EXAMENSARBETE INOM INFORMATIONSTEKNIK,
AVANCERAD NIVÅ, 30 HP

STOCKHOLM, *SVERIGE 2016*

Performance of frameworks for declarative data fetching

An evaluation of Falcor and Relay+GraphQL

MATTIAS CEDERLUND

KTH
SKOLAN FÖR INFORMATIONS- OCH KOMMUNIKATIONSTEKNIK

Performance of frameworks for declarative data fetching

An evaluation of Falcor and Relay+GraphQL

MATTIAS CEDERLUND

2016-07-04

Master's Thesis
Examiner: Fredrik Kilander
Supervisors: Fredrik Kilander and Patric Dahlqvist

TRITA

Abstract

With the rise of mobile devices claiming a greater and greater portion of internet traffic, optimizing performance of data fetching becomes more important. A common technique of communicating between subsystems of online applications is through web services using the REpresentational State Transfer (REST) architectural style. However, REST is imposing restrictions in flexibility when creating APIs that are potentially introducing suboptimal performance and implementation difficulties.

One proposed solution for increasing efficiency in data fetching is through the use of frameworks for declarative data fetching. During 2015 two open source frameworks for declarative data fetching, Falcor and Relay+GraphQL, were released. Because of their recency, no information of how they impact performance could be found.

Using the experimental approach, the frameworks were evaluated in terms of latency, data volume and number of requests using test cases based on a real world news application. The test cases were designed to test single requests, parallel and sequential data flows. Also the filtering abilities of the frameworks were tested.

The results showed that Falcor introduced an increase in response time for all test cases and an increased transfer size for all test cases but one, a case where the data was filtered extensively. The results for Relay+GraphQL showed a decrease in response time for parallel and sequential data flows, but an increase for data fetching corresponding to a single REST API access. The results for transfer size were also inconclusive, but the majority showed an increase. Only when extensive data filtering was applied the transfer size could be decreased. Both frameworks could reduce the number of requests to a single request independent of how many requests the corresponding REST API needed.

These results led to a conclusion that whenever it is possible, best performance can be achieved by creating custom REST endpoints. However, if this is not feasible or there are other implementation benefits and the alternative is to resort to a "one-size-fits-all" API, Relay+GraphQL can be used to reduce response times for parallel and sequential data flows but not for single request-response interactions. Data transfer size can only be reduced if filtering offered by the frameworks can reduce the response size more than the increased request size introduced by the frameworks.

Keywords

Web services, Declarative data fetching, Frameworks, Representational state transfer, Falcor, Relay, GraphQL

Sammanfattning

Alteftersom användningen av mobila enheter ökar och står för en allt större andel av trafiken på internet blir det viktigare att optimera prestandan vid data-hämtning. En vanlig teknologi för kommunikation mellan delar internet-applikationer är webbtjänster användande REpresentational State Transfer (REST)-arkitekturen. Dock introducerar REST restriktioner som minskar flexibiliteten i hur API:er bör konstrueras, vilka kan leda till försämrad prestanda och implementations-svårigheter.

En möjlig lösning för ökad effektivitet vid data-hämtning är användningen av ramverk som implementerar deklarativ data-hämtning. Under 2015 släpptes två sådana ramverk med öppen källkod, Falcor och Relay+GraphQL. Eftersom de nyligen introducerades kunde ingen information om dess prestanda hittas.

Med hjälp av den experimentella metoden utvärderades ramverken beträffande svarstider, datavolym och antalet anrop mellan klient och server. Testerna utformades utifrån en verklig nyhets-applikation med fokus på att skapa testfall för enstaka anrop och anrop utförda både parallellt och sekventiellt. Även ramverkens förmåga att filtrera svarens data-fält testades.

