therefore a friend graph – exist in YouTube as well. They are called "channels" and "abonnements", but the concept stays the same. The prefetching application may observe that the user is a *follower* of some other user and prefetch every video published by this user, or it may observe that the user is an *interest-cluster-peer* of other users and prefetch the matching videos. This behavior is evaluated in 6.4.

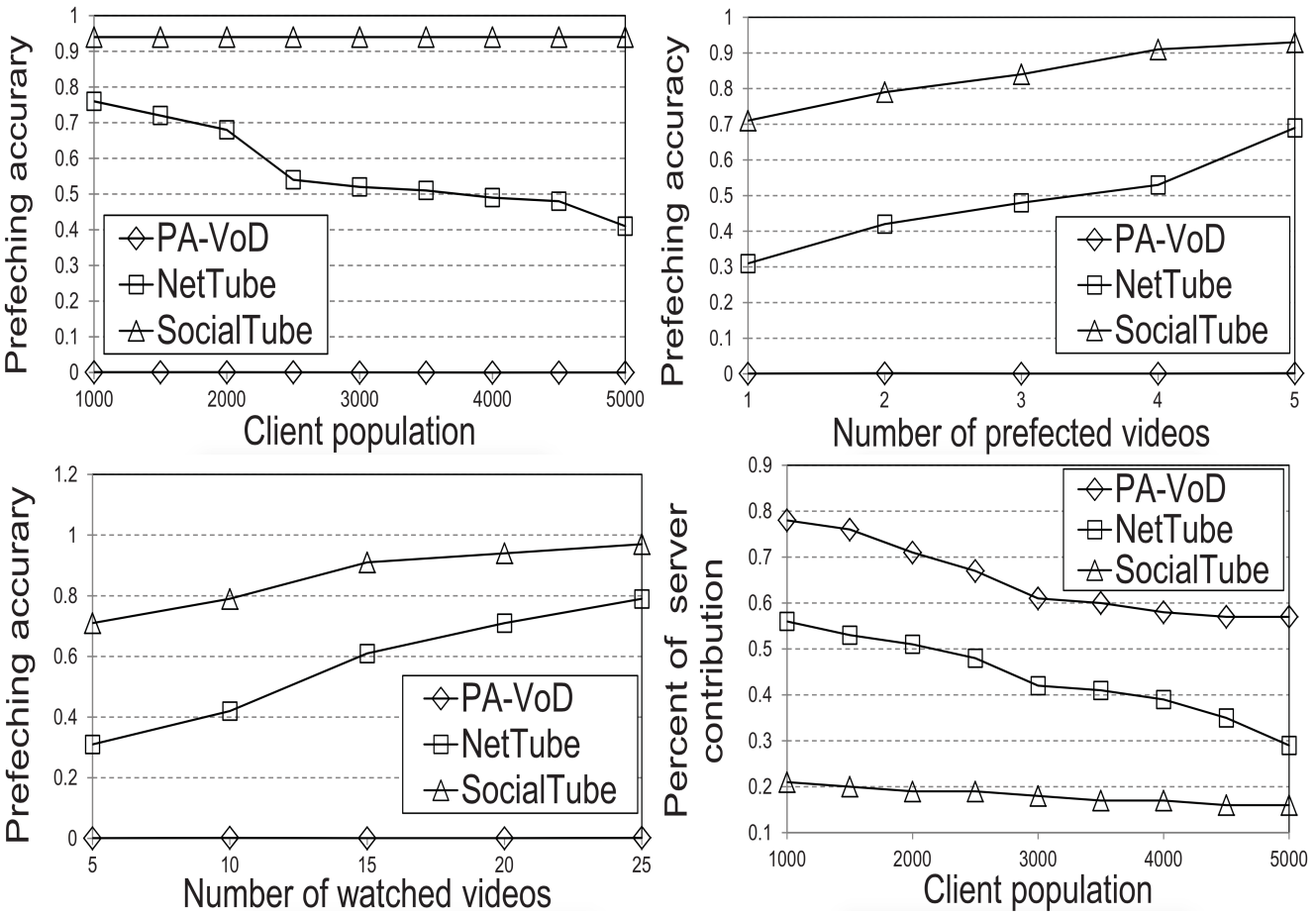

Figure 3.3.: Evaluation results of SocialTube versus NetTube and PA-VoD. (a) client population → prefetching accuracy, (b) number of *prefetched* videos → prefetching accuracy, (c) number of *watched* videos → prefetching accuracy, (d) client population → percent of server contribution; from [LSW+12, p.2889]

3.3 Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model

In 2006, the video-on-demand service *Netflix*[2] announced a one million dollar trophy to the research group that would beat their video recommendation system, *CineMatch*, by at least 10% (on the given training-/test-set). Yehuda Koren, a member of the team that won this competition, provided insight into their algorithm in [Kor08].

Their key advantage is that they combined a *neighborhood model* with a *latent factor model* for a *collaborative filtering* based recommendation system (2.2.2). Those two models were mutually exclusive before. For the neighborhood model, they stuck to an *item* neighborhood (in contrast to a *user* neighborhood) as this provides simple explanations for new recommendations (e.g. "you watched 'The Lord of the Rings: The Fellowship of the Ring'[3], you may also like 'The Lord of the Rings: The[4] Two Towers' and 'The Lord of the Rings: The[5] Return of the King'."). As sources of feedback both explicit ratings (from 1 to 5 stars) and implicit ratings – the fact that a user did rate a video in any way – were used and combined.

One crucial trick was to refine the task of *recommending videos to a user* to *estimating ratings a user would give, then showing the ones with the highest estimated rating*. The final calculation to find an estimated rating $\hat{r}_{ui}$ from user u for video/item i is shown in equation (3.1). It is sometimes referred to as *SVD++* (e.g. in [DPN+14, p.67], [ML13, p.315]).

$$\begin{aligned}\hat{r}_{ui} = \mu + b_u + b_i + q_i^\intercal \left(p_u + \frac{\sum_{j \in N(u)} y_i}{\sqrt{|N(u)|}} \right) \\ + \frac{\sum_{j \in R^k(i;u)} \left(r_{uj} - b_{uj} \right) w_{ij}}{\sqrt{\left| R^k(i;u) \right|}} \\ + \frac{\sum_{j \in R^k(i;u)} c_{ij}}{\sqrt{\left| N^k(i;u) \right|}}\end{aligned} \tag{3.1}$$

where parameters are defined as follows:

- μ the global average of all ratings
- w_{ij} empirically found global weights
- b_u the bias of user u to the average
- b_i the bias of item i to the average
- b_{ui} a baseline for the estimated rating
- c_{ij} an offset to the baseline, depends on implicit preference of item j over i
- $q_i, y_i \in \mathbb{R}^f$ item factor vectors
- $p_u \in \mathbb{R}^f$ the user factor vector
- $N(u)$ items rated implicitly (here: rated explicitly in any way) by user u
- $R(u)$ items rated explicitly (here: rated 1-5 stars) by user u
- $S^k(i;u)$ the k nearest neighbors to i for u
- $N^k(i;u) = N(u) \cap S^k(i;u)$
- $R^k(i;u) = R(u) \cap S^k(i;u)$

It has to be noted, that although this algorithm won the Netflix Prize competition, it was never actually used in production at Netflix. Amatriain and Basilico from Netflix explain this in [AB12a, AB12b]:

[2] netflix.com, checked on 2015-04-28
[3] imdb.com/title/tt0120737, "The Lord of the Rings: The Fellowship of the Ring" in the IMDb, checked on 2015-04-28
[4] imdb.com/title/tt0167261, "The Lord of the Rings: The Two Towers" in the IMDb, checked on 2015-04-28
[5] imdb.com/title/tt0167260, "The Lord of the Rings: Return of the King" in the IMDb, checked on 2015-04-28

We evaluated some of the new methods offline but the additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment.

Nonetheless, the work behind it has a serious value, as it is the baseline for further research – and it won the authors a million dollars, what could be called worth itself. For YouTube – the video sharing service of interest in this thesis – some adoptions would have to be made in order to use SVD++ as a recommendation system. There is for example no 1-5 star rating system[6], but one could e.g. calculate the probability of a video being "upvoted" as $r_i = \frac{\text{upvotes}_i}{\text{views}_i}$ and see this as the explicit rating.

The question whether SVD++ will perform equally well on YouTube data is still present. Netflix content is professionally crafted and long (usually over 45 minutes), whereas YouTube content is user-generated and short (usually shorter than 10 minutes due to a historical constraint on video length). Therefore, at least the weights and feature vectors would have to be learned totally new. Thus it is left to further research to implement SVD++ for YouTube.

3.4 The YouTube Video Recommendation System

As YouTube is the practical video sharing service of interest in this thesis, it is of great value that Davidson *et al.*, developers of the YouTube recommendation system, published internals of the algorithm in [DLL+10]. The goals of this recommendation system are to provide *relevant videos*, but also to present *new content* that is not necessary similar to the content watched by the user.

Similar to SVD++ (3.3), the YouTube recommendation system follows the collaborative filtering approach (2.2.2), but uses only a neighborhood model and no latent factor model. The *item neighborhood* is constructed based on "co-visitation counts" c_{ij}– the number of times videos i and j have been watched together in a given time period (24 hours in that case). Relatedness is then obtained as $r_{ij} = \frac{c_{ij}}{f(i,j)}$ (where f is a normalization, in the simplest case $f(i,j) = c_i \cdot c_j$ with c_k the number of times k has been watched in this period).

To create a set of recommendations C_{final}, the algorithm starts at a *seed set* S of videos the user did watch, and iteratively adds the N most related videos R_i^N for every video i in the set. After some (K) iterations, the seed set videos are removed from the then final recommendations set. Equation (3.2) shows the algorithm. These recommendations are afterwards ranked based on video quality, user specifity and diversification, and a small subset of the final recommendations is shown.

$$\begin{aligned} C_0 &= S \\ C_n &= \bigcup_{i \in C_{n-1}} R_i^N \\ C_{\text{final}} &= \left(\bigcup_{i=0}^{K} C_i \right) \setminus S \end{aligned} \tag{3.2}$$

The fact that videos are added to the recommendations set based on relatedness to a seed video watched by the user makes it easy to explain why a recommendation is given: "you get this recommendation, because this video is often watched together with 'Epic Funny Cats', which you have watched".

In A/B-testing evaluation [Huf08] over three weeks, personalized recommendations based on this algorithm were contrasted to "traditional" recommendation systems, namely "top favorited", "top rated" and "most viewed" lists. The focus lay on "click through rate" (CTR) – the probability of a recommended

[6] There was indeed a 1-5 star rating system, but users almost always used 5 star ratings [Raj09], so they changed it to a simple *voting* system [FK10]

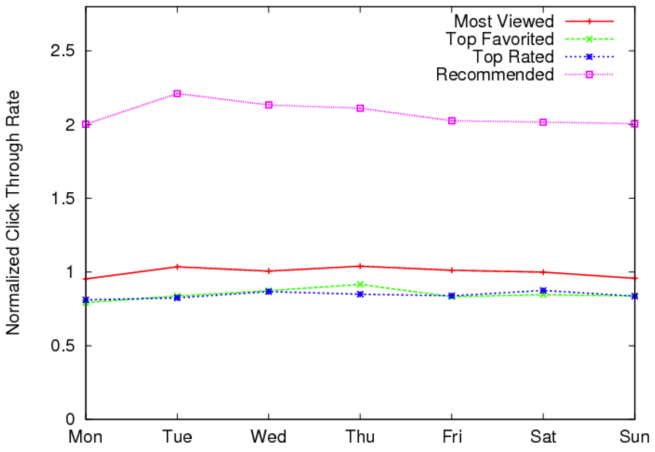

Figure 3.4.: Normalized click through rates per day, gathered over 21 days; from [DLL+10, p.296]

video actually being watched. In this evaluation the personalized recommendations performed in average 207% better than the "most viewed" list, both other lists nearly performed the same. Figure 3.4 illustrates the results.

The insights gained through this research are of great interest for this thesis. First and foremost it is to see that the personalized recommendations are by far the best-performing method of recommendations at YouTube. Therefore it seems reasonable to use them as a base for a prefetching algorithm. However, the personalized recommendations have an explicit goal to "highlight the broad spectrum of content that is available on the site" [DLL+10, p.293]. Sadly it was not evaluated how these *new content* recommendations perform in contrast to *similar content* recommendations – this could have been done using A/B-testing as well, but it is not reported in the paper. It could therefore easily be the case that *similar content* recommendations perform very well but *new content* recommendations do not. If the to-be-designed prefetching algorithm trusted every recommendation the same, it might be harming itself. This leads to the need of telling *new content* and *similar content* recommendations apart to evaluate them separately.

3.5 One of a Kind: User Profiling by Social Curation

Geng *et al.* recently investigated ways to leverage manually created tags for media content to provide more meaningful and therefore more accurate recommendations [GZS+14]. They focused on the image-sharing site Pinterest[7], where users "pin" images to named "boards" (e.g. a board named "Cookies" or "Hair") and other users share ("repin") and comment images and boards.

Using board names as *tags*, they were able to automatically construct user preference ontologies by iteratively refining a base ontology of Wikipedia[8] entries with the boards/tags users used for different images. This gave them the opportunity to correctly cluster images of the same category. Figure 3.5 shows how images of the same category were spread randomly across the space before refinement, and how they are clustered afterwards.

The generated ontology allowed to fine-tune the tags associated with images. I.e. an image tagged "outfit" was then tagged as "outfit", "maxi dresses", "sweater" and "tank tops". The researchers state an average F_1-score of 0.48 on an 770k-image test set for this tag re-association. The recommendation system based on the learned ontologies outperformed both content-based as well as collaborative filtering methods by 13.3% to 54% on the same test set.

7 pinterest.com, checked on 2015-04-28

8 en.wikipedia.org, Wikipedia, The Free Encyclopedia, checked on 2015-04-28

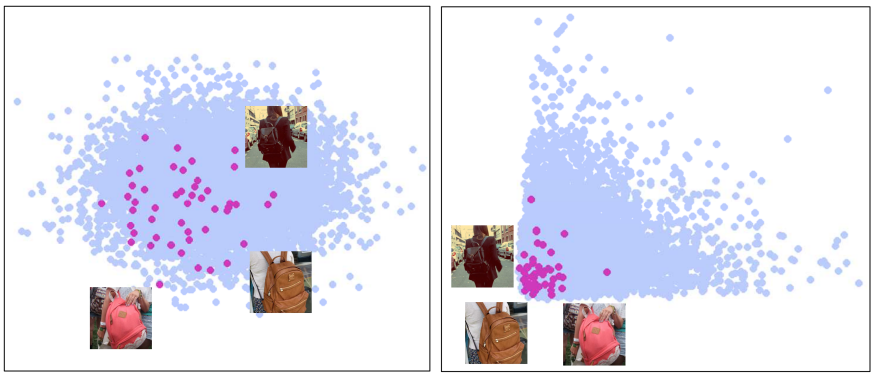

Figure 3.5.: Images of the same category spread randomly before refinement (a) and clustered afterwards (b); from [GZS+14, p.572]

While the results are great for manually curated images, it is questionable whether they could be used for video sharing services. In this field, it is much more uncommon to manually classify content. YouTube offers "Playlists" as a way for user-curated content, but while "boards" are a key concept of Pinterest, "Playlists" are a more niche product on YouTube– here, "channels" and the search are way more important. YouTube "Channels", on the other hand, are not necessarily topic-focused, and should be seen as "registered users" rather than "boards". This all makes it really hard to learn a speaking ontology of videos, and a good ontology is the base of all work in [GZS+14].

3.6 Conclusion

In the previous sections, related work on the topics of prefetching strategies and recommendation systems was presented.

It was shown that *NetTube* (3.1) achieved 55% prefetching accuracy by prefetching the first 10 seconds of 3 videos in the related videos list at YouTube on every client, and that *SocialTube* (3.2) managed to maintain a prefetching accuracy of almost 95% by pushing chunks of published videos to users in "interest-clusters" of the source user. However, these strategies relied on a peer-to-peer overlay, which is not applicable for a single mobile device. Nonetheless the proposed strategies are realised for evaluation purposes in (6.4).

Current recommendation systems like the ones of Netflix (3.3) and YouTube (3.4) use collaborative filtering approaches to provide recommendations personalized for the single user, rather than a global approach (like "top k", "most viewed" or "recently added" lists). Koren *et al.* won the Netflix Prize by combining an item neighborhood model with a latent factor model in *SVD++*, while YouTube uses an iterative item neighborhood model based on co-visitation counts to get click-through rates twice as high as from global recommendation lists. Sadly, SVD++ was tuned for a traditional video sharing service with professionally crafted videos of 1-2 hours of length – this is hardly applicable for the user-generated 10-minute videos at YouTube. Moreover, the YouTube recommendation system integrates "new content" (to explore the platform) and "similar content" and no evaluation was made how the click-through rates differ between the two. Therefore, the algorithms may be taken cum grano salis. The ontology-based recommendation system of Geng *et al.* (3.5) is, as good as its results are, not useful for YouTube, as the necessary requirements are not met.

Nonetheless it is to note that both prefetching strategies as well as recommendation systems are an active field of research that benefit heavily from technical progress, as bigger memory allows to prefetch more, and more computational power allows to create better recommendations.

4 Design

The following pages describe the process that led to the design of an application that allows to evaluate prefetching strategies on a single mobile device. The result is the design for an application for mobile devices running Android. The application opens a proxy server right on the device that behaves as a man in the middle for every request performed in any application running on the device. Once a request for a prefetched video comes in, the proxy is able to serve it directly and will not forward the request to the original server. This makes the prefetching *transparent* to the user, who can just use the applications he was using before. Details on the realization of this application are given in chapter 5.

The process to this design involved deciding on the host operating system (4.1)) as well as pondering alternatives of a transparent application (4.2)) versus the YouTube web page in a browser-like WebView (4.3)) versus a new application with user interface that displays videos directly (4.4)). A conclusion (4.5) sums it up.

4.1 Host operating system

The smartphone market divides into different operating systems. There is a heavy trend towards the google-supervised Android (4.1.1), the main competitor, with a market share of 84.7 percent (≈ 190 million devices) in mid 2014, followed by iOS (4.1.2) from Apple with 11.7 percent (≈ 30 million devices) and Windows Phone (4.1.3) with 2.5 percent (≈ 6 million devices) [IDC14]. To target a large audience, only those three competitors are considered.