Vid användning av Falcor visade resultaten på en ökad svarstid i alla testfall och en ökad datavolym för alla testfall utom ett. I testfallet som utgjorde undantaget utfördes en mycket omfattande filtrering av datafälten. Resultaten för Relay+GraphQL visade på minskad svarstid vid parallella och sekventiella anrop, medan ökade svarstider observerades för hämtningar som motsvarades av ett enda anrop till REST API:et. Även resultaten gällande datavolym var tvetydiga, men majoriteten visade på en ökning. Endast vid en mer omfattande filtrering av datafälten kunde datavolymen minskas. Antalet anrop kunde med hjälp av båda ramverken minskas till ett enda oavsett hur många som krävdes vid användning av motsvarande REST API.

Dessa resultat ledde till slutsatsen att när det är möjligt att skräddarsy REST API:er kommer det att ge den bästa prestandan. När det inte är möjligt eller det finns andra implementations-fördelar och alternativet är att använda ett icke optimerat REST API kan användande av Relay+GraphQL minska svarstiden för parallella och sekventiella anrop. Däremot leder det i regel inte till någon förbättring för enstaka interaktioner. Den totala datavolymen kan endast minskas om filtreringen tar bort mer data från svaret än vad som introduceras genom den ökade anrops-storleken som användningen av ett frågespråk innebär.

Nyckelord

Webbtjänster, Deklarativ data-hämtning, Ramverk, Representational state transfer, Falcor, Relay, GraphQL

Contents

Contents

Chapter 1

Introduction

1.1 Background

With the rise of mobile devices that are claiming a greater and greater portion of online traffic, suboptimal performance when fetching data becomes more visible. This is largely due to longer network round trip times and a significantly greater overhead cost in mobile networks compared to wired networks [1, p. 145]. Because of the increase in mobile users, optimizing performance for mobile platforms becomes important when providing online services.

A commonly used technique of communicating between sub-systems of online applications is through web services. In particular, the REpresentational State Transfer (REST) architectural style is often used, providing constraints aiming to minimize latency and load on the network while increasing independence and scalability of the services [2]. However, the strict constraints of REST impose restrictions in flexibility when creating Application Programming Interfaces (APIs), which in turn may introduce suboptimal performance and implementation difficulties. This have resulted in pragmatic implementations of REST, not complying with all constraints defined in the original REST specification [3, p. 62].

Using the RESTful architectural style for web services is light-weight in comparison to other web service protocols like Simple Object Access Protocol (SOAP), both in terms of message size and response times [4]. However, when implementing REST APIs there are tradeoffs to be made which affects the performance of data fetching.

Often REST APIs are implemented in a granular way, consisting of multiple resources. It is possible and likely that multiple resources are needed to build a single page in a client application [3, p. 65]. On the contrary, another pattern for building REST APIs is the "one-size-fits-all" tactic, potentially resulting in over-fetching. If multiple client applications are using the same API, there is a great chance that the

API is optimized for none of them, which will result in unnecessary data transfer and suboptimal performance [5]. A third design pattern for REST APIs is to create custom endpoints, returning data tailored for specific applications or functionalities. However, this will result in an increased number of resources needed, which will only grow over time as new clients and variants are introduced [6].

Multiple solutions for increasing efficiency in data fetching have been proposed, of which some are concerning data formats while others are trying to optimize the number of network requests. A recent trend involves frameworks for declarative data fetching, where the client applications specify what data it needs on a per field basis rather than fetching everything from a specific location defined by a URL. The frameworks will then optimize the communication with the servers in order to fetch the data efficiently.

1.1.1 Schibsted

Schibsted Media Group is an international media group with many well known digital media products. In Sweden, their products Aftonbladet, Blocket.se and Hitta.se are among the most visited. Combined, Schibsted's products are reaching half the Swedish population every day [7]. This degree project is executed in the context of Schibsted, more specifically in the context of their product Aftonbladet.

During the first 4 weeks of 2016, 63 % of all visits to Aftonbladet were from mobile devices [8].