4.1.1 Android

Based on a Linux kernel, Android evolved from a niche project to the worlds most popular smartphone operating system – developed at Google, open-source in most parts and highly customizable by (experienced) users. This makes Android popular for research as well. As root access can be obtained easily, researchers can do pretty much whatever they want with their devices.

Android applications are developed using Java [GJS+13] – altough the compiled bytecode is not run in a traditional Java Runtime Environment, but in an android-specific runtime called “Dalvik” (soon to be replaced by “Art”, as of the time writing). They are published via the default “Google Play Store” or other stores (like Amazon App-Shop[1]). More interestingly, the compiled application can be installed directly onto the device from a file (i.e. obtained via usb connection or downloaded from the web). This roughly means: *everyone can release everything and everything can be installed everywhere*.

There is a strong security concept behind application execution on Android [EOM+09]. Applications run in their own *sandboxes* and are not able to access resources outside of their sandbox. They have to ask for *permissions* for certain activites at installation time – i.e. an application that needs internet access explicitly asks for `android.permission.INTERNET`, which the user has to accept in order to install the application.

The high market share, the user-friendly security system, and the common programming language led to the decision to realize the prefetching application for Android devices. However, the considered alternatives should be laid out in the following sections for the sake of completeness.

[1] amazon.com/gp/mas/get/android, checked on 2015-04-28

4.1.2 iOS

Running on the first iPhone, iOS brought the advent of smartphones to the global market. However, as it is closed-source developed by Apple and used only on their iPhone lineup, it targets only mid to high-end audience – there is simply no device below 200$, where Android gains 60% of its market share from.

To develop applications for iOS, one has to use an Apple Mac[2], as applications can only be developed with the proprietary Xcode[3] IDE. They used to be developed using the language "Objective C" [Koc11], but recently shifted to "Swift"[4], which is more functional-programming oriented. An application that should be published in the iOS App Store – the only way to get applications from – is reviewed by teams at Apple and only applications that pass this review are allowed to be published. In contrast to the permission system used in Android (see 4.1.1), the user has to rely on the manual reviewing process to avoid maleficent applications.

The review process is an actual blocker for realizing the prefetching application on iOS. This application will, by intent, change the way other applications work (4.2) or pretend to be something it isn't (4.3). Such behavior is explicitly banned[5]. This leaves a new application with user interface that displays videos from YouTube (4.4) as the only option – which has it's own drawbacks.

4.1.3 Windows Phone

Although Microsoft has a long history of creating smartphones based on their *Windows Mobile* platform, modern-day smartphones running *Windows Phone*[6] have a global market share of only 2.7 percent. This may be due to their late start to the market – 3 years after the first iPhone – as well as missing updates and relatively few existing applications.

Applications for Windows Phone are developed using Microsoft .NET technologies, like C#, C++ or JavaScript. The improvement for developers is, that the application will run on the desktop version of the operating system as well – so there is a seemless integration of both, and applications can in fact target an insane amount of devices. They are published in the Windows Store, where they undergo a manual review, just like iOS applications do in the Apple app store.

While it is not explicitly stated in the Windows Store submission guidelines[7], it may be assumed that an application that alters the behavior of other applications (4.2) would be rejected during the manual review process. It is to note that at the moment of writing, there is an official YouTube[8] application, it is however not maintained by YouTube itself but by Microsoft directly – this means there could be a totally different user experience.

4.2 Transparent application

From the research point of view, a transparent application would be perfect. Imagine users that use the official application of their favorite video sharing service just as if nothing ever happened. The research results would be generalizable, the user experience great.

Several options have been considered to achieve this type of application. First and foremost, a proxy based on the open-source SandroProxy[9] application is the way to go (see 4.2.1). It has some important

2 actually, one could develop locally without Xcode and let a cloud service compile the sources – but that's somewhat out of the scope of this paper
3 developer.apple.com/xcode, checked on 2015-04-28
4 developer.apple.com/swift, checked on 2015-04-28
5 developer.apple.com/app-store/review/guidelines article 8.5, checked on 2015-04-28
6 windowsphone.com, checked on 2015-04-28
7 msdn.microsoft.com/hh694062.aspx the submission guidelines, checked on 2015-04-28
8 windowsphone.com/de-de/store/app/youtube/dcbb1ac6-a89a-df11-a490-00237de2db9e the official YouTube application, checked on 2015-04-28
9 sandrop.googlecode.com, checked on 2015-04-28

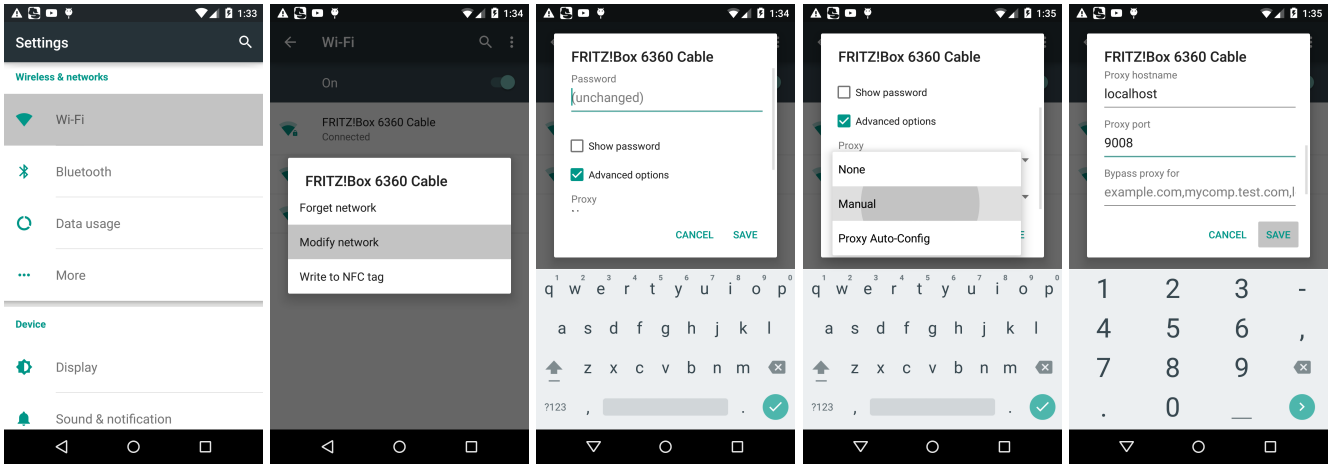

Figure 4.1.: Process to set a proxy in Android: "Settings" → "Wi-Fi" → active connection → "Modify network" → "Advanced options" → "Proxy" = Manual → host and port → "SAVE"

features already implemented in it, and is the option that has actually been selected. Alternatives were using a *VpnService* (4.2.2), which would be "the Android way", and using *iptables* (4.2.3), which would require root access to the device, excluding everyone who uses a stock Android smartphone.

4.2.1 Using a proxy

A proxy is a software, that sits in the middle of every network connection between client applications and servers. To the client, it impersonates the server, to the server, it impersonates the client. This is especially useful in business or restricted environments, where a proxy can hide the client from the server (by impersonating someone else), can filter the traffic (to prevent information leakage), or – perfect for the prefetching application – answer requests directly instead of forwarding them to the server.

To set a proxy in Android, one has to perform *eight* "simple" steps (figure 4.1 illustrates them):

1. open the *Settings* application
2. open the *Wi-Fi* settings
3. *long-tap* on the active Wi-Fi connection
4. tap on *Modify network*
5. check *Advanced options*
6. set *Proxy* to *Manual*
7. fill in *host* and *port*
8. *SAVE* the changes

This is somewhat complicated, but it is a way to proxy *all the traffic* from the device that *does not require root access*. The prefetching application can open a proxy server on the device itself – i.e. on `localhost:9008`, and will never even leak private information to the wrong hands, as the proxy is right on the device.

If every connection was not encrypted, the application would be done already. Seeing every request, performing it themselves, returning the result. Unfortunately, this is not the case. YouTube, the host application this research is interested in, requests videos via HTTPS [Res00, §2.1], so literally everything a naïve proxy sees is `CONNECT googlevideo.com:443`, not the actual request. This makes responding with locally prefetched videos impossible – the proxy never sees *which* video actually is requested, just *that* any video is requested.

So, there is a need to decrypt HTTPS connections in the proxy. If the proxy was a "man-in-the-middle" before, it has now to become a "man-in-the-middle *attacker*" (although a friendly one). The considered attack is as follows:

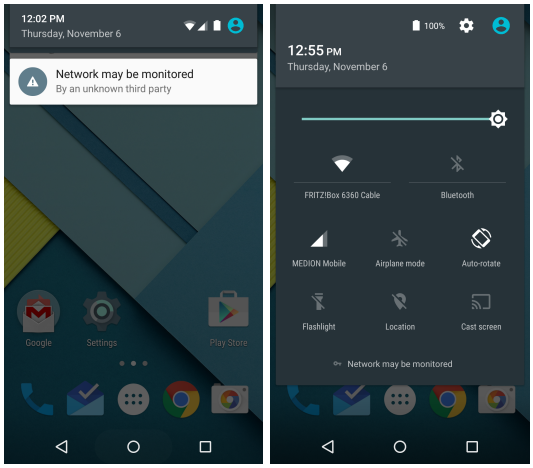

Figure 4.2.: a) permanent warning from Android that a root certificate is installed, b) permanent warning from Android that a VpnService is running

1. Register a *root certificate* on the device. Only downside: the user will permanently be warned by Android that a custom root certificate is installed. Figure 4.2 a) shows how.
2. *Generate certificates* for HTTPS requests dynamically, pretend to be the requested server.
3. Create new HTTPS connections with the actual servers, pretend to be the requesting client. Or answer the request directly, if possible.
4. Forward the answer from the server to the client, using the generated certificate. The client will trust this connection, as the registered root certificate trusts the generated certificate.

Luckily, there is an ancient Java penetration testing tool, *WebScarab*[10], that is capable of performing a man-in-the-middle attack of that kind. It has been ported to Android for the open-source proxy application *SandroProxy*. So, the prefetching application uses the SandroProxy library, that removes the user interface and replaces it with a simple user interface (5.2.1) to start/stop the prefetching and register video-sharing site accounts.

4.2.2 Using a VpnService

To create virtual private network applications, Android (since Android 4.0 “Ice Cream Sandwich”, mid 2011) offers the official way of registering a `VpnService`, where all traffic is routed through. This is, like a proxy, a way that *does not require root access*. But, as the proxy using a root certificate, a VpnService means permanent notifications to the user that the network is monitored. Figure 4.2 b) illustrates this.

Proxy (4.2.1) and VpnService share the same problem here: as YouTube uses HTTPS, one has to decrypt the connection, which requires a man-in-the-middle attack. While such an attack is already realized using a proxy (4.2.1), it would have to be realized from scratch for a VpnService. This is the main reason a proxy is the technique of choice to realize the prefetching application.

4.2.3 Using iptables

Android is basically a Linux running on a smartphone. On Linux it is quite common to alter network traffic using *iptables*[11], a very fundamental firewall/routing application. With a plethora of configuration

[10] github.com/OWASP/OWASP-WebScarab, checked on 2015-04-28
[11] netfilter.org/projects/iptables, checked on 2015-04-28

options, one can block/forward requests and redirect traffic on a packet level based on source/destination IP addresses, protocols, networking interfaces and many more. This gives network administrators the ultimate freedom of configuring their network.

So, a possibility to realize the prefetching application, would be to use the iptables program built-in to Android. Sadly, this requires *root access*, which would exclude every user from the prefetching application that has not gained root access to his device. To get root access, a user has to break the Android security concept, using well-known exploits to the underlying Linux operating system to raise from user to root access.

SandroProxy, on which the prefetching application is actually based, is capable of using iptables rules if it detects that it is running on a "rooted" device. But then again, forcing a user to exploit his smartphone operating system is one of the worst user experiences possible, which is why it was decided to not realize prefetching relying on iptables.

4.3 YouTube in a WebView

If a transparent application (4.2) would not have been possible, the next-best alternative is to display the web applications of video sharing services – i.e. `m.youtube.com` – in a so-called "WebView". A WebView is essentially a browser without the browser user interface and with additional capabilities, like access to native methods. Reviewed alternatives are the official Chrome-based WebView (4.3.1), a Chromium-based port for older devices (4.3.2), the Mozilla GeckoView (4.3.3) as well as patched versions of the Chromium-based WebView (4.3.4).

4.3.1 Chrome-based WebView

Since Android 4.4 ("KitKat", late 2013) the official WebView uses exactly the same bowels as the current Chrome browser shipped with Android. So, every web application that runs in the browser, can run in a WebView as well. This could be used to display the mobile-optimized web applications of various video-sharing sites as a "native" application.

The Android WebView even offers methods to intercept *every request* made from the web application inside the WebView and answer it directly instead of letting it go through. This concept seems promising: look at every request, intercept requests for (parts of) videos that have been prefetched locally, answer them directly, done. And it is. It's the little details that hinder here.

m.youtube.com requests videos from subdomains of googlevideo.com. This violates the "Same-Origin Policy" [Bar11, §3], so the server has to respond with corresponding "cross-origin resource sharing" (CORS) headers [vK14] to allow the browser to actually fetch the response. googlevideo.com does this. But until Android 5.0 ("Lollipop", late 2014) the WebView did not allow to set custom headers to responses for intercepted requests. So, there was no way to answer so that the browser was allowed to accept. This changed with Android 5.0, but setting that as the required operating system does again exclude nearly every Android user, as new versions tend to be adopted slowly by the user base (in 11/2014 only 27% of users did use the current version, 4.4, while 17% did use a version that was 3 years or older) [MV14].

4.3.2 Chromium-based WebView

As aforementioned, Android users tend to use old versions of their operating system instead of updating – due to carrier constraints or simply the "never touch a running system" policy. This is the reason that, at the time of writing, even the Chrome-based WebView (introduced in Android 4.4, late 2013) is only available to 27.7 percent of users [MV14].

There has been effort to bring a full-fledged WebView to older devices by compiling from the Chromium sources directly[12]. The goal was to provide the *exact same* API as the official, modern Chrome WebView (Java-class `android.webkit.WebView`) in a third-party library (Java-Class `com.mogoweb.chrome.WebView`).

Unluckily, as the goal was to totally resemble the official API, even the obvious flaws in API design were resembled. In particular, this is the missing way of setting response headers to intercepted resources. So, the reason the Chromium-based WebView can not be used is exactly the same as the reason the pre Android 5.0 Chrome-based WebView can not be used (see 4.3.1).

4.3.3 Mozilla GeckoView

Not only Google, but also Mozilla try to bring their browser as a "headless" version to Android. Mozilla calls this project "GeckoView"[13], as the used rendering engine is named "Gecko", and the "Web" in *Web*View hints to "Webkit", the engine used in Chrome/Chromium.

Sadly, this project is in it's earliest infancy. They themselves describe it as

> *"Note that GeckoView is NOT ready to be used in a production environment. It is currently possible to load webpages, but that's about it."*

The main problems occuring are missing support for HTTPS, no way to bind Java code (application-side) to JavaScript code (web-side), as well as missing codecs for the videos displayed on YouTube– which led the YouTube web-application to no longer display the videos directly but add links that will open in the native Android YouTube application.

4.3.4 Patched versions of WebView

Patched XMLHttpRequest

For the Chrome-/Chromium-based WebViews (4.3.1/4.3.2) the main problem was that the "Same-Origin Policy" did not allow resources from different origins that do not provide necessary CORS headers. But both WebViews allow to register "Java⟷JavaScript-Bridges" that execute application-side code when called from the website inside the WebView, and execute web-side code when called from the application – this is intended to provide native functionality to the web-application. It could be exploited by replacing the whole implementation of `XMLHttpRequest`[14] with an implementation that does not check for CORS headers. Unfortunately, the XMLHttpRequest has not only methods but fields as well, and fields can not be shared with a Java⟷JavaScript-Bridge. This makes a 100 percent re-implementation impossible. And even if this was possible, the re-implementation of this core browser feature would easily go beyond the scope of this thesis.

Patched response

The Chromium-based WebView project (4.3.2) compiles from the Chromium sources directly. So, it would be possible to change the return type of `shouldInterceptRequest`[15] – the method that decides whether to load a local resource or forward the request to the server – from a custom response without headers to e.g. a default `HttpResponse`. In the process of patching this one method, one has to touch nearly every part of Chromium. Therefore, this approach has been rejected – it simply exceeds every reasonable effort.

[12] github.com/mogoweb/chromium_webview, checked on 2015-04-28

[13] wiki.mozilla.org/Mobile/GeckoView, checked on 2015-04-28

[14] developer.mozilla.org/docs/DOM/XMLHttpRequest, checked on 2015-04-28

[15] android.webkit.WebViewClient#shouldInterceptRequest(WebView,String), added in API level 11 (Android Honeycomb, early 2011)

Patched `<video>`**-tag**

If YouTube did never request videos from different origins, the whole problem of Same-Origin Policy and CORS would have vanished. To achieve this, it was tried to replace the html5 `<video>`-Tag that actually displays the video in the YouTube website with a different one – one that points to a local file instead of a different origin. But then, the security policy of the WebViews does not allow non-local websites (like `m.youtube.com`) to load resources using the `file://` protocol. The workaround[16] used in the Cordova project[17] involves copying all resources to the internal storage of the device and serving from there – which is not practicable in the case of the prefetching application, as it would narrow the size of the prefetching cache even more.

4.4 Standalone application

In the case that both a transparent proxy application (4.2) and YouTube in a WebView (4.3) failed, a complete standalone application with own user interface would be the only possible way. The application could fetch videos from YouTube and display them in a user interface similar to the official one – but could also enhance this user interface.

This has the great advantage of offering additional tools that support the prefetching directly. I.e. a "prefetch this" option, where a user manually selects videos, channels or playlists to be prefetched, could give great insights in the users decision-making. He may never watch the YouTube-recommended videos (although the recommendations are personalized), but watch every video his favorite channel publishes.

Another advantage would be that this application could merge videos from different video sharing services into a single user interface.

But the great disadvantage – and the reason this path was not followed far – is that research results from such an application can hardly be generalized to the original video sharing service. Videos may be presented in a whole different way in the original service, which would lead to different viewing habits and wrong assumptions, if the results were generalized.