1.2 Problem description

During 2015, Netflix released Falcor [9] and Facebook released Relay+GraphQL [10, 11] to the public. They are both open source frameworks for declarative data fetching, trying to solve the aforementioned problems in the RESTful architectural style. However, there were little information available of how these frameworks affects performance.

With a lot of different client applications on multiple platforms, of which some are mobile, there is a need for a flexible yet high performant service for data fetching at Schibsted. This degree project evaluates how the choice of technology for data fetching will affect performance in client applications and answer the following questions:

- Which data fetching method, Falcor, Relay+GraphQL or REST APIs, will provide the fastest data fetching in terms of latency?

- Which data fetching method will cause the least amount of traffic over the network? Which will need the least number of requests and send the least amount of bytes?

1.3 Purpose

The purpose of this degree project was to study how the choice of technology for data fetching will affect performance in comparison to each other and answer the questions presented in section 1.2. The study was conducted as a case study in the context of Schibsted, to provide insights from real world scenarios.

As no previously published research mentioning either Falcor or Relay+GraphQL could be found, the purpose of this project was also to lessen that knowledge gap and create credible insights about the performance of the frameworks.

1.4 Goals

The goal of this degree project was to evaluate methods for efficient data fetching in terms of performance. This would provide insights to Schibsted and other tech companies about the choice of data fetching methods. Subgoals of the degree project involved:

- Identify performance metrics for data fetching.
- Evaluate how different choices of data fetching methods affect the performance metrics identified in the initial analysis.
- Provide empirical data on the performance of data fetching methods, including previously not evaluated frameworks Falcor and Relay+GraphQL.

1.5 Method

The degree project work was divided in three main phases:

- **Literature study**

  In this phase relevant background information of data fetching technologies, Falcor, Relay, GraphQL, and related work were studied. This was needed for the continued work and experimental evaluation.

- **Experiment design**

  In this phase the current implementation of the test application and data fetching options were examined, and measurement techniques were identified.

Hypotheses for changes in latency, data volume and number of requests were stated and experiments to test the hypotheses were designed. Also different scenarios based on the test application were constructed to provide deeper, more generalizable insights.

- **Experiment implementation and execution**

  In this phase, implementations exposing the test data were implemented using REST, Falcor and Relay+GraphQL. The experiments designed to test performance of the data fetching methods were implemented and executed. The results were assembled and analysed, leading to the conclusion.

1.6 Delimitations

This study focused only on the frameworks Falcor and Relay+GraphQL, even though there may exist additional solutions for declarative data fetching. These are hypertext based data fetching methods, which provides good compatibility. Supporting them in both web browsers and mobile applications would be straight forward using built-in HTTP clients if no client library for the respective platforms were available.

1.7 Benefits

Tech companies can benefit from this study, as it provides guidance in making well-founded decisions in their selection of appropriate technologies for data fetching. It benefits Schibsted in particular, as the study evolves around realistic use cases from their operations.

The framework creators and their open source community may benefit from this study as it highlights potential performance problems and implementation difficulties, which could lead to improvements of the frameworks. Proposed improvements for the frameworks were included in section 8.3.

The end-users of services provided using the technologies evaluated in this study may benefit by getting better and higher performant services, because the service providers had more information to base their selection of technologies on.

1.8 Ethics and risks

As of writing, both the Falcor and GraphQL reference implementations were marked as previews by their respective developers. The GitHub descriptions contained warnings about bugs and future changes that may not be backward compatible.

Because none of the frameworks are mature, they are more likely to contain unknown bugs than mature alternative solutions. When selecting a technology for data dissemination, this risk should be taken into consideration. The benefit of potential performance improvements needs to be weighed against the risk of introducing problems caused by faulty software. For example, a bug could cause inaccurate data to be delivered that is misleading the users or cause an interruption in the data delivery. Both these cases could have economic consequences. This is especially important for a news application, as the users expect information that is both accurate and highly available.

The report need not to expose information and data that could harm Schibsted's operations.