4.5 Conclusion

In the design phase, multiple approaches to realize an evaluation application for prefetching strategies on mobile devices have been considered:

- a *transparent application* that intercepts internet traffic based on a proxy, a VpnService or iptables-redirects, which would be the most ideal approach from the research point of view;
- displaying a mobile web site in a *WebView* – either the stock Android WebView, the Chromium-based equivalent, Mozilla's GeckoView, or a patched version of one of the above;
- playing videos in a *standalone application* that resembles the user experience of applications of popular video sharing services, but offers additional research-focused elements like e.g. a "prefetch this" button that's usage could be directly evaluated.

The proxy-based transparent application approach was chosen as it promises the best results. As the host operating system, it was decided to use Android, as it is the most popular mobile operating system in research.

[16] github.com/odbol/Html5Video, checked on 2015-04-28
[17] cordova.apache.org, checked on 2015-04-28

5 Realization

The previous chapter (4) explained the thought process behind creating a transparent proxy as an Android application, that man-in-the-middle-attacks the communication between the YouTube Android application and the YouTube servers.

This chapter now will guide through the actual realization of this application, as well as showing the key challenges and their solutions.

First, the main goals when realizing the application are explained (5.1) together with a short recap of the motivation behind them. Then, an overview over the global application architecture is given (5.2). The focus is then laid on the realization of multiple prefetching strategies (5.3). It is shown how the infrastructure for evaluating the application was set up (5.4) and what noteworthy side products were spinned off (5.5). The chapter ends with a conclusion on the challenges and solutions during this realization (5.6).

5.1 Main goals

In the realization of the Android application, multiple goals were weighed against each other. First and foremost, the ability to evaluate prefetching strategies (5.1.1) was key. Once this was achieved, the application had to work realiable and be extensible to allow future work (5.1.2). Then, being useable in everyday life (5.1.3) was the nice-to-have goal that was always looked after. Finally, there was a fixed time/workload frame that implicitly capped meeting all set goals at once.

5.1.1 Evaluate prefetching strategies

Previous research on prefetching strategies (3.1,3.2) was based on log dumps collected over a period of time. Li *et al.* for example crawled Facebook to collect a 1 million entry dataset and simulated a "natural" environment based on network bandwith and usage statistics to evaluate SocialTube [LSW+12, p.2889]. This fit well to the purpose of their work, proposing a globally operating prefetching algorithm. However, for the goal of this thesis, to study prefetching on a single mobile device, such a global approach was not applicable.

It was instead opted for collecting data right on the device. This allows data crunching on the device as well. It set the cornerstone for an application that could even *adapt its prefetching strategy* dynamically to the users behavior. At the same time it preserves user privacy, as there is no need for the application to communicate with a common server. This way, all data stays on the device and no one except the user is able to know which videos were viewed, cached or prefetched.

One should note that, for the sake of evaluation of the proposed strategy, pseudonymized calls to an evaluation server were integrated (5.4). But this does not interfere with the decisions in prefetching/caching made by the application on the device. It can easily be removed without impact – except rendering a *global* evaluation of prefetching strategies impossible.

5.1.2 Reliable and extensible

Being reliable and extensible is a key feature for success and a must-have for almost any application. Not only does any user expect reliability, but every developer working on the application in the future has a

hard time if the application was never designed with reliability and extensibility in mind. Naturally, the Android application realized in this thesis focuses on those features as well.

To stick to the goal of reliability, most application code was developed in a *test-driven* [Bec03] manner. This approach in one sentence would be "write tests that fail, make them pass, rinse and repeat". Doing so in the Android environment is not particularily easy: tests can not be run on the development machine but must be run on an actual device (or an emulator) as the binaries needed to execute Android code are only available on the device. Therefore, running a single test can easily take 5-10 minutes instead of milliseconds. To deal with this hindrance, the application was split in a generic Java library and an Android-specific part. The library was then tested using plain JUnit[1] tests executed during the gradle[2] build whereas testing the Android part was based on the provided `ActivityInstrumentationTestCase2`[3]. One has to note that no tests were written for third party modules – e.g. the SandroProxy submodule (5.2.1) – so overall, only a low test coverage was achieved.

An extensible structure was obtained by adhering to object-oriented application development best practices, as described e.g. by Grady Booch [Boo06] and Robert C. Martin [Mar08]. The application code was held modular by, amongst others, following the *Single Responsibility Principle* [Mar08, p.138-140], enforcing *Inversion of Control* [Fow04, p.3-4] and using *asynchronous events* [GHJV94,] instead of synchronous/blocking calls for communication. Again, the large amount of third party components was not necessarily realized in that way.

Finally, some concepts were borrowed from *functional programming* [Hug89], especially methods to deal with collections (lists, sets, arrays and the like) of data. Those concepts, like recursion abstractions of `map`, `filter` and `fold`, lead to a way more readable and more understandable code as they abstract from CPU instructions and allow a more problem-focused style of writing. The main challenge was to use those concepts on Android: while they have already been introduced in the Java 8 language specification [GJS$^+$15, p.310,321-329], Android devices run Java 7 – or even Java 6 on devices before Android 4.4 (Kitkat, late 2013). Because of that, it was decided to create a backwards-compatible helper library *underscore.java*, inspired by *underscore.js*[4], that allows this functional programming style on Java 6 (explained in more detail in 5.5.1).

5.1.3 Useable in everyday life

Scientific contribution was the main goal for the application development. But especially in the early design decisions a real improvement for everyday life was targeted. As motivated (1.1), the prefetching should lower the users mobile bandwidth load but be as transparent as it could – without introducing additional complexity to the end user.

To achieve this goal, the application was realized with a user-centric user interface. A user onboarding process, that launches when the application is opened for the very first time, guides the user through the setup and usage of the application. Figure 5.1 shows the steps involved. From then on, the user can simply hit the "start proxy" button, leave the application, and forget about its existence.

To achieve the main goals, a lot of third party libraries were used. They did not always work as expected (5.6.1). Furthermore, the time constraints were confining. After all, the main goal – scientific evaluation – should not be endangered. Therefore, actual prefetching was omitted. Instead, multiple prefetching strategies were simulated. This allowed a great complexity reduction. Still, it did not remove parts relevant for the scientific results. This solution is not by any means perfect, but with the given time/workload constraints it was the best possible option. Realizing actual prefetching is a time-

[1] junit.org, Java unit test framework, checked on 2015-04-28
[2] gradle.org, build automation tool, checked on 2015-04-28
[3] developer.android.com/reference/android/test/ActivityInstrumentationTestCase2.html, functional testing base class, checked on 2015-04-28
[4] underscorejs.org, JavaScript functional programming library, checked on 2015-04-28

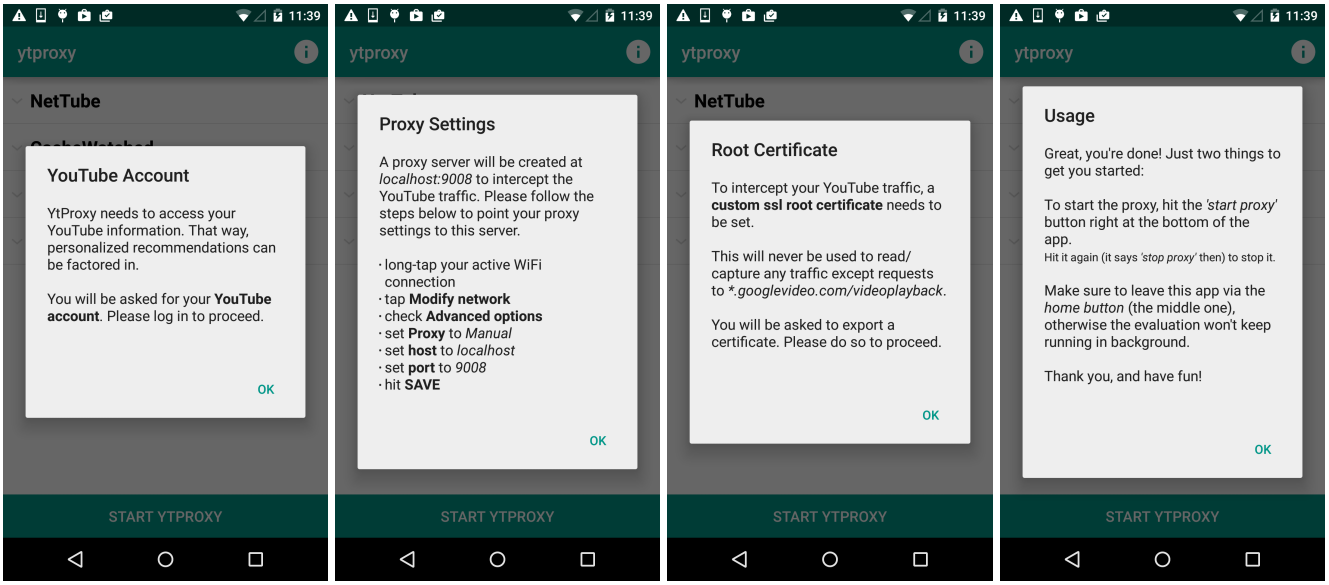

Figure 5.1.: The user onboarding process. a) linking a YouTube account, b) setting the Android proxy to `localhost:9008`, c) adding the custon root certificate, d) using the application

consuming but well-understood task (e.g. many concepts may be borrowed from *youtube-dl*[5]) and is left open for future work (7.2).

5.2 Architecture overview

The application realized during this thesis is a large program. 11597 lines of code plus 8901 lines of comments (30293/18769, when including the SandroProxy library) are spread across 84 classes (292, including SandroProxy).

As this is too large to be described in every aspect, focus is laid on the most relevant topics. Therefore, the following section gives only a birds-eye view of the project as a whole (5.2.1), and focuses then on the scientifically more crucial parts of simulating prefetching (5.2.2), simulating file-system management (5.2.3) and giving an on-device evaluation of the simulated prefetching strategies in real-time (5.2.4).

5.2.1 A birds-eye view

Figure 5.2 gives an overview about the modules involved in the realized application. The basic working mechanism is "Route all outgoing network traffic through a local proxy, man-in-the-middle all communication, react on video requests, and display results".

Once the application is started, the user gets to see the *MainActivity*. When he starts the proxy, a new thread running a custom version of the open source application *SandroProxy* starts. This routes every http request through a *HttpClient* that watches for requests for videos from the Google servers. The HttpClient finally uses the *event-based architecture* to notify, among others, the simulated prefetching strategies about new videos, which then update their simulated caches accordingly.

MainActivity

The `.ui.MainActivity` is to the Android application, what the **public static void** `main(String[] args)` is to a vanilla Java program: it is (configured to be) the main entry point to start the application. Figure 5.3 shows how the activity looks to the user.

[5] rg3.github.io/youtube-dl, Python-based command line application to download YouTube videos, checked on 2015-04-28

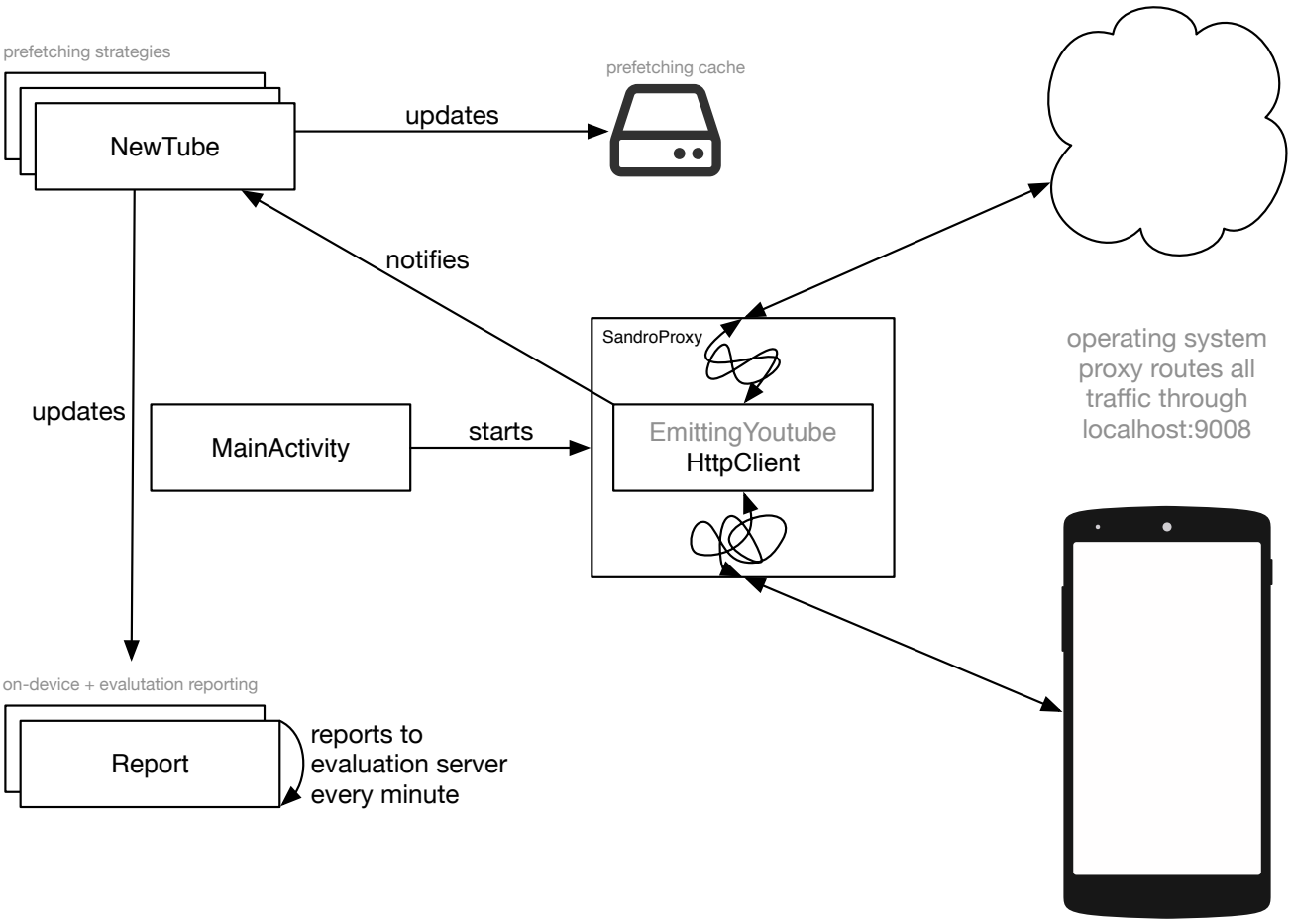

Figure 5.2.: Overview over the Android application structure

Once it is started for the first time, the MainActivity guides the user through four onboarding dialogs to set up everything that can not be set up automatically – a linked YouTube account, the system-wide proxy and the custom root certificate – and to introduce him to the usage of the application. Figure 5.1 and section 5.1.3 provide more details on this.

Using the white *i* button in the top right of the action bar, the user can re-init the onboarding dialog at any time – e.g. because he logged in to a new WiFi connection and needs to remember the correct system proxy settings to set it up in the right way (section 4.2.1 shows the steps involved in the process).

The center region of the screen displays the simulated prefetching strategies in a collapsible list. Once the user expands any of the prefetching strategies, he is provided a very short description of what the strategy is all about together with a short link to the paper that proposed it – like

> *"When watching a video, prefetch the first 10 seconds of 3 videos in the related videos list that you don't have prefetched yet. goo.gl/QCRp9A"*

for NetTube (3.1) – as well as a basic live performance measure of this strategy, the "hit rate":

$$\text{hitrate}(\textit{cache}) = \frac{\text{number of requests the } \textit{cache} \text{ could have answered}}{\text{total number of requests}}$$

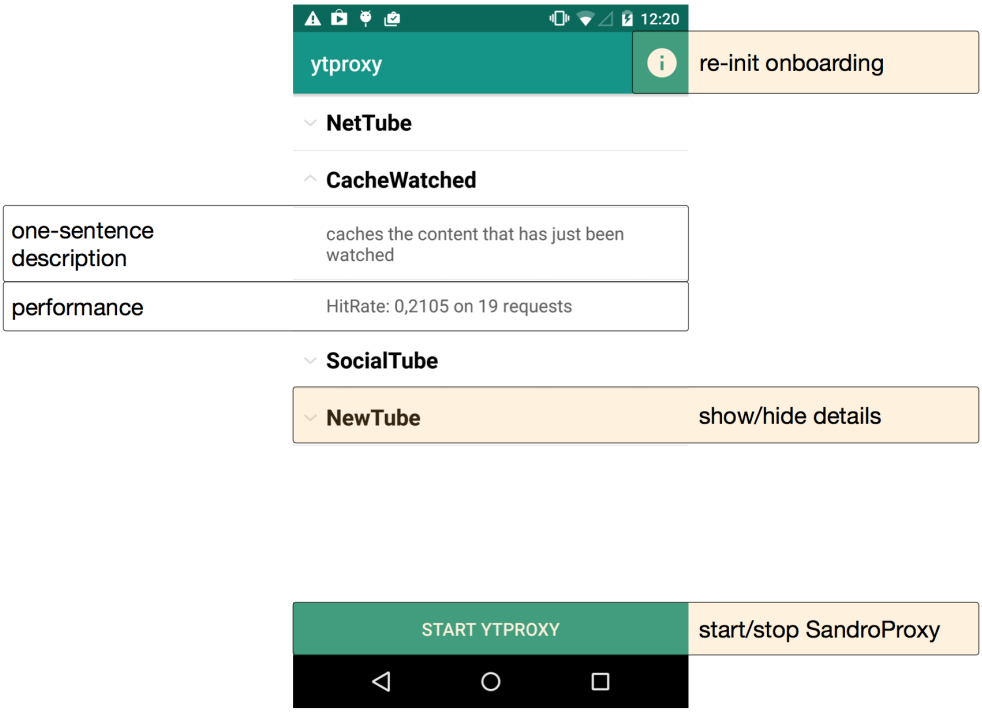

Figure 5.3.: MainActivity of the realized Android application, explained

SandroProxy

To actually read the http/https traffic the YouTube app generates, the open-source GNU GPL-v2 licensed library *SandroProxy*[6] is used. It is essentially a port of the Java penetration testing tool *WebScarab*[7] to the Android environment, together with hooks for using the tool in ones application.

Figure 5.4 gives an overview about the internal flow in this library – see it as a zoom into the SandroProxy block of figure 5.2.

SandroProxy, as motivated in section 4.2.1, is used to not only read http traffic but also to break the encryption of https connections. This is necessary as the YouTube application talks to the YouTube servers solely via https. Therefore a *root certificate* is installed during the onboarding process (5.2.1), which is used to impersonate servers to client applications and client applications to servers in any https communication. With this certificate, a custom ssl certificate can be generated for every https connection, which allows the application to decrypt and read encrypted traffic. [MN14, p.4] describes this technique in detail, along with possible mitigation strategies. Applications could for example use *certificate pinning*, where the client does not trust any certificate trusted by any registered certificate authority but only certificates trusted by a list of certificate authorities hard-coded into the client's source code. The desktop version of *Chromium*[8] (the open source project behind Google Chrome[9]) for example uses certificate pinning since mid 2011 to secure the Google Mail[10] web client [Goo11].

Unfortunately, there is no option in the SandroProxy/WebScarab library to only decrypt some specific https connections – i.e. connections to a specified host like `*.googlevideo.com`. If that existed, the https man-in-the-middle attack could be accurately targeted, not intercepting connections that not have to be intercepted. There are in fact only two cases in which the https connection would have to be encrypted:

6 sandrop.googlecode.com, the SandroProxy project on Google Code, checked on 2015-04-28
7 github.com/OWASP/OWASP-Webscarab, the WebScarab project on GitHub, checked on 2015-04-28
8 chromium.org, the chromium projects, checked on 2015-04-28
9 google.com/chrome, the chrome browser, checked on 2015-04-28
10 mail.google.com, the google mail web client, checked on 2015-04-28

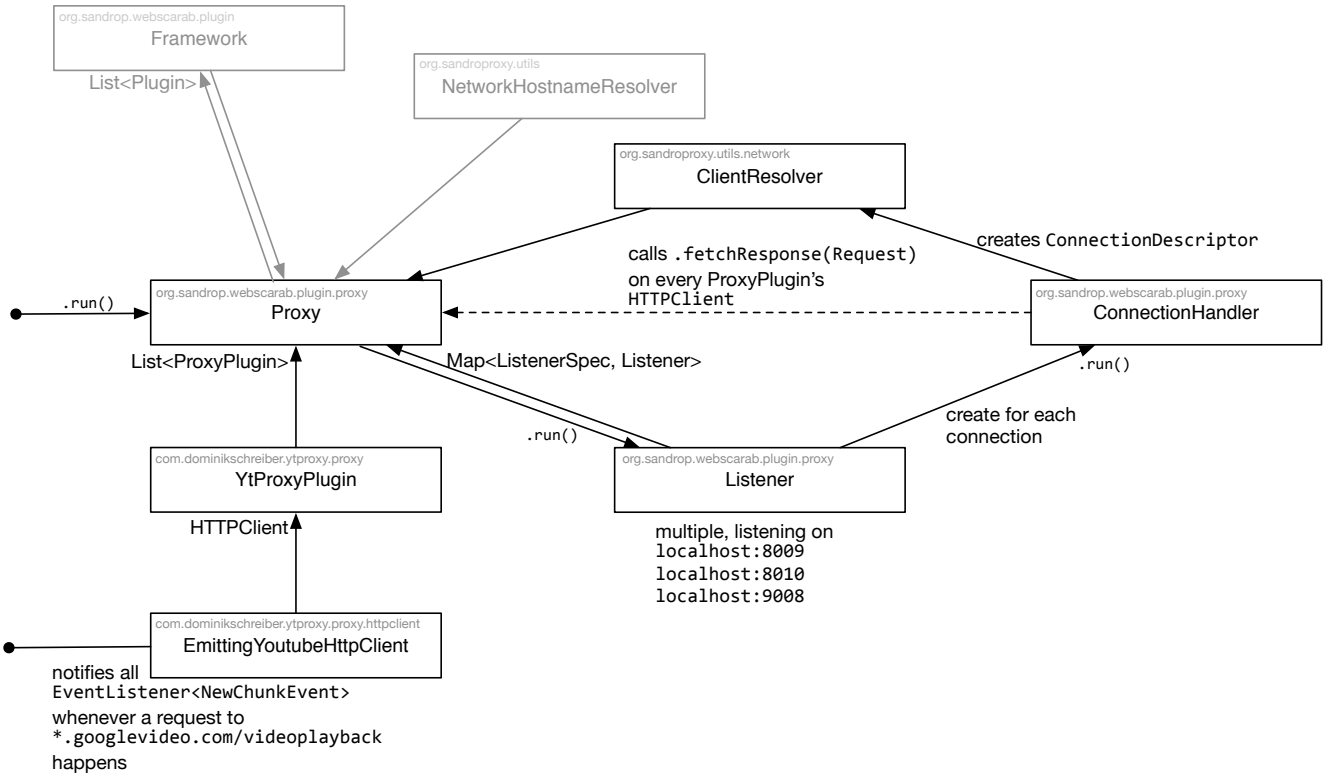

Figure 5.4.: The internal flow inside the SandroProxy library

CONNECT *.googlevideo.com: if this results in a `GET *.googlevideo.com/videoplayback` (targeting a random googlevideo.com subdomain), this is a request for an actual video chunk. So, this is the type of requests the application is most focused on.

CONNECT s.youtube.com: if this results in a `GET s.youtube.com/*?docid=...` (the actual path does not really matter, only the `docid` query parameter is important), this shows the YouTube video id of the currently requested video – a parameter that is unfortunately not present in the `*.googlevideo.com/videoplayback` requests.

Once the https traffic can be read, the main task of the SandroProxy library is to inform a custom `ProxyPlugin` with an HTTPClient that analyzes all traffic looking for requests to the aforementioned urls.

HTTPClient

The `.proxy.httpclient.EmittingYoutubeHTTPClient` is the actual workhorse in the SandroProxy using environment. It not only informs all prefetching strategies – and whoever else registered as a `EventListener<NewChunkEvent>` – about incoming video chunk requests, but also offers means to answer those requests *directly*, without forwarding them to the YouTube servers.

A large part of this response creation is getting all http headers right. Listing 5.1 shows a sample request with the full url and headers, together with the response from the YouTube servers:

Listing 5.1: Sample video chunk request

```
# URL with query parameters
https://r19---sn-4g57kndl.googlevideo.com:443/videoplayback
    ?itag=136
    &sver=3
    &lmt=1428688475063757
    &ipbits=0
    &key=yt5
    &mime=video%2Fmp4
```

```
    &expire=1428950802
    &gir=yes
    &nh=IgpwcjAxLmZyYTA1KgkxMjcuMC4wLjE
    &requiressl=yes
    &source=youtube
    &pl=20
    &initcwndbps=1957500
    &dur=209.417
    &fexp=11200028%2C900720%2C907263%2C924645%2C932627%2C932631%2C934954%2C9405988
          %2C9406708%2C9408093%2C9408246%2C9408347%2C9408704%2C9408788%2C9408919
          %2C940940%2C947243%2C948124%2C948703%2C951703%2C952612%2C957201%2C961403
          %2C961404%2C961406
    &sparams=clen%2Cdur%2Cgir%2Cid%2Cinitcwndbps%2Cip%2Cipbits%2Citag%2Ckeepalive
             %2Clmt%2Cmime%2Cmm%2Cms%2Cmv%2Cnh%2Cpl%2Crequiressl%2Csource%2Cupn
             %2Cexpire
    &ip=91.67.12.14
    &clen=41540402
    &keepalive=yes
    &id=o-AMhmJNX3p1W3kF73BTIuwDXqPgh2Ih3l5g4brtLdJmal
    &upn=dHCCpFBXSsE
    &mm=31
    &signature=0F78ED7DE82549C6C53184909FEDD12B0E6FD9CE
               .938009D643928F473E4A6DC501C73BFBECDD5374
    &ms=au
    &mv=m
    &mt=1428929075
    &dnc=1
    &cpn=EMQZf2uQegMhqotp

# Request headers
User-Agent: com.google.android.youtube/10.11.55(Linux; U; Android 5.1; de_DE;
            Nexus 5 Build/LMY47I)
Host: r19---sn-4g57kndl.googlevideo.com
Accept-Encoding: identity
Range: bytes=1221-2283074
Connection: Keep-alive

# Response headers
Server: gvs 1.0
Content-Range: bytes 1221-2283074/41540402
X-Content-Type-Options: nosniff
Last-Modified: Fri, 10 Apr 2015 17:54:35 GMT
Date: Mon, 13 Apr 2015 12:46:45 GMT
Alternate-Protocol: 443:quic,p=0.001
Accept-Ranges: bytes
Cache-Control: private, max-age=21297
Expires: Mon, 13 Apr 2015 12:46:45 GMT
Content-Length: 2281854
Content-Type: video/mp4
Connection: keep-alive
```

In the analyzation of multiple request/response pairs, it was observed that the response headers either

- follow directly from query parameters and request headers – e.g. `Content-Length: 2281854` is essentially `{{to}} - {{from}} + 1` in `Range: bytes=1221-2283074`,

- can be derived from the current state – e.g. `Date: Mon, 13 Apr 2015 12:46:45 GMT` is the very date the response was created,

- or can be chosen accordingly – e.g. `Last-Modified:` can always be set to the current date.

The response body – the actual video bytes – follows the *MPEG-DASH* standard for adaptive-bitrate multimedia streaming [Sod11].

With the given time constraints on this thesis it was not possible to realize actual prefetching and immediate answering of requests for prefetched content, but this is merely a well-known task and is left open for further work. One could for example take *Squid*[11] as an orientation and realize the actual prefetching from there.

Event-based architecture

To distribute information across the application, an event-based architecture following the *Observer Pattern* [GHJV94, p.293-304] was chosen. Java comes with a builtin `java.util.Observable` base class, but it has major drawbacks – like missing generics support. Therefore all communication is done via `.async.Emitter<E>`s that call `.async.EventListener<E>`s `#on(E)` with `E` being a subclass of `.async.Event`. This allows code reuse through subclassing and the utilization of Java generics.

Most importantly, the `.proxy.httpclient.EmittingYoutubeHTTPClient` is an `.async.Emitter<NewChunkEvent>` with a list of prefetching strategies (which are `.async.EventListener<NewChunkEvent>`s) and the `.statistics.ChunkHistory` – all configured in the `.ui.MainActivity`. So, whenever a video chunk is requested, all prefetching strategies get updated as well as the history of watched chunks (which is only present for evaluation purposes).

Together with the prefetching strategy updates, the on-device performance evaluation gets updated as well. Each strategy is in charge of updating its counts for cache hits/misses accordingly, and `.evaluation.PerformanceEvaluators` use this information to compute a *score* between 0 (worst) and 1 (best). This score is shown in the main activity along with the count of requests it is computed from (see figure 5.3).

5.2.2 Prefetching

5.1.1 motivated that evaluating video prefetching strategies is the core goal of this thesis. Therefore, an extensible architecture for prefetching strategies was realized:

A base `.prefetching.strategy.YouTubePrefetchingStrategyStub` abstracts all tasks common for all prefetching strategies and leaves only the implementation of `#process(NewChunkEvent, Supplier<Boolean>)` open to the concrete subclasses. This method is called whenever a new chunk is requested (5.2.1 shows how the event bubbles from SandroProxy to the strategies). It is to note that its execution may be asynchronous – e.g. a strategy requests the YouTube API (like described in 5.3). Therefore, to mark a video id as processed, one can not rely on return values but instead has to provide a callback – the `Supplier<Boolean>`. An implementation may call this function to indicate that it does not want to be informed about subsequent requests for the video id in the `.async.event.NewChunkEvent`. This prevents unwanted multiple executions that e.g. lead to multiple YouTube API requests and therefore an enormous performance drop. The prefetching strategy caches get notified anyhow and update their information independently.

[11] squid-cache.org, the Squid project, checked on 2015-04-28

5.2.3 File-system management

To actually store prefetched video chunks – or at least simulate it – a `.fs.cache.Cache` class hierarchy was introduced. It uses a subclass of `.fs.replacement.ReplacementAlgorithm` to stay smaller than `Cache#mMaxBytes`.

The most notable cache subclass is `.fs.cache.SimulatedCache`, that does not cache anything but just simulates doing so (again, actual prefetching is left open as described in 5.2.1). It maintains a map of `videoid` ⇒ `mimetype` ⇒ `[range]` mappings, meaning: for each videoid it has a list of "cached" ranges for each mimetype. Once the cache is requested to `#cache(Chunk,byte[])` a video chunk, the SimulatedCache tries to merge the new byte range into already cached ranges (copying stored request timestamps for the ranges), and finally calls its replacement algorithm to clean up.

There is a strange observation in the communication between the YouTube application and the corresponding servers: *the video chunk requests do not contain the video id* but only a session-based id. So it may happen that a `NewChunkEvent` bubbles with an empty video id. The public video id can be read from a separate request that may be answered after the video chunk request. So, the cache needs to clean up video chunks that do not have an id. Therefore, whenever a chunk *with* a video id comes in, the cache checks if there are any chunks *without* an id. It then assumes that they arrived right before this request – otherwise they would have been found in a previous search – and belong to the same video id.

5.2.4 Prefetching strategy evaluation

On-device evaluation of the simulated prefetching strategies is a great benefit arising from the amount of user-specific data collected on the device. It allows meaningful evaluation without any privacy breach as the collected data will not be sent out of the single device.

Therefore, a class hierarchy of *PerformanceEvaluator*s was created that need only to implement a `#score(int,int)` method that calculates a score from the count of cache hits and misses (which are obtained from the prefetching strategies). Any evaluator only needs to be added to the `mEvaluators` list of the `.ui.MainActivity` and will automatically be applied to all strategies whenever they update their hits/misses counts and displayed on the main screen of the application (see Figure 5.3).

For demonstration purposes, a simple `HitRate` evaluator was realized, that gives the percentage of cache hits to total requests as its score – as shown in listing 5.2.

Listing 5.2: HitRate performance evaluator (in class `.evaluation.strategy.HitRate`)

```
public double score(final int hits, final int misses) {
  final double total = hits + misses;
  return (total == 0) ?
      1 : // no request made => perfect score
      hits / total;
}
```

This is only an early stage. Further work may build on this on-device evaluation and offer for example speedup or miss penalty scores [SS95]. Refactoring the `#score` methods to use the whole information from a prefetching strategies cache might turn out to be necessary.

For the evaluation of prefetching strategies as done in this thesis, on-device evaluation is only of particular interest. Pseudonymized information is reported to the evaluation server and evaluated across all evaluation devices (see 5.4). Future work may however fearlessly remove this reporting, as it will not interfere with the on-device activities.

5.3 Realized prefetching strategies

To be able to compare the proposed prefetching strategy to related work, some existing prefetching strategies have been adapted and realized for the scenario of prefetching YouTube on a single device. Chapter 3 describes them in detail.

The main challenge when adapting those strategies is, that they were designed for prefetching in peer-to-peer networks, not for prefetching on a single device. Hence, they need to be interpreted in a way that they can be used on such a device. Obviously, while this interpretation tries to be as close to the source as possible, the results have to be taken with a pinch of salt. It can not be expected that the so-realized strategies deliver the same performance as their originally described parents, because every interpretation has to cater some trade-offs.

Another challenge was to request metadata from YouTube. There is a well documented, feature-rich API[12], with client libraries for many well-known programming environments, including Android. But for the relatively small amount of operations that are needed to realize the prefetching strategies, this API and the client library were fairly verbose and unsorted. Therefore, a wrapper for the client library was realized, that allowed limited operations in a convenient way – listing 5.3 gives an example.

Listing 5.3: Example call of the YouTube API wrapper

```
new YouTube.Channels(
  // configure the request by setting optional parameters
  // (uses default values if nothing specified)
  new YouTubeTask.Options.Builder().setMaxResults(5l).build(),
  // success callback: channels is the list of channels
  // matching the channel uuid the request is executed with
  (channels) ->
    _.each(channels, (c) -> Log.d("Channels", c.getSnippet().getTitle())),
  // error callback: expected to return a graceful value
  // (i.e. not null but an empty list)
  (exception) -> {
    Log.w("Channels", exception.getMessage());
    return Collections.emptyList();
  }
).execute("UC2OUuOe5Hx6FKy7EXr-_rvA"); // a channel id
```

5.3.1 Baseline: cache watched videos

One of the most simple strategies to select content to cache on the local device is to *prefetch nothing* but *keep watched content in cache* as long as possible. This may not be called a prefetching strategy in essence, but every prefetching strategy that performs worse than this canonical approach should really be reconsidered.

This simple approach has been realized without much effort, as shown in listing 5.4:

Listing 5.4: Realization of cache watched strategy (in class `.prefetching.strategy.CacheWatched`)

```
@Override
protected void process(final NewChunkEvent event, final Supplier<Boolean> done) {
  if (!mCache.isCached(event.chunk))
    mCache.cache(event.chunk, new byte[] {});
}
```

Evaluating the realized strategies shows that this simple approach in fact worked surprisingly well (6.4).

[12] developers.google.com/youtube/v3, YouTube Data API (v3), checked on 2015-04-28

5.3.2 NetTube: prefetch three related videos

The prefetching strategy proposed in [CL09] (referred to as *NetTube*), described in (3.1), was chosen as a promising competitor, as it performs exceptional – reportedly 90% prefetching accuracy after five videos – with comparably low efforts.

To realize NetTube, some adjustments had to be made:

NetTube is peer-to-peer-based. Each client "caches all its previously played videos" to share them with other clients [CL09, p.4]. There is no peer-to-peer component in the problem of this thesis, but cache size is of great importance, as there is not much storage available for caching on mobile devices. So, caching previously played videos is *not done* with this purpose. A watched video might stay in cache until the cache is full, but will never be held in cache to give it to other devices.

Furthermore, the peer-to-peer overlay creation of NetTube – "functional" lower-layer and "social" upper-layer – as well as the indexing of cached videos for search are not adopted, for the same reason that the algorithm will run on one isolated device, not in a peer-to-peer setting.