1.9 Sustainability

Different data fetching methods need different amount of server capacity, which will affect how much hardware is needed and their power consumption. This would also affect the cost of operating the servers.

The different data fetching methods may transfer various amount of data, both in terms of number of requests and data size. This would affect the need of networking resources, their power consumption and cost of operation.

The social impact of data fetching performance may result in differences in time spent waiting for the online services, affecting productivity for the service consumers.

1.10 Outline

Chapter 1

Introduced this degree project by providing an overview of the problem, goals and methods.

Chapter 2

Introduces relevant background knowledge, the main concepts of Falcor, Relay and GraphQL and related work needed for the execution of this project.

Chapter 3

Provides a detailed description of the research methodology used during this project. The research hypotheses are stated.

Chapter 4

Presents the experiment design, including measurement techniques and an initial discussion about the impact of the possible solutions. The test application, experiments, test cases and data sets are introduced.

Chapter 5

Describes the implementation of the Falcor and Relay+GraphQL services and the test application. The setup used for executing the experiments is also presented.

Chapter 6

Presents the results of the experiments.

Chapter 7

Presents a discussion of the results in relation to the hypotheses. Additional findings and concerns are discussed.

Chapter 8

Concludes this report by summarizing the report and presenting the conclusions. Finally, future work is suggested.

Chapter 2

Theoretical background

This chapter presents an extended background and briefly introduce related theory and technologies used during this degree project.

Firstly, broader concepts related to distributed systems and software architectures are introduced. Secondly, web services are introduced because the data fetching technologies that were evaluated during this project can be categorized as web services and they implement concepts like choreography and orchestration. The chapter continues by exploring serialization, data formats and caching because they affect performance of data fetching. The chapter proceeds with a presentation of the technologies that were evaluated during this project: REST, Falcor, GraphQL and Relay. Finally, a presentation of related research and insights from the open source community of Falcor and Relay+GraphQL concludes the chapter.

2.1 Distributed systems and architectures

Coulouris *et al.* [12, p. 2] describes distributed systems as being characterized by multiple hardware or software components running on devices connected to a network, communicating only by message passing.

The architecture of a software system is defined by the systems components and relationships between the components. Much like architecture of buildings, the goal is to provide a frame of reference for how the system is designed in order to ensure that it will meet demands in terms of, for example, reliability and manageability [12, p. 40].

The client-server architecture is both historically and currently the most employed. In the client-server architecture, the component requesting a resource takes on the role of the client and the requestee adopts the server role. The clients invoke services on the server in order to access shared resources that are managed by the server.

In turn, servers may take on the client role in communication with other servers. For example, web servers are likely to be clients of a file server that is holding the actual web page data persistently [12, p. 46].

An architectural pattern common in software systems is partitioning through layering. In a layered system a particular layer may use services provided by a layer below. This way, higher layers are unaware of the implementation details of lower layers, which may improve maintainability. Middlewares are software layers providing a higher level programming abstraction of underlying services for the developers of distributed systems. For example, a middleware could hide networking by providing an interface for remote procedure calls [12, pp. 17, 51-52, 58].

2.2 Web services

Web services provide servers with an interface that can be used for interaction with client applications. This interface consists of a collection of available operations that can be invoked over the internet. When a resource is invoked, it will typically trigger execution of a program on the server and possibly return the result of the operation. For example, when searching on Google the response is the result of a program execution [12, pp. 381-384].

2.2.1 Choreography and orchestration

Sometimes interaction between client applications and web services are more complex than a single request-response. Multiple requests may need to be performed sequentially, in a specific order, or based on the response of a previous request. Coulouris *et al.* [12, p. 411] provides the following example: When booking a flight in a flight booking system, the availability of tickets and their price needs to be retrieved before the user may request to book a ticket. If a web service itself is doing the interaction with other web services, a protocol governing the interactions is needed; a protocol for *choreographing* the interactions of the system [12, pp. 411-412] [13]. Orchestration on the other hand describes the sequence of operations performed by a single client, rather than the entire business process which involves multiple parties [13].