NetTube suggests to prefetch the first 10 seconds of videos. The application developed in this thesis however works with byte ranges directly, not with the more abstract concept of seconds. Moreover, these byte ranges are compressed and split into a video and an audio stream (see 5.2.1 for details on the used protocol). To adequately convert "10 seconds" to "*m* bytes of video, *n* bytes of audio", 30 YouTube videos were analyzed and their byte ranges after 10 seconds were collected and averaged. Table A shows the details. As the result, the implemented algorithm does not simulate prefetching a "10 second" prefix, but the specific bytes `0-2204380` of video and `0-160785` of audio content.

The realization of the NetTube strategy, after all abovementioned issues had been addressed, was as follows: "when a new video is requested, request the list of related videos from YouTube, and prefetch the first `2204380/160785` bytes of the first three videos not already cached". Listing 5.5 shows the actual code:

Listing 5.5: Realization of the NetTube strategy (in class `.prefetching.strategies.NetTube`)

```
@Override
protected void process(final NewChunkEvent event, final Supplier<Boolean> done) {
  new YouTube.Related(
    new YouTubeTask.Options.Builder()
        .setParts(_.list("id"))
        .setMaxResults(20l)
        .build(),
    (result) -> {
      new _<>(results)
          // mIsCached, a Predicate<SearchResult>, is a superclass member
          // that returns true if the video id is already in the cache
          .reject(mIsCached)
          .first(3)
          // mCachePrefixes, a Consumer<SearchResult>, is a superclass member
          // that actually caches the 2204380/160785 byte
          // prefixes of the specified videos
          .each(mCachePrefixes);
      done.get();
    },
    // superclass function that (creates a Function<Exception,List<T>> that)
    // logs the exception and returns Collections.emptyList()
    logException(SearchResult.class)
  ).execute(event.chunk.id);
}
```

5.3.3 SocialTube: prefetch followers and interest-clusters

Where NetTube (5.3.2) was relatively easy to adapt and realize, *SocialTube* – the prefetching strategy proposed in [LSW+12] – was not. But, as the performance in the evaluation of the original paper was more than twice as good as NetTube, it was decided to realize it as another evaluation reference value.

SocialTube was designed for peer-to-peer video sharing of YouTube videos that are embedded and watched in other social networks – namely facebook[13]. Adaptions to use it for the YouTube application on a single device, with the social graph provided by YouTube directly, and without the peer-to-peer component, had to be made.

In the "social graph" of YouTube, every user is a *channel*. Some primarily watch videos – that's what we would typically think of when thinking of a user –, some mainly publish. SocialTube has the concepts of *non-followers* and *followers*. Both are friends in facebook, that watch nearly *all* videos a user posts (followers) or watch all videos of interesting categories a user posts (non-followers) (3.2). To realize those, the channel subscriptions of YouTube were used. The *follower ratio* – the ratio of videos a user has to watch before being considered a follower – was set to 80%, analogous to $T_h = 80\%$ in [LSW+12, p.2887 O2]. But as T_l is not specified in SocialTube, the *non-follower ratio* has been set to 30% without model in SocialTube.

The high performance of SocialTube is partly due to the *push-based* manner in which videos are published. Once a user uploads a video, he actively pushes the first prefix to all his followers, and all his non-followers that follower the interest category the video is in. As the single Android device has no control over all channels the user follows/non-follows, this push-based approach can not be realized. There are efforts to deliver push-based information from the YouTube API[14], but at the time of writing, this has not made its way into the public YouTube API. Push emulation methods – like *long polling* and *streaming* [LSASW11] – can be applied neither, as this would need changes in the YouTube API servers behavior. Client-side polling would create unacceptable network traffic. Therefore the realized strategy updates, whenever a video is watched, the prefetching cache for the channel that published the current video.

The strategy is now realized as

> *When a video is watched, find the channel that published it, get the number of videos it published, compare it with the number of videos the user watched from this channel. If this is above* `FOLLOWER_RATIO`*, prefetch prefixes of* all *of the channels videos. If this is above* `NONFOLLOWER_RATIO`*, prefetch prefixes of all of the channels videos* of categories of interest *for the user.*

To find the categories of interest – here the official YouTube categorization is used –, whenever a video is watched, its category gets counted. The most-watched four categories (similar to [LSW+12, p.2887 O4]) are considered interest categories of the user.

SocialTube recommends to prefetch the *first chunk* of a video. As with NetTube, as the proxy application works directly on the byte range level, this has to be "translated". For simplicity, the same byte ranges as from NetTube, 2204380 bytes of video, 160785 bytes of audio are used.

Listing 5.6 shows an excerpt of the final strategy:

Listing 5.6: Realization of SocialTube (in class `.prefetching.strategy.SocialTube`)

```
@Override
protected void process(final NewChunkEvent event, final Supplier<Boolean> done) {
  final Consumer<Video> processVideo = (video) -> {
    // update watched category count
    final String categoryId = video.getSnippet().getCategoryId();
```

[13] facebook.com, checked on 2015-04-28

[14] published in a talk during Google I/O 2013, live recording at youtu.be/NlZZghBnfdM, checked on 2015-04-28

```
    count(mVideosPerCategory, categoryId);

    // update watched videos per channel count
    final String channelId = video.getSnippet().getChannelId();
    final int channelCount = count(mVideosPerChannel, channelId);

    new YouTube.Channels(
      new YouTubeTask.Options.Builder().setParts(_.list("statistics")).build(),
      (channels) ->
        _.each(channels, (channel) -> {
          // update channel video counts
          int channelVideoCount = channel.getStatistics().getVideoCount().intValue();
          mChannelVideos.put(channel.getId(), channelVideoCount);

          if (isFollower(channelCount, channelVideoCount)) {
            requestAndCacheAllPrefixes(done, channelId);
          } else if (isNonFollower(channelCount, channelVideoCount)
              && isInCategoryCluster(categoryID)) {
            requestAndCacheAllPrefixes(done, channelId, categoryId);
          }
        }),
      logException(Channel.class)
    ).execute(channelId)
  }

  new YouTube.Videos(new YouTubeTask.Options.Builder()
        .setParts(_.list("snippet"))
        .setMaxResults(1l)
        .build(),
    (videos) -> _.each(videos, processVideo),
    logException(Video.class)
  ).execute(event.chunk.id);
}

private void requestAndCacheAllPrefixes(
    final Supplier<Boolean> done,
    String... params) {
  final Consumer<List<SearchResult>> cacheAllPrefixes = (searchResults) -> {
    new _<>(searchResults)
        .reject(mIsCached)
        .each(mCachePrefixes);
    done.get();
  };

  new YouTube.ChannelVideos(new YouTubeTask.Options.Builder()
      .setMaxResults(50l)
      .build(),
      cacheAllPrefixes,
      logException(SearchResult.class)
  ).execute(params); // YouTube.ChannelVideos filters by category == params[1]
                     // if this is set
}
```

5.3.4 NewTube: prefetch YouTube personalized recommendations

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
> *— Phil Karlton*

This is especially true for the prefetching strategy proposed in this thesis. For lack of a better name, we will call it *NewTube* all along. In essence, it is proposed to leverage the personalized recommendations diverse video sharing services, lead by YouTube, offer.

Speaking of YouTube, one can access personalized recommendations using the YouTube Data API (v3)[15]. Any Android application leveraging this needs to register itself as an application in the Google Developers Console[16] and request to use the API. It is then able to authenticate its users with their Google Plus/YouTube accounts via OAuth2 [Har12], once they gave permission to do so in a consent dialog. With an authenticated user, the application can request `https://www.googleapis.com/youtube/v3/activities?home=true` (using the users OAuth2 token in an `Authorization: Bearer <token>` header) and filter the results to only contain items with `snippet.type == "recommendation"`. If the API is requested with `part=id,snippet,contentDetails` as a query parameter, one can even see the *reason* and *seedResourceId* that are responsible for the recommendation (as described in 3.4) in the `contentDetails` of the recommendation.

The *NewTube* prefetching strategy is now realized as

> *Whenever a new video is requested, prefetch the first 10 seconds of up to 50 personalized recommendations for the authenticated user.*

The video prefixes are kept as long as the cache is not exceeded, and are replaced using a least frequently used replacement algorithm [PB03, p.379-380]. With an assumed 256 MiB cache and an average of $2.10_{22605896}$ MiB (video) + $0.15_{3336525}$ MiB (audio) = $2.25_{55971146}$ MiB per resource (see 5.3.2), 113 resources can be prefetched before the cache replacement needs to take place.

To request more than 50 activities from the YouTube API, one has to use pagination tokens as every request may only return up to 50 results. A useful abstraction in the YouTube API wrapper classes (see 5.3) would be to allow any value for `.setMaxResults(Long)` and paginate with 50s steps through the API responses until the value is reached. Unfortunately, the YouTube API does not support filtering `snippet.types` server-sided, so there will always be `playlistItems`, `uploads`, `bulletins` and the like that need to be filtered out client-side.

The precise prefetching strategy is shown in listing 5.7:

Listing 5.7: Realization of the NewTube strategy (in class `.prefetching.strategy.NewTube`)

```
@Override
protected void process(final NewChunkEvent event, final Supplier<Boolean> done) {
  new YouTube.Recommendations(
    new YouTubeTask.Options.Builder()
        .setMaxResults(50l)
        .setParts(_.list("id", "contentDetails"))
        .build(),
    (recommendations) ->
      new YouTube.Videos(
        new YouTubeTask.Options.Builder().build(),
        (videos) -> new _<>(videos)
            .reject((video) -> isCached(video.getId()))
            .each((video) -> cachePrefixes(video.getId())),
        logException(Video.class)
```

[15] developers.google.com/youtube/v3, the YouTube Data API documentation, checked on 2015-04-28
[16] console.developers.google.com, the Google Developers Console, checked on 2015-04-28

```
      ).execute(new _<>(recommendations)
          .map((recommendation) -> recommendation
              .getContentDetails()
              .getRecommendation()
              .getResourceId()
              .getVideoId())
          .value()
          .toArray(new String[] {})),
    logException(Activity.class)
  ).execute();
}
```

5.4 Evaluation infrastructure

To evaluate the realized prefetching strategies (5.3), a reporting infrastructure based on a RESTful web service [Fie00, p.76-106] was set up. Every minute, the Android application sends information about

- the caches of the prefetching strategies,
- the videos watched since the last report and
- the recommendations for the user made by YouTube

to the REST API at `http://yt.dominikschreiber.com/api/v1/`. The raw data is then be exposed at `http://yt.dominikschreiber.com` in a live dashboard view. It is furthermore accessed by a script that renders the graphs in the evaluation chapter (6) from the most recent data.

5.4.1 Data acquisition

With the architectural decisions described in 5.2, it is most important that each byte range (represented in `.fs.Range`) stores in itself the timestamp it was *created* at as well as a list of timestamps it was *requested* at. Combined with a global viewing history (a `.statistics.ChunkHistory`), one can retrieve all necessary information. A cache hit rate (5.2.1) could for example be computed as the number of requests all ranges of a strategy have in a certain time period relative to the number of entries in the viewing history for that period.

Data is actually a little bit more structured, with the following data types (shown as JSON [Cro06]):

Range: a single byte range for a single mimetype of a single video: listing 5.8

Listing 5.8: the Range datatype

```
{
    "from": Number // indicates the start index of the byte[] array
  , "to": Number // indicates the last index of the byte[] array
  , "added": Number // unix timestamp in milliseconds
  , "requested": [ Number ] // list of unix timestamps in milliseconds
}
```

Mimetype: a list of *Ranges* of a single video: listing 5.9

Listing 5.9: the Mimetype datatype

```
{
    "type": String // the mimetype, either "audio/mp4" or "video/mp4"
  , "chunks": [ Range ] // list of Ranges
}
```

Video: a list of *Mimetype*s that each store their own *Ranges*: listing 5.10

Listing 5.10: the Video datatype

```
{
    "videoId": String // sha256-Hash of the public YouTube video id
  , "mimetypes": [ Mimetype ] // list of Mimetypes
}
```

Strategy: a list of *Video*s that show the cached state at a specific time: listing 5.11

Listing 5.11: the Strategy datatype

```
{
    "title": String // the title of the Strategy, e.g. "NetTube"
  , "cache": [ Video ] // list of Videos
}
```

HistoryEvent: a range of a mimetype of a single video requested at a specific time: listing 5.12

Listing 5.12: the HistoryEvent datatype

```
{
    "timestamp": Number // unix timestamp in milliseconds
  , "videoId": String // sha256-Hash of the public YouTube video id
  , "mime": String // the requested mimetype, "audio/mp4" or "video/mp4"
  , "range": {
        "from": Number // start index of the requested byte[] array
      , "to": Number // last index of the requested byte[] array
    }
}
```

Recommendation: a recommended video as reported by the YouTube API: listing 5.13

Listing 5.13: the Recommendation datatype

```
{
    "videoId": String // sha256-Hash of the recommended video id
  , "reason": String // explanation of the recommendation, one of
                     // "unspecified", "videoFavorited",
                     // "videoLiked" or "videoWatched"
  , "seedVideoId": String // sha256-Hash of the video that caused
                          // the recommendation
}
```

In the communication between the Android application and the evaluation server, the JSON serialization format is used. This means, the Java data structures need to be serialized to be reported. Therefore, multiple *Renderers* were created that fulfill this job by turning the data structure into a structure of `Maps`/`Lists`, that are then turned into JSON strings using `_.stringify(Object)`. Listing 5.14 shows exemplarily how this is done for the `ChunkHistory`:

Listing 5.14: serializing a `ChunkHistory` to JSON (in class `.statistics.renderer.HistoryJsonRenderer`)

```
public static String render(final ChunkHistory chunks,
                            final Pseudonymizer<String> p14n) {
    return new _<>(chunks.get())
            .map((e) ->
                _.stringify(_.dictionary(
                    _.entry("timestamp", Long.toString(e.timestamp, 10)),
                    _.entry("videoId", p14n.pseudonymize(e.chunk.id)),
                    _.entry("mime", e.chunk.mime),
```

```
                    _.entry("range", _.stringify(_.dictionary(
                        _.entry("from", Integer.toString(e.chunk.range.from, 10)),
                        _.entry("to", Integer.toString(e.chunk.range.to, 10))
                    )))
                )))
            .stringify();
}
```

Every minute, the MainActivity starts a `.statistics.Report` that collects the most recent information – the history, the raw recommendations and the states of the prefetching strategy caches – creates JSON strings from them using the corresponding JsonRenderers (if anything, specifically the `#hashCode()`, changed since the last report) and reports them to the evaluation server using http `POST` requests.

5.4.2 Aggregating data on the server

The RESTful API to collect the data on the server is exposed at `http://yt.dominikschreiber.com/api/v1`. It runs on io.js[17] and uses a CouchDB[18] key/value-storage to persist the data.

With *express*[19] as a web framework, *nano*[20] as the CouchDB wrapper and *express-persistent-resource*[21] (see 5.5.2) for automated REST service creation, the total server was created in 26 SLOC – shown in listing 5.15.

Listing 5.15: the evaluation server REST API (in `lib/api.js`)

```
var express = require('express')
  , resource = require('express-persistent-resource')
  , db = require('nano')(require('./dburl'))
  , api = express.Router()
  , resources = {
        strategy: {
            fields: 'uuid,entry:(title,cache:(videoId,mimetypes:' +
                    '(type,chunks:(from,to,added,requested))))'
        },
        history: {
            fields: 'uuid,entry:(timestamp,videoId,mime,range:(from,to))'
        },
        information: {
            fields: 'uuid,entry:(recommendation:(videoId,reason,seedVideoId))'
        },
        uuid: {
            id: function() {
                return [
                    (new Date()).getTime(),
                    (Math.PI + Math.random()).toString(36).slice(2, 12)
                ].join('-');
            }
        }
    };
for (var r in resources) if (resources.hasOwnProperty(r))
    api.use('/' + r, resource(r, db, resources[r]));
```

[17] iojs.org, the io.js project, a fork of Node.js (nodejs.org), both checked on 2015-04-28
[18] couchdb.apache.org, the Apache CouchDB project, checked on 2015-04-28
[19] expressjs.com, the express io.js web framework, checked on 2015-04-28
[20] github.com/dscape/nano, the nano io.js CouchDB wrapper, checked on 2015-04-28
[21] github.com/dominikschreiber/express-persistent-resource, the express-persistent-resource CRUD-ready io.js express middleware, checked on 2015-04-28

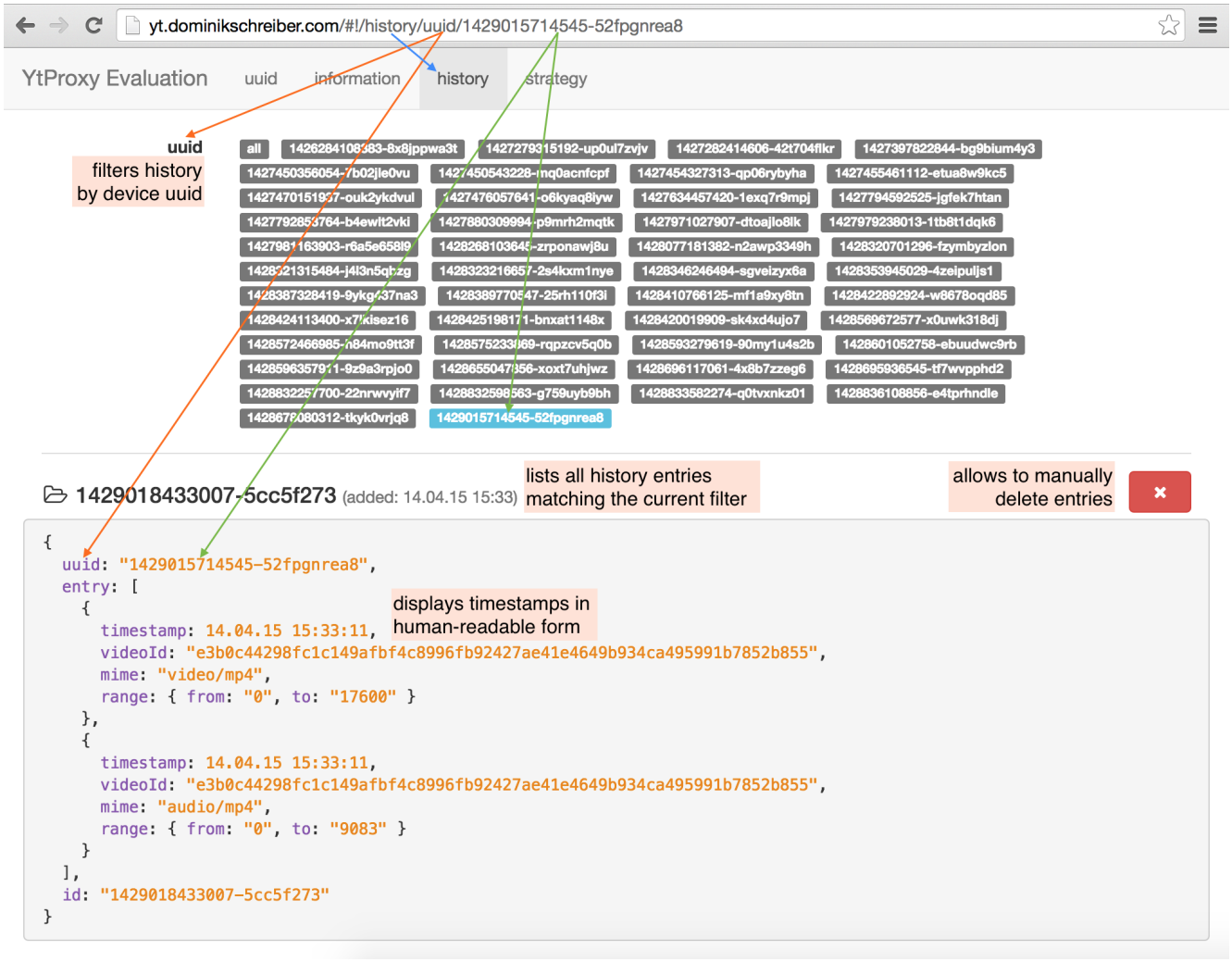

Figure 5.5.: The evaluation server dashboard for the history view, captured at 2015-04-16

```
module.exports = api;
```

The dashboard view at `http://yt.dominikschreiber.com` was created as a single-page application using AngularJS[22] and Bootstrap[23]. It builds itself nearly automatically – based on the data served by the REST server. One could add any API endpoint to the list of endpoints and with that one word added, one gets a new dashboard view for the data served by that endpoint, dynamically able to filter by all fields that are not unique in the served data. Currently this is only the `uuid` field, but more may come with further work. Figure 5.5 shows the dashboard for the history view.

To be able to identify devices without breaking anonymity, every application gets assigned a universally unique identifier (`uuid`) that is sent with every report. This uuid can not be resolved to a named user as it is only application but not device-dependent (especially does it not rely on the user/device-specific `ANDROID_ID`[24]).

22 angularjs.org, the AngularJS JavaScript framework, checked on 2015-04-28
23 getbootstrap.com, the Bootstrap css framework, checked on 2015-04-28
24 developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID, `android.provider.Settings.Secur` 64-bit hexadecimal string unique for each user on each device, checked on 2015-04-28

5.5 Noteworthy side products

In the process of creating the Android application, some modules were created so independently that they could live on as side projects without any further connection to the work of this thesis. One is *underscore.java* (5.5.1), a port of underscore.js[25] to the Java environment. The other is *express-persistent-resource* (5.5.2), a CouchDB-backed io.js express middleware creating a RESTful resource API.

5.5.1 underscore.java

Java 8 [GJS+15] brings the joy of functional programming to the Java world. On Android however, one is doomed to use Java 7 until the build tools are upgraded – and even this was only introduced in late 2013, more than two years after Java 7 was released. To be able to use functional programming anyways, the library *underscore.java* was created.

In its methods, and in its general appearance, it tries to resemble the *underscore.js* library, which brings the same functionality to JavaScript (with polyfills for legacy environments and native language features where possible). It uses equivalents of the Java 8 functional interfaces (`java.util.function.*`), but defines them below the `com.dominikschreiber.underscore` namespace as the Android build tools do not allow adding classes in the `java` namespace to the project.

The `com.dominikschreiber.underscore._` class offers a wide range of well-known functional programming vocabulary like the `_.filter(values, predicate)`, `_.map(values, transformation)` and `_.reduce(values, combination, initial)` recursion abstractions, as well as a ton of utility functions like `_.range(from, to, stepwidth)` (to create a list of integers), `_.list(args)` (to create a list of the function parameters) or `_.join(values, separator)` (to join a list of strings). Even Java reflections are used to `_.extend(defaults, extension)` an object with values from another.

For quick usage, any function is provided as a static member of _, whereas _ can be instantiated with a list of values (precisely: any `java.lang.Iterable<T>` will work) to chain method calls – it then provides a `#value()` method that returns the un-wrapped result of the computation. Listing 5.16 shows how both can be used.

Listing 5.16: Example usage of _

```
// static method calls
String secretMessage = _.join(
  _.map(
    _.list(12, 0, 15, 15, 14, 14, 11, 10, 11, 14),
    (num) -> Integer.toString(num, 16)
  ),
  ""
);
// => secretMessage = "c0ffeebabe"

// chained method calls
// (example from .prefetching.strategy.SocialTube#isInCategoryCluster)
boolean isInCategoryCluster = new _<>(mVideosPerCategory.entrySet())
    .sortBy(e -> (long) -e.getValue())
    .first(4)
    .map(e -> e.getKey())
    .contains(categoryId)
```

Pre Java 8 these methods internally translate to some foreach loop, `_.each()` is shown as an example in 5.17. However, adapting the underscore library to Java 8 will greatly improve the overall performance

[25] underscorejs.org, underscore.js – the functional swiss army knife of JavaScript, checked on 2015-04-28

without changing a single line of production code. Listing 5.17 shows as well how this can be done. One could offer multiple library builds for different Java versions and provide maximum speed at minimum effort.

Listing 5.17: Realization of `_.each()` with and without Java 8 features

```
// Java 6 version
public static <T> void each(Iterable<T> values, Consumer<T> function) {
  if (values == null) return;
  for (T value : values) function.accept(value);
}
// Java 8 version
public static <T> void each(Iterable<T> values, Consumer<T> function) {
  values.forEach(function);
}
```

The library was completely developed using the test-driven development approach. This leads to great confidence in the behavior of the library and makes future refactorings (like native Java 8 support) a joy of heart.

5.5.2 express-persistent-resource

[Mul12] characterizes industrial best practices in defining RESTful [Fie00, p.76-106] APIs. It occurs that these are resource-independent and repeatedly implementing them for new resources leads to a high portion of duplicate code.

Therefore, *express-persistent-resource* was created. It is a middleware for the Node.js/io.js web framework *express* that exposes a CRUD (create, read, update, delete [CST+10, p.7]) interface to the defined resource and handles everything from requests to content negotiation to data persisting independently.

The library uses a declarative approach for defining resources: a `fields` string defines resource properties and can be used to request partial responses as well. Listing 5.18 shows how this string is built. It omits data types as the underlying CouchDB key/value store uses JSON which does not have a strong type system.

Listing 5.18: The `fields` string in *express-persistent-resource*

```
// "," separates properties
"this,is,a,fields,string"
// => objects with matching properties are valid
//    (both values and lists of values are allowed):
// {"this": 0, "is": false, "a": 13.37, "fields": "?", "string": [true, false]}

// ":()" denotes nested properties
"this:(is,nested)"
// => nested objects with matching properties are valid:
// {"this": {"is": true, "nested": "yes!"}}

// example fields string for the Video datatype
"videoId,mimetypes:(type,chunks:(from,to,added,requested))"
```

Using this fields string logic, advanced API features like *partial responses* are easy to provide. *express-persistent-resource* provides them out of the box. A full list of features is available from the project page at GitHub: `https://github.com/dominikschreiber/express-persistent-resource`.

5.6 Conclusion

This chapter led through the realization of an Android application that uses a system-wide proxy to intercept the communication between the YouTube application and the YouTube servers. It does so to evaluate various prefetching strategies that perform on the low level of byte ranges of YouTube videos. For the sake of evaluating this thesis, an evaluation infrastructure was presented, that relies on minutely updates from the Android application and shows the pseudonymized collected data on a web dashboard.

A great part of the realized Android application was made solely to set up for further work. The raw evaluation server functionality would not need event-based communication inside the application, or even the modelling of prefetching strategies on the client. It would be enough to have a HTTPClient (5.2.1) that directly reports any video chunk request to an evaluation server, and everything else could be done on the server as well – like aggregating data or simulating prefetching strategies. But this work was not done in a devil-may-care manner. Extensibility and adaptability were key goals throughout the process.

5.6.1 Key challenges and their solutions

Encrypted traffic

The main challenge for the whole approach was that *YouTube uses https to communicate* with its servers. This is perfectly reasonable and users highly appreciate it. But for the research purpose of analyzing this traffic it is obviously a big hindrance.

SandroProxy (5.2.1) was a great tool for dealing with encrypted traffic. This library abstracts over the details of dealing with http connections, ssl encryption, local proxies et cetera.

However, intercepting a https connection by dynamically generating valid certificates for both the client and the server is a *time-consuming task*. It takes so much time, that, in a live environment, the first request for any YouTube video (made from the YouTube application) *times out*. The application interprets this as a "network error" and asks the user to "check its network connection". Once he reloads the content, the request succeeds – the certificate genertion just took so long that the first request timed out.

Within the given time, the best solution was to admit that *it is research*, that dynamic certificate generation *takes time* and to accept that the user has to click twice on each video to view it.

SandroProxy promises IPv6 but does not deliver

Another challenge was that during the evaluation period some ISPs switched their DNS hostname resolving from IPv4 to IPv6. That should not be a problem at all, as SandroProxy opens proxys for IPv6 connections as well as for IPv4 connections.

Unfortunately, it was a problem. Traffic did get routed into the application, but it was never answered as SandroProxy did not forward it.

Here, the modest solution was chosen: the evaluation devices were equipped with a third-party application[26] that *disables IPv6* for the active wifi connection. From then on the android device only used IPv4, which SandroProxy was fine with.

5.6.2 Outlook: further steps

Many elements were realized in the Android application only to support further work. Some practical steps include

- realizing *actual prefetching* – borrowing from Squid[27] could be a great entry point here

[26] play.google.com/store/apps/details?id=de.lennartschoch.disableipv6, [Root] Disable IPv6, checked on 2015-04-28

[27] squid-cache.org, the Squid cache project, checked on 2015-04-28

- improving *on-device evaluation* – taking `.evaluation.strategy.HitRate` as a simple start, one could offer various cache performance metrics for multiple strategies; refactoring `#score(int,int)` to `#score(Cache,int)` could be a good idea in that case
- improving *overall performance* – first thing could be to intercept only https `CONNECT` requests to domains that could provide any interesting information (see 5.2.1)

Overall speaking, the application realized in this thesis works well for research purposes and offers new ways of easily evaluating new prefetching strategies, but it is not ready for providing a good end-user experience.

6 Evaluation

In the previous chapters, the lifecycle of an Android application that simulates prefetching of YouTube videos has been discussed – from a motivation to create such application (1) over discussion of background information (2) and related work (3) to the design (4) and realization (5) of the application.

This all is helpful, but only an evaluation of the work leads to reliable results. Because of that, an evaluation based on the infrastructure described in (5.4) has been conducted.

What follows are a description of the evaluation setup (6.1), characteristics of the created dataset (6.2), the rationale behind the metrics used (6.3), the actual evaluation results (6.4) and a conclusion of the findings (6.5).

6.1 Setup

The evaluation of the data gathered using the infrastructure described in (5.4) was done using a python script with the *matplotlib* [Hun07] library.

The RESTful API offers a way to retrieve all documents for a resource with a single request (using the `?include_docs` query parameter) and to specify the file type the documents should be serialized to (using the `Accept:` header). Using this, the data was requested as `application/json` and immediately turned into a python dictionary using the builtin `json` library.

To provide intuitive access to the evaluation data, the three major types of `strategy`, `history` and `information` were structured as shown in Listing 6.1.

Listing 6.1: Data structure for the evaluation script

```
devices = {
  '{{uuid}}': { # a single device, e.g. 1429515823779-72vdle6hzo
    'strategy': {
      '{{title}}': { # title of the strategy, e.g. "NetTube"
        '{{timestamp}}': { # timestamp (in ms) the strategy cache was reported
          '{{videoid}}': { # sha256-hash of the prefetched video id
            'audio/mp4': [
              {
                'from': str,
                'to': str,
                'added': str, # timestamp (in ms)
                'requested': [str] # list of timestamps (in ms)
              },
              # ...
            ],
            'video/mp4': [
              # analogous to 'audio/mp4'
            ]
          },
          # ...
        },
        # ...
      },
      # ...
    },
```

```
    'history': {
      '{{videoid}}': { # sha256-hash of a video id
        'audio/mp4' {
          '{{timestamp}}': { # timestamp (in ms) the chunk was requested
            'from': str,
            'to': str
          }
        },
        'video/mp4': {
          # analogous to 'audio/mp4'
        }
      },
      # ...
    },
    'information': {
      'recommendation': {
        '{{timestamp}}': [ # timestamp (in ms) the recommendations were obtained
          {
            'reason': str, # reason the video is recommended
            'seedVideoId': str, # sha256-hash of the seed video id
            'videoId': str # sha256-hash of the recommended video id
          },
          # ...
        ],
        # ...
      }
    }
  ,
  # ...
}
```

The acquisition data structures (5.4.1) were designed to create self-containing single reports. At evaluation time however, the data can be rearranged to be uuid (= device) dependent. This makes them more intuitively useable – although they will again be rearranged once they are plotted, as each plot needs its own data arrangement.

Plotting the evaluation targets was realized using *matplotlib* and abstracting the actual plot creation as much as possible – multiple plot "tasks" are run that each create `matplotlib.figure.Figures` that get plotted to PDF files centrally. And even the figure creation is abstracted as much as possible, where single tasks only create *strategy→value* mappings and let a lower-level method (`really_plot_bar_chart_for_strategies()`) perform the interactions with matplotlib.

6.2 Characteristics of the dataset

For the evaluation, data was gathered from 15 users (5 female, 10 male, an average of 23.93 years old) over a time span of 19 days. 4096 requests for 210 YouTube videos with a total of 1.63*GB* of content were captured.

Figure 6.1 shows how captured requests were distributed day-wise across the evaluation time span (on the left), and hour-wise across a single day – accumulated from all requests (on the right).

It is noteable that especially the requests→hour-of-day plot has two conspicuities:

- Requests are more or less gaussian distributed across the day (leaving out the night time of 1 AM until 6 AM, where people are assumed to sleep), but there is a significant *drop in the afternoon* (2 PM until 3 PM). It can only be guessed why this drop happened. However, as it happened

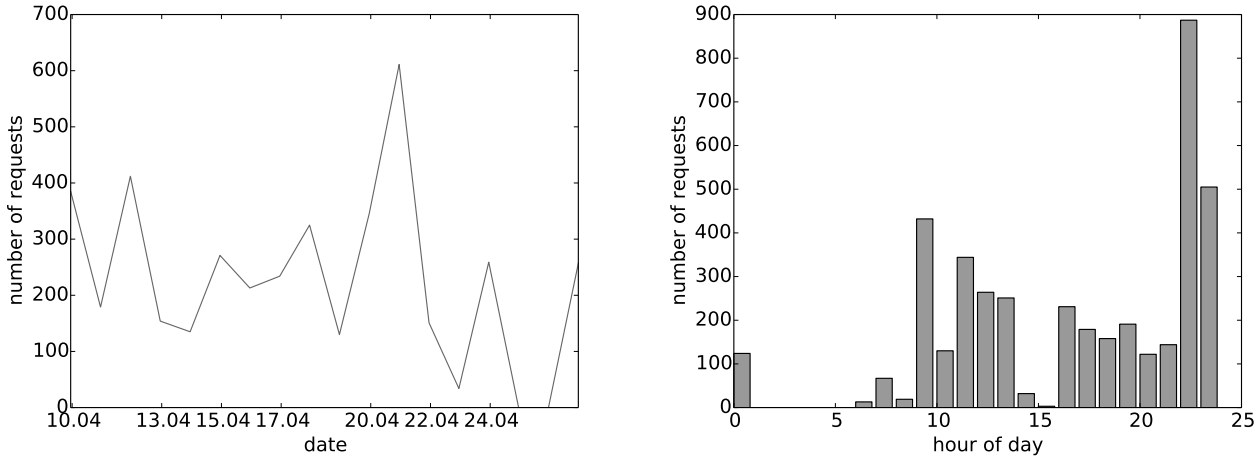

Figure 6.1.: Number of captured requests a) per day, b) per hour of day

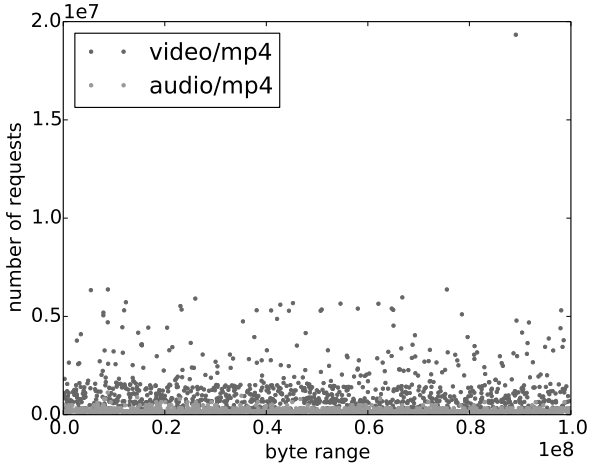

Figure 6.2.: Number of captured requests to specific byte ranges

right after the lunch break it could be that the probands were hard working and had no time for evaluation.

- There is a huge boost in requests in the late evening – after 9 PM. This may be explained with the fact that all users were voluntary and may have decided to watch some YouTube videos after all work of the day was done. Moreover, watching YouTube videos in bed before sleeping could be a common pattern that is fleshed out by the accumulated data.

To suggest a meaningful prefix length for prefetching strategies that only prefetch prefixes, the number of requests to byte ranges of videos is examined. Figure 6.2 shows their distribution. This is a plot of total requests to single bytes in 10000 byte buckets: for example the 50000-59999 bucket contains the number of all requests that requested bytes in the bucket, weighted by the requested byte range – e.g. the request `51000-51999` would count for 2000 requests (as 2000 bytes were requested once).

It is surprising that the requests are almost equally distributed. Prefix-prefetching strategies would assume that the first buckets (as long as the prefixes are) are requested more often than the rest. That means however that users watch whole videos instead of skipping through a list of videos until they find what they watch.