2.2.2 Mashups

A mashup application is a service created by combining multiple pre-existing web resources available on the internet. The goal of such applications is to create value through discovery of new innovative, creative and useful use cases [12, p. 414]. For example, a map application could be combined with a list of housing properties

for sale to assist people looking for houses based on geography. These applications typically make use of multiple publicly available web services and APIs for retrieving their data [14]. Thus, web service orchestration may be useful to define the data fetching behavior.

2.3 Serialization and data formats

When an object is to be transferred between two sub-systems, it first needs to be converted to a format that is suitable for transmission over the network. This process is called serialization or marshaling and the produce is called the external data representation [12, p. 158].

External data representations can be grouped into two approaches:

- **Binary representation**

  The data is marshaled into a non human readable format. Java Object Serialization [15] and Google's Protocol Buffers [16] are examples of binary representation formats.

- **Textual representation**

  The data marshaled into a human readable and editable format. Two examples of textual representation formats are XML [17] and JSON [18].

The choice of external data representation will affect the performance of data transmission. Generally, the size of the marshaled data will be larger using a textual representation than a binary [12, p. 159], but the unmarshalling process may be quicker [19] depending on what formats are used.

2.4 Caching

A cache is a temporary storage holding copies of resources that was recently used by one or multiple clients. For example, a web browser may save the responses it receives in a local cache. When the same resource is requested again, the browser will first check if it is already available in the cache and if its up-to-date before making a new request to the server. Caching can also be used on the server side, for example through the use of web proxies which are providing a shared cache for resources used by multiple clients [12, pp. 49-50]. This is both increasing the clients perceived performance, and reducing the load on the network and servers.

2.5 REST

REST is a software architectural style first introduced in the PhD dissertation of Fielding [2]. The style is characterized by six constraints [2]:

- **Client-Server**

  Systems using the REST architectural style are using the client-server model, separating the user interface from data storage and other business logic that is located on the server side. This improves portability by allowing creation of platform specific user interfaces and scalability as the server components are kept simple.

- **Stateless**

  The server component is not allowed to keep any stateful session data. Instead, all stateful data must be kept on the client and included in every request from client to server. This constraint improves visibility as only single requests needs to be considered when monitoring the system. Reliability is improved because recovering from partial failures is easier and scalability is improved because the server does not have to manage resources across requests.

- **Cache**

  Data within responses must be labeled as cacheable or non cacheable to give client caches the right to reuse previous responses for future requests. This improves efficiency and scalability as some interactions can be eliminated, and perceived performance will be improved by reducing the average latency of requests.

- **Uniform Interface**

  Interfaces are created uniformly, simplifying the architecture and making interactions more visible. On the other hand, the efficiency of interaction is decreased as information is transferred in a standardized way rather than the most optimal for the specific application. The interfaces are constrained by an additional set of constraints:

  - **Identification of resources**

    All information is abstracted as resources and any nameable piece of information can be a resource; documents, images or abstract services like today's top news. Resources are identified by Uniform Resource Locators (URLs).

  - **Manipulation of resources through representations**

    A representation of a resource is the sequence of bytes describing the resource in a way that can be transferred over the network. Manipulation

of resources is made by sending representations back and forth between client and server together with control data. The control data describes, for example the requested action (GET, POST, or other HTTP verbs) and what media type the representation is using.

  - **Self-descriptive messages**

    Interactions are stateless; all data is available in each request. Standard methods and media types are used; the HTTP verbs describe actions and well known media types are used. Responses explicitly indicate cacheability. This information should be enough for both clients and servers to interpret messages without any additional external information.

  - **Hypermedia as the engine of application state**

    The application state is stored on the client and changes in state is caused by the clients. States are only changed by making HTTP requests and reading the responses, and possible future requests are determined by hypermedia controls in the previous responses. Hypermedia is therefore what is moving the application state forward [20, p. 348].