6.3 Evaluated metrics

The evaluation is led from the question "which prefetching performs best on a single mobile device for prefetching videos from YouTube". As the setting – on a single mobile device, not in a whole peer-to-peer network – is different to previous research, it is not clear wether strategies suited for peer-to-peer networks, like NetTube or SocialTube, work as well in this scenario.

Writing the main question out, other questions arise:

- "What is the impact of the media streaming protocol used?" – optimizing for a specific protocol would mean *overfitting* and falling into a deep pit once the protocol is changed.
- "When is the point, where the benefits of using a prefetching algorithm exceed its costs?" – costs are for example storage reserved for caching of chunks that may never be requested.

6.3.1 Evaluated strategies

The evaluated strategies are:

NetTube

Prefetching 10-second prefixes of three videos in the list of related videos once a video is watched. Section 3.1 discussed [CL09], which proposed the strategy. Section 5.3.2 showed how NetTube was realized in the Android application.

SocialTube

Prefetching 10-second prefixes of all videos in the follower and interest clusters. [LSW+12] is laid out in 3.2 and the realization is described in 5.3.3.

NewTube

Prefetching 10-second prefixes of up to 50 personalized video recommendations. The strategy is proposed in this thesis and its realization is shown in 5.3.4.

CacheWatched

Cache video chunks that have been already watched once until the cache is full. This is more of a baseline dummy than a real strategy, but the realization is described in 5.3.1.

6.3.2 Key metrics

For each strategy, two key metrics are evaluated:

Cache hitrate

$$\text{cache hitrate} = \frac{\text{number of answered requests}}{\text{number of all requests}}$$

The amount of requests the strategy cache would have answered relative to the amount of all requests in the time span. Yields a percentage value, higher is better. This metric is presumably the most meaningful in this evaluation, as it measures what the prefetching strategy *already does right*.

Prefetching accuracy

$$\text{prefetching accuracy} = \frac{\text{number of accessed videos}}{\text{number of cached videos}}$$

Number of videos in the strategy cache that have been accessed *at least once* relative to the total number of videos in the strategy cache. Again, yields a percentage value, the higher the better. A prefetching strategy performing well in this metric did a good job at predicting the video selection of the user, and could at least improve on the selection of which video chunks to prefetch. This means, the Video Hit Rate metric measures a "general direction" rather than hard statistics – if a well-performing strategy just updated the way it selects video chunks, it should get a large performance boost.

It has to be noted that more sophisticated cache evaluation metrics – from speedup gain to miss penalty [SS95] – could not be evaluated due to the fact that caches were always just simulated. Furthermore, the whole network intercepting infrastructure costs time as well, so a time-based miss penalty would hardly be accurate – requests were slowed down by the continuous man-in-the-middle attacks (5.2.1), and partialing this out of timing information would be impossible (or at least highly inaccurate).

6.3.3 Evaluation dimensions

To value prefetching strategies that perform right in the "general direction" but need more fine-tuning, metrics are evaluated in three sub-dimensions each:

Bare data

All gathered data is used. This favors strategies that already have a high cache hitrate. But it also factors in non-human factors like duplicate requests made by the YouTube application the user does not even know about.

No "re-requests"

To concentrate of the actual video viewing habits of users, metrics were evaluated only on requests that happened *after more than 10 seconds* since the last request to a single chunk. Keep in mind that this is chunk-based, so requesting the single chunk `0-100` (byte range) twice – in 10 seconds – will not count, but requesting `0-100` and `101-200` will.

This is necessary as video requests are captured between the YouTube application and the YouTube servers – not between the user and YouTube (5.2.1). Therefore, multiple requests arising from the use of the MPEG-DASH adaptive media streaming standard are captured as multiple requests, where it is only one request in the eye of the user.

Obviously, a prefetching strategy should focus on what *actually happens* (like the adaptive streaming), not what *should have happened*. Here, adapative streaming and prefetching interfere heavily. If prefetching was realized *directly* in the YouTube application, it could be utilized before any adaptive streaming requests are made – this would lead to the best results. But in the early stage of research for prefetching strategies on mobile devices, where strategies are only simulated, the focus should stay on the viewing habits of the users and not on prefetching for a single video streaming protocol – those may change, while users tend to stay the same.

only 10-second-prefix requests

Finally, metrics were evaluated only on requests that request a video chunk in the first 10 second prefix of videos. This is again done to evaluate the "general direction" of prefetching strategies rather than the concrete performance.

	bare data	only 10s prefixes	no re-requests
CacheWatched	0.0083984375	0.0333656644035	0.00118815104167
NewTube	0.000276692708333	0.00109925638539	0.000276692708333
NetTube	0.000797526041667	0.00316844487553	0.000797526041667
SocialTube	0.000764973958333	0.00303912059489	0.000699869791667

Table 6.1.: Cache hitrates of the evaluated prefetching strategies

Three of four evaluated prefetching strategies – all except *CacheWatched* – only prefetch 10-second prefixes. To see if they predicted the right videos, no matter if they predicted the right chunks of those videos, this dimension is measured.

6.4 Results

The results of evaluating four prefetching strategies (6.3.1) in two metrics (6.3.2) with three dimensions each (6.3.3) are presented in the following section.

6.4.1 Cache hitrate

First of all, the *cache hitrate* is measured. Figure 6.3 shows plots of the results for all considered dimensions, table 6.1 shows precise data. The major findings are that all prefetching strategies perform bad on a single mobile device, but that the CacheWatched "strategy" performs orders of magnitude better than all sophisticated prefetching strategies. More evaluation results flesh out the statement that this comes from requests due to the media streaming protocol.

Observation: all strategies perform bad

The first – and most astonishing – observation is that *all strategies perform bad*. Looking at the bare data, NetTube and SocialTube perform at around 0.8% of cache hits, NewTube even below 0.3%. This bad performance stands in contrast to the observations made in [CL09] and [LSW+12]. It can only be explained by the fact that both evaluations had a full peer-to-peer network to their hands, while in this evaluation only a single mobile device is investigated.

Observation: CacheWatched performed better

The second observation is that *CacheWatched*, the dummy caching scheme that was only introduced as "everything worse than this is not worth any further investigation", performed *an order of magnitude better* than all sophisticated prefetching strategies. Around 8.5% cache hitrate is not by any means perfect, but compared to 0.8% and 0.3%, this is a whole other level.

In fact, this goes with observations in other fields of research, where the naïve approach surprisingly works best. Those are for example *data mining*, where a Naïve Bayes Classifier performs equal to Support Vector Machines and the likes on large data sets [HLL03] [MS12], or even *polynomial multiplication* [Fat13].

Observation: cache hitrate is highly user-dependent

The confidence intervals of all cache hitrate measures reach from about $\frac{1}{5}$ of the average to 5 times the average. This means, the actual cache hitrate highly depends on the user that is served by the prefetching strategy. Some tend to use YouTube like prefetching strategies expect them to do, this is where the cache hitrate is vastly higher than average. But there are also users that obviously have a completely unexpected behavior, which drops cache hitrates to almost 0.

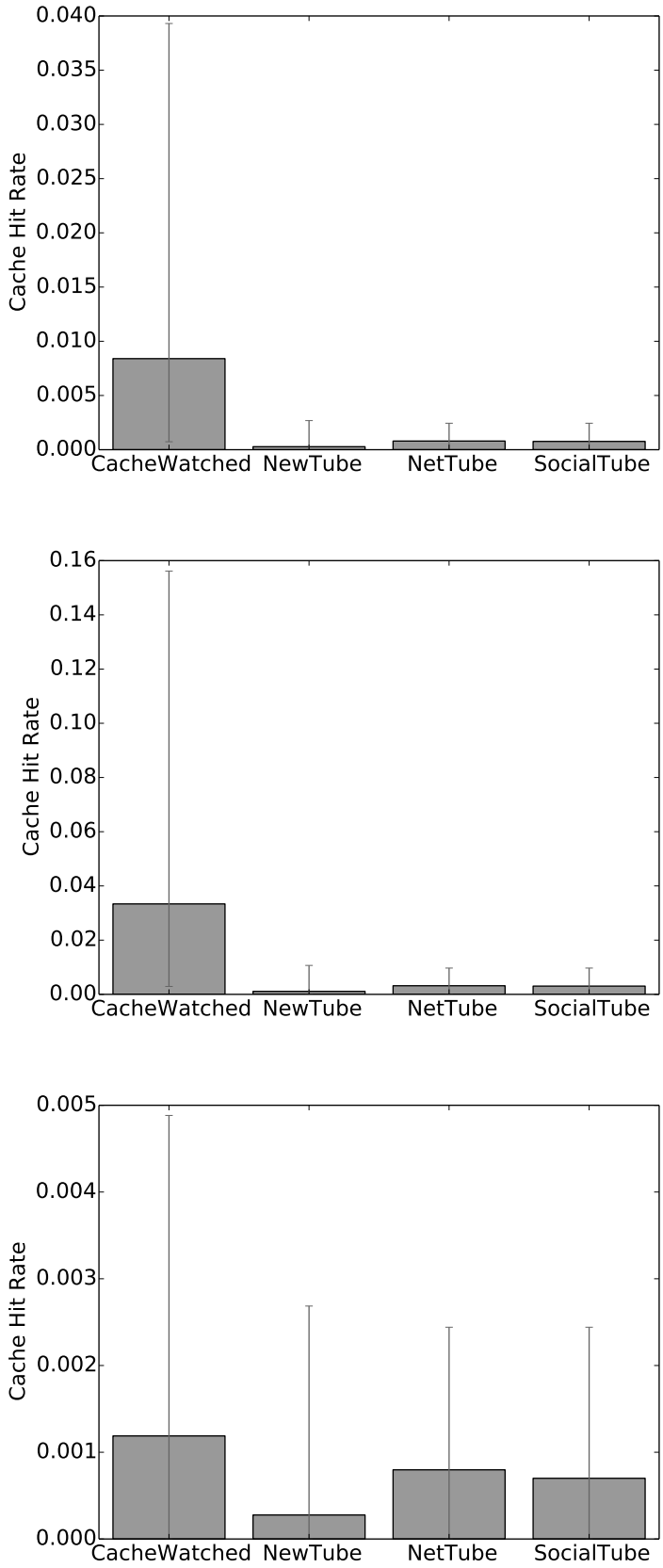

Figure 6.3.: Cache hitrates of the evaluated prefetching strategies a) with unfiltered data b) only including requests to up to the first 10s of videos c) without requests to the same video chunk within 10s

	bare	only 10s prefixes	no re-requests
CacheWatched	1.0	0.70095620151	0.164072242589
NewTube	0.0814814814815	0.0814814814815	0.0814814814815
NetTube	0.0915698575145	0.0915698575145	0.0915698575145
SocialTube	0.410770377125	0.410770377125	0.343602457326

Table 6.2.: Prefetching accuracy of the evaluated prefetching strategies

Observation: cache hitrate on 10s-prefixes is 4 times higher

Moving from bare data to 10 second prefixes, it is observed that the *cache hitrates quadruple*. This gives CacheWatched a hit rate even above 33%.

The good performance of CacheWatched in this scenario relative to the other prefetching strategies is surprising. It was assumed that the strategies that only prefetch 10-second prefixes get a higher increase in their performance, when looking only at such requests.

Observation: most CacheWatched requests are re-requests

When leaving out requests to the same chunk that appear in between 10 seconds, the performance of CacheWatched drops significantly (from 0.0083984375 to 0.00118815104167). This does not increase the performance of other strategies, but allows to conclude that a large part of the performance of CacheWatched is due to the fact that it serves the actual media streaming scheme better.

Serving the media streaming scheme well is of high value in the short term. But in the long term, it is assumed to be better to predict the right videos rather than the right chunks. Media streaming protocols may change, but user preference may not (at least, it can not be measured in hard facts).

6.4.2 Prefetching accuracy

Next up, *prefetching accuracy* is evaluated. The actual data is shown in table 6.2, plots can be seen in figure 6.4. It is observed, that the simple CacheWatched "strategy" performs best, but this comes to a great extent from requests ascribed to the media streaming protocol. Furthermore, one observes that the here-proposed NewTube performs about equal to NetTube.

Observation: Prefetching accuracy as well is highly user-dependent

With all prefetching accuracy measures having a 0% to 100% confidence interval, it is to observe that prefetching accuracy is highly depending on the user. Some users behave "like expected" – then the prefetching accuracy is even at 100% –, some behave completely "unexpected" – then the prefetching accuracy is 0.

This questions the whole result, which is based only on the evaluation of the behavior of 15 users. In average though, a prefetching accuracy of $\approx$ 10% (NewTube, NetTube), $\approx$ 40% (SocialTube) and 100% (CacheWatched, see next paragraph) has been reached.

Observation: CacheWatched is best, again

As with the *cache hitrate* (6.4.1), the *CacheWatched* "prefetching" strategy beats all other strategies by an order of magnitude.

The solid 100% prefetching accuracy on the bare data was expected, as this is the very way CacheWatched works: add everything to the cache that has been requested. Obviously, as this is done for all videos, every video has been in the cache for some time. Therefore, the 100% video hit rate follows naturally.

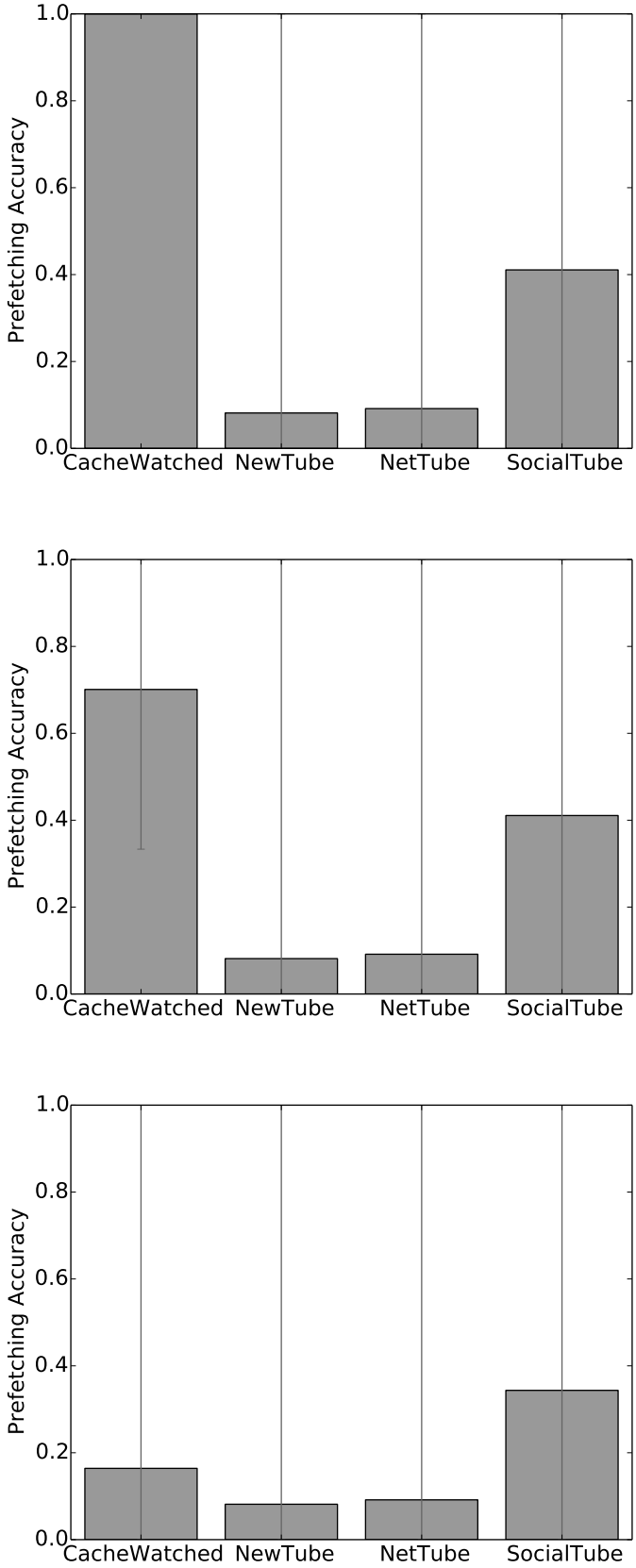

Figure 6.4.: Prefetching accuracy of the evaluated prefetching strategies a) based on all requests b) based on requests to 10s prefixes c) based on requests without re-requests

As somewhat expected, the prefetching accuracy of CacheWatched collapses when examining 10-second prefixes and filtered-out "re-requests". It however stays always above the performance of NetTube and NewTube.

Observation: SocialTube outperforms NetTube and NewTube

The average prefetching accuracy of SocialTube stays consistently around 40%, even when filtering out re-requests. This makes SocialTube the only prefetching strategy that beats CacheWatched in any metric.

It is however to note that the confidence interval is still 0..100%, so this should be taken with a great grain of salt.

Observation: NewTube performs about equal to NetTube

This finally is an encouraging finding: altough the prefetching accuracy of NewTube and NetTube is merely low in all dimensions, the here proposed NewTube performs about equal to NetTube. This fortifies the use of personalized recommendations as a source for prefetching decisions.

6.5 Conclusion

The evaluation of four prefetching strategies for YouTube content on a single mobile device yielded surprising results:

- the *CacheWatched* dummy strategy performed orders of magnitude better than any sophisticated prefetching strategy.
- the here-proposed NewTube performed about equal to NetTube (3.1) in prefetching accuracy.
- SocialTube (3.2) had the best prefetching accuracy when re-requests were filtered out.

With cache hitrates below 1%, the costs of using a sophisticated prefetching strategy vastly exceed their benefits. Overly simplified, to serve n bytes from the prefetching strategy cache, the strategy has to prefetch around $100 \times n$ bytes of content. Given a cache size of $256MiB$ – the size that was used in the simulations –, this would mean $2.56MiB$ of successfully prefetched content, which is around the size of *one single 10s prefix* (5.3.4). The other way around, to stint 200 MB of mobile traffic, one has to prefetch *20 GB* of content – more than the space that is available on the average mobile device.

It is to note that all prefetching strategies were simulated, and actual prefetching could lead to slightly different results. Furthermore, it should be said that caching what has been watched already, especially in this simulation environment, is favored by the adaptive media streaming protocol, as a large amount of requests are in fact "re-requests" that are triggered by the protocol (not the user) in order to adapt to the network bandwidth.

From the observation that NewTube, the prefetching strategy based on personalized recommendations, performed equal to than NetTube in prefetching accuracy, it can be concluded that personalized recommendations are a valid source for prefetching decisions. A lot of fine-tuning needs to be made in order to make this capable of competing with others, but in the general direction, it worked.

However, simply caching the watched video chunks performed way better in every scenario, around 10 times in cache hitrate, around 8 times in prefetching accuracy (considering 10s-prefixes). Therefore, it has to be asked if – as the naïve approach worked best – the best way to prefetch content on mobile devices would be a fine-tuned naïve algorithm.

In the short term, this would as well have the benefit of dealing with the media streaming protocol in the right way. And in the long term, when thinking of changes to the media streaming protocol, a naïve approach could be adapted to the new conditions with less effort.

7 Conclusion

In this chapter, the work presented in this thesis is shortly summarized (7.1). Furthermore, an outlook for further research is given (7.2). It ends with some final words on the topic (7.3).

7.1 Retrospect

The goal of this thesis (1) was to join the research topics of *prefetching strategies*, *recommendation systems* and *mobile devices* to one assortative approach. Questions were wether prefetching strategies, that were merely designed for peer-to-peer networks, work on mobile devices as well, and if personalized recommendations could be used to improve prefetching decisions on those devices.

Background information and related work

To dive into the topic, *background information* (2) and *related work* (3) were discussed. Focus was laid on the theory behind *recommendation systems* (2.2) – which use the general concepts of *association rule mining* (2.2.1) and *collaborative filtering* (2.2.2) – and their application in video sharing services (3.3, 3.4, 3.5). On the other hand, *prefetching strategies* were motivated (2.1), and two well-known approaches were discussed. Those were NetTube (3.1), where 10-second prefixes of three related videos are prefetched when a user watches a YouTube video, and SocialTube (3.2), where a social network is used to identify followers and non-followers, whose video uploads are prefetched in a push manner.

Design and realization

The main part of the work for this thesis was the *design* (4) and *realization* (5) of an *Android application* that allows prefetching of YouTube videos on a single mobile device. In the design phase, multiple scenarios were considered, and of *showing the YouTube website in a WebView* (4.3), *developing a YouTube-clone* (4.4), and *tape-recording traffic between the YouTube application and the YouTube servers* (4.2), it was decided to develop an application based on a transparent proxy, that reads all traffic using a man-in-the-middle impersonation attack on https content. This was realized using the library *SandroProxy* (5.2), simulating *NetTube* (5.3.2) and *SocialTube* (5.3.3) as well *NewTube* (5.3.4), the recommendation-based strategy proposed in this thesis, and *CacheWatched* (5.3.1), a simple cache-what-has-been-watched dummy.

Evaluation

Finally, an evaluation was run with 15 users, in that $1.63GB$ of YouTube content were captured over 19 days (6). The evaluation yielded the surprising result that the naïve *CacheWatched* approach performed orders of magnitude better than all sophisticated prefetching strategies (6.4). This led to the conclusion that, on single mobile devices, a simple but fast algorithm works better than a sophisticated one – an observation that has been reported from other fields of research as well. It was also observed, that NewTube did perform about equal to NetTube and SocialTube, which lets conclude that personalized recommendations are in general a good source for prefetching decisions.

7.2 Outlook

As described throughout this thesis, all work is primal research, and lacks tuning, improvement and deeper thoughts. It was mainly created to provide scientific results, but has always been realized with expandability in mind. Therefore, further research can build on some key points:

Naïve caching strategies

The evaluation (6) showed that a naïve approach worked way better than anything else. This should be a starting point for research deepening this fact. Fine-tuning in terms of cache size, cache replacement strategies and cache target selection should be performed (and evaluated).

It should be kept in mind that applications of video sharing services might implement caching and/or prefetching as well, and that that might interfere with a caching algorithm. Therefore, it would ideally be realized directly in a video sharing service application. On the other hand, this would render sophisticated multi-source approaches near impossible.

Sophisticated prefetching strategies

Realizations of NetTube, SocialTube and NewTube performed rather badly in the evaluation (with a bare cache hitrate below 0.8%). To be useful in real life and not stagnate in research, those strategies need to be adapted to mobile devices in a high degree. Such adaptations could include a more fine-grained video selection or selecting less videos with higher confidence and prefetching them in total (not just 10 seconds).

Here it has to be noted that different use cases lead to different prefetching strategy design decisions. NetTube and SocialTube aim for a fast initial load and assume that the rest of the video can be loaded while the prefetched 10s prefix is played. This assumption may not hold for mobile devices, where permanent high-speed connectivity is not available. Therefore, prefetching a video in total – or in a larger part – may become more important.

Adaptive prefetching strategies

As the realized application offers the opportunity to run multiple prefetching strategies *in parallel*, an adaptive prefetching strategy that reacts on the users behavior could be realized.

A first step into this direction may be to simulate all available strategies in parallel and dynamically setting the best-performing strategy as the active one. It may go on with all strategies working in parallel and weighting their decisions to pick the best of both worlds.

Multi-source prefetching strategies

The fact that the realized application can read all network traffic could be leveraged for a whole new level of prefetching: combining multiple sources of information to understand the user and make the best prefetching decisions.

This could be the user browsing Facebook, coming to a video post. A multi-source prefetching strategy could notice that a video is incoming and start prefetching it. By the time the user clicks the video post (and is redirected to the YouTube application to watch it), the video is already (partly) prefetched and he can instantly start watching. Or, this could be an intense tracking of the user's behavior – all on the local device only to provide privacy – that can be used to understand him near-perfect.

Practical realization

It is of the least scientific interest, but a realization of an application that *actually prefetches* and serves videos would be a great improvement. The scientific benefits are more reliable evaluation results, but the user can then actually gain something from using the application.

The base for this practical realization was created in this thesis. Simply put, all that is need to be done is realizing `.prefetching.strategy.YouTubePrefetchingStrategy#cachePrefixes(String)` in a way that actually caches the prefixes and refactoring `.prefetching.strategy.PrefetchingStrategyStub#on(NewChunkEvent)` to return the cached prefix. For the realizations, one can for example borrow from the Squid[1] project.

[1] squid-cache.org, the Squid project, checked on 2015-04-28

7.3 Final words

Prefetching on mobile devices is a relatively new research topic. It stimulates the dream of instant video access even in areas with bad connectivity. This thesis shows that it is a long way to go until this dream is reality, but that there is for sure possibility to make it.

Personalized recommendations are a valid source for prefetching decisions. A combination of multiple prefetching strategies or even the use of multiple video sharing services for prefetching decisions can drastically improve prefetching accuracy. Until then, a naïve approach works wonders.

The Android application realized in this thesis lays a good ground to start further research on.

A Tables

video	bytes video	bytes audio
youtu.be/LBqmFHXYyOQ	1287326	149249
youtu.be/vp_9kte83pE	539178	152565
youtu.be/LGcG96vrxPo	2524897	152934
youtu.be/bGIBHmzjVnk	2773159	160785
youtu.be/ahu_nRO-JbI	2486610	160785
youtu.be/wfVHzMcEzFM	2221707	160785
youtu.be/cV9dxx2Ohnw	2286611	160785
youtu.be/yCYnz3gsp6M	2295772	160785
youtu.be/kbMvUehm0tE	2454697	160785
youtu.be/U1qA9fLQT6k	2761764	160785
youtu.be/1H7ZH85g-fU	2650766	160785
youtu.be/zyhf6lOJv5s	2728257	160673
youtu.be/kVxBWLwIU_o	2753885	160785
youtu.be/-xf3u_KebaE	2756241	160785
youtu.be/RveKvEZPiRw	2332153	160785
youtu.be/8eAYek78h0s	2498089	160785
youtu.be/RpkCUySdDFU	2330618	160785
youtu.be/U6OZZDyQAEA	1898046	160785
youtu.be/oZtyuyiDuPA	2187053	160785
youtu.be/v3ZIBD7-02Y	1668106	161153
youtu.be/2RVdgjMJiFI	1675162	161153
youtu.be/C5B16PaSF24	1338149	161153
youtu.be/TvrB4OdvDpw	1567919	161153
youtu.be/EJPfefq-JS8	1688429	161153
youtu.be/OUvlamJN3nM	34370	163438
youtu.be/8zqSSJSQlWo	492440	160208
youtu.be/cc2ViC1p6Tc	430403	175551
youtu.be/16-V0XnYEZQ	95088	164839
youtu.be/3RrkItljObk	459972	173683
youtu.be/rA0GAYpFlgg	1494770	149249
median	2204380	160785
average	1823721	160664

Table A.1.: bytes at 10 seconds for 30 videos, collected on 2015-03-02

B Bibliography

[AB12a] Xavier Amatriain and Justin Basilico. Netflix Recommendations: Beyond the 5 stars (Part 1). http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html, checked on 2014-11-12, April 2012.

[AB12b] Xavier Amatriain and Justin Basilico. Netflix Recommendations: Beyond the 5 stars (Part 2). http://techblog.netflix.com/2012/06/netflix-recommendations-beyond-5-stars.html, checked on 2014-11-12, June 2012.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216. ACM, 1993.

[All15] Open Handset Alliance. Code Style Guidelines for Contributors. http://source.android.com/source/code-style.html, checked on 2015-03-31, 2015.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, volume 1215, pages 487–499, 1994.

[Bar11] Adam Barth. RFC 6454: The Web Origin Concept. http://tools.ietf.org/html/rfc6454, checked on 2014-11-06, December 2011.

[Bar14] Alex Barredo. A Comprehensive Look at Smartphone Screen Size Statistics and Trends. https://medium.com/@somospostpc/e61d77001ebe, checked on 2015-04-20, May 2014.

[Bec03] Kent Beck. *Test-Driven Development: by Example*. Addison-Wesley Professional, 2003.

[Bha97] Dileep Bhandarkar. RISC versus CISC: A Tale of Two Chips. *SIGARCH Computer Architecture News*, 25(1):1–12, 1997.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *The Journal of Machine Learning Research*, 3:993–1022, March 2003.

[Boo06] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education, 2006.

[Cis14] Cisco White Paper. Cisco Visual Networking Index: Forecast and Methodology, 2013–2018. Technical report, Cisco, 2014.

[Cis15] Cisco White Paper. Cisco Visual Networking Index: Forecast and Methodology, 2014–2019. Technical report, Cisco, February 2015.

[CL09] Xu Cheng and Jiangchuan Liu. NetTube: Exploring Social Networks for Peer-to-Peer Short Video Sharing. In *INFOCOM 2009, IEEE*, pages 1152–1160, April 2009.

[Cro06] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF, July 2006.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM, 2010.

[DDM+95] Asit Dan, Daniel M. Dias, Rajat Mukherjee, Dinkar Sitaram, and Renu Tewari. Buffering and Caching in Large-Scale Video Servers. In *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers.*, pages 217–224, March 1995.

[DLL+10] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The YouTube Video Recommendation System. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 293–296. ACM, 2010.

[DPN+14] Wei Deng, Rajvardhan Patil, Lotfollah Najjar, Yong Shi, and Zhengxin Chen. Incorporating Community Detection and Clustering Techniques into Collaborative Filtering Model. *Procedia Computer Science*, 31(0):66–74, May 2014.

[EOM+09] William Enck, Machigar Ongtang, Patrick Drew McDaniel, et al. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[ERK11] Michael D. Ekstrand, John T. Riedl, and Joseph A. Konstan. Collaborative Filtering Recommender Systems. *Foundations and Trends Human-Computer Interactions*, 4(2):81–173, February 2011.

[Fat13] Richard Fateman. Dumb = Fast for Polynomial Multiplication. Or, You have Plenty of Memory. Use Some. `https://www.cs.berkeley.edu/~fateman/papers/dumbisfast.pdf`, checked on 2015-04-23, January 2013.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FK10] Julian Frumar and Igor Kofman. The Video Page Gets a Makeover. `http://youtube-global.blogspot.de/2010/01/video-page-gets-makeover.html`, checked on 2014-11-25, January 2010.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection Pattern. `https://blog.itu.dk/MMAD-F2013/files/2013/02/3-inversion-of-control-containers-and-the-dependency-injection-pattern.pdf`, checked on 2015-03-29, January 2004.

[FRS01] Karl Pearson FRS. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[GJS+13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, February 2013.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, February 2015.

[Goo11] Google Chrome Developers. New Chromium Security Features, June 2011. `https://blog.chromium.org/2011/06/new-chromium-security-features-june.html`, checked on 2015-04-13, June 2011.

[GZS+14] Xue Geng, Hanwang Zhang, Zheng Song, Yang Yang, Huanbo Luan, and Tat-Seng Chua. One of a Kind: User Profiling by Social Curation. In *Proceedings of the ACM International Conference on Multimedia*, MM '14, pages 567–576. ACM, 2014.

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, October 2012.

[HLL03] Jin Huang, Jingjing Lu, and Charles X Ling. Comparing Naive Bayes, Decision Trees, and SVM with AUC and Accuracy. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 553–556. IEEE, 2003.

[HLR07] Cheng Huang, Jin Li, and Keith W. Ross. Can Internet Video-on-demand Be Profitable? In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 133–144. ACM, 2007.

[HRM04] T. Halonen, J. Romero, and J. Melero. *GSM, GPRS and EDGE Performance: Evolution Towards 3G/UMTS*. Wiley, 2004.

[Huf08] Scott Huffman. Search Evaluation at Google. `http://googleblog.blogspot.de/2008/09/search-evaluation-at-google.html`, checked on 2014-11-13, September 2008.

[Hug89] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[IDC14] IDC. Worldwide Smartphone Shipments Edge Past 300 Million Units in the Second Quarter; Android and iOS Devices Account for 96% of the Global Market, According to IDC. `http://www.idc.com/getdoc.jsp?containerId=prUS25037214`, checked on 2015-04-13, August 2014.

[JSS12] Hosein Jafarkarimi, Alex Tze Hiang Sim, and Robab Saadatdoost. A Naïve Recommendation Model for Large Databases. *International Journal of Information and Education Technology*, 2(3):216–219, 2012.

[Kob09] Evan Koblentz. How it Started: Mobile Internet Devices of the Previous Millennium. *International Journal of Mobile Human Computer Interaction (IJMHCI)*, 1(4):1–3, 2009.

[Koc11] Stephen G Kochan. *Programming in Objective-C*. Addison-Wesley Professional, 2011.

[Kor08] Yehuda Koren. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 426–434. ACM, 2008.

[KZGZ12] Samamon Khemmarat, Renjie Zhou, Lixin Gao, and Michael Zink. Watching User Generated Videos with Prefetching. *Signal Processing: Image Communication*, 27(4):343 – 359, 2012.

[LHM99] Bing Liu, Wynne Hsu, and Yiming Ma. Mining Association Rules with Multiple Minimum Supports. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 337–341. ACM, 1999.

[LL14] Adam Lella and Andrew Lipsman. The U.S. Mobile App Report. `http://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report`, checked on 2015-04-20, August 2014.

[LSASW11] Salvatore Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. *Internet Engineering Task Force, Request for Comments*, 6202(2070-1721):32, 2011.

[LSW+12] Ze Li, Haiying Shen, Hailang Wang, Guoxin Liu, and Jin Li. SocialTube: P2P-assisted Video Sharing in Online Social Networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 2886–2890, March 2012.

[Mar08] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

[ML13] Nima Mirbakhsh and Charles X. Ling. Clustering-based Factorized Collaborative Filtering. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 315–318. ACM, 2013.

[MN14] John Mattsson and Mats Näslund. Detection and Mitigation of HTTPS Man–in–the–Middles and Impersonators. In *Web Cryptography Next Steps*. W3C, September 2014.

[MS12] Stan Matwin and Vera Sazonova. Direct Comparison between Support Vector Machine and Multinomial Naive Bayes Algorithms for Medical Abstract Classification. *Journal of the American Medical Informatics Association*, 19(5):917, June 2012.

[Mul12] Brian Mulloy. Web API Design: Crafting Interfaces that Developers Love. `https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf`, checked on 2015-04-16, 2012.

[MV14] Uwe Maurer and Mathijs Vogelzang. Top Android SDK versions. `http://www.appbrain.com/stats/top-android-sdk-versions`, checked on 2014-11-06, November 2014.

[Pat07] Arkadiusz Paterek. Improving Regularized Singular Value Decomposition for Collaborative Filtering. In *Proceedings of KDD Cup and Workshop*, volume 2007, pages 5–8, 2007.

[PB03] Stefan Podlipnig and Laszlo Böszörmenyi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[Raj09] Shiva Rajaraman. Five Stars Dominate Ratings. `http://youtube-global.blogspot.de/2009/09/five-stars-dominate-ratings.html`, checked on 2014-11-25, September 2009.

[Res00] Eric Rescorla. RFC 2818: HTTP Over TLS. `http://tools.ietf.org/html/rfc2818`, checked on 2014-11-06, May 2000.

[Sch97] R.R. Schaller. Moore's Law: Past, Present and Future. *Spectrum, IEEE*, 34(6):52–59, June 1997.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the 24th International Conference on Machine Learning*, pages 791–798. ACM, 2007.

[Sod11] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, (18):62–67, 2011.

[Spo12] YouTube Spotlight. Holy Nyans! 60 hours per minute and 4 billion views a day on YouTube. `http://youtube-global.blogspot.de/2012/01/holy-nyans-60-hours-per-minute-and-4.html`, checked on 2015-04-17, January 2012.

[SS95] Rafael H. Saavedra and Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *Computers, IEEE Transactions on*, 44(10):1223–1235, October 1995.

[SVBW11] Matthew J Stollak, Amy Vandenberg, Andie Burklund, and Stephanie Weiss. Getting Social: The Impact of Social Networking Usage on Grades Among College Students. In *Proceedings from ASBBS annual conference*, pages 859–865, 2011.

[THI+10] Juan M. Tirado, Daniel Higuero, Florin Isaila, Jesús Carretero, and Adriana Iamnitchi. Affinity P2P: A Self-Organizing Content-Based Locality-Aware Collaborative Peer-To-Peer Network. *Computer Networks*, 54(12):2056 – 2070, 2010.

[vK14] Anne van Kesteren. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, checked on 2014-11-06, January 2014.

[Wan99] Jia Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Computer Communication Review*, 29(5):36–46, October 1999.

[Zak00] Mohammed J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.