- **Layered System**

  The REST architecture allows components to be composed in layers. This reduces the system complexity and improves independence of components. Intermediaries may also be introduced to improve scalability, for example through providing load balancing or caching.

- **Code-On-Demand**

  REST architectures allow transfer of client functionality on demand through downloading and executing applets or scripts. This simplifies the clients and improves extensibility as new client functionality can be introduced without a re-deployment. However, this constraint is considered optional.

2.6 Falcor

Falcor is a middleware created by Netflix used to optimize communication between layers within an online application, for example between client applications and the backend servers [21].

2.6.1 The Falcor model and JSON paths

In Falcor, all backend data is modeled as a single JSON resource[1] located on the server, accessible by the clients on demand. Clients request parts of the JSON resource by passing JSON paths to the server. Actually, data in Falcor is traversed in the exact same way as any local JSON object [21].

JSON data can be traversed by a sequence of keys, defined from the root of the JSON object. This sequence is called a JSON path, and marks a location in the JSON data [22]. In figure 2.1, the value "E" can be retrieved by querying the path "a.d.e", and the value "G" can be retrieved with the path "a.f[0].g".

Figure 2.1. A JSON object containing nested objects.

```
1  a: {
2    b: "B",
3    c: "C",
4    d:
5    {
6      e: "E"
7    },
8    f:
9    [
10     {
11       g: "G"
12     }
13   ]
14 }
```

Falcor also accept paths as arrays of keys. In figure 2.1, the value "E" can be retrieved by querying ["a", "d", "e"], and the value "G" can be retrieved with ["a", "f", "0", "g"]. This format is preferred by the Falcor framework, as all paths will be transformed to the array format on the client side before they are sent to the server [22].

The server will respond with only the subset of the JSON document that was requested by the client provided path. If the client requested the path "a.d.e" when the entire JSON object on the server is displayed in figure 2.1, the response would only contain the data displayed in figure 2.2. All unnecessary fields are trimmed, and thus response size is minimized.

2.6.2 JSON Graph

To model graph data as a JSON object, a convention called JSON Graph is introduced. A JSON Graph is like any JSON object, but with additional types to allow

[1] The reader is assumed to have basic knowledge of JSON [18].

Figure 2.2. Response from requesting the path "a.b.e" from the JSON object in figure 2.1.

```
a: {
  d: {
    e: "E"
  }
}
```

representation of graph data [23].

Because JSON objects model tree structures duplicates may be introduced when modeling graphs, causing unnecessary data to be sent over the network. It is also possible that data in a JSON object may become stale if data is duplicated across the graph and then modified. As a modification only affects the specific copy while other copies remains intact with the old, stale data, inconsistent data may be presented in the client application. To combat this, client applications would need to introduce logic to remove the duplicates themselves [23].

To model a graph as a JSON object without introducing duplicates, entities with unique identifiers are kept in a separate collection. The path to an entity within the collection is called the entity's identity path, and should be globally unique within the JSON object. These entities can then be referenced by other entities by using identity paths [23].

A reference is a value type introduced for linking to values in entity lists, much like symbolic links in the Unix filesystem. When a reference is encountered during path evaluation, the path within the reference will be evaluated instead [23]. Figure 2.3 shows a reference to an object located at the path ["article", 1].

Figure 2.3. A reference to an article at the path ["article" , 1].

```
{ $type: "ref", value: ["article", 1] }
```

Figure 2.4 shows a graph containing news data. The list with the key "news" contains two news items, both referring to articles in a separate collection. The article with id "2" has one related article, namely the article with id "1". Note that this JSON object does not contain any duplicate data entries.

2.6.3 Routes

When all data is exposed through a single URL client applications can request all the data they need in a single request, possibly avoiding sequential network round trips. Instead of identifying resources by URLs, resources are identified by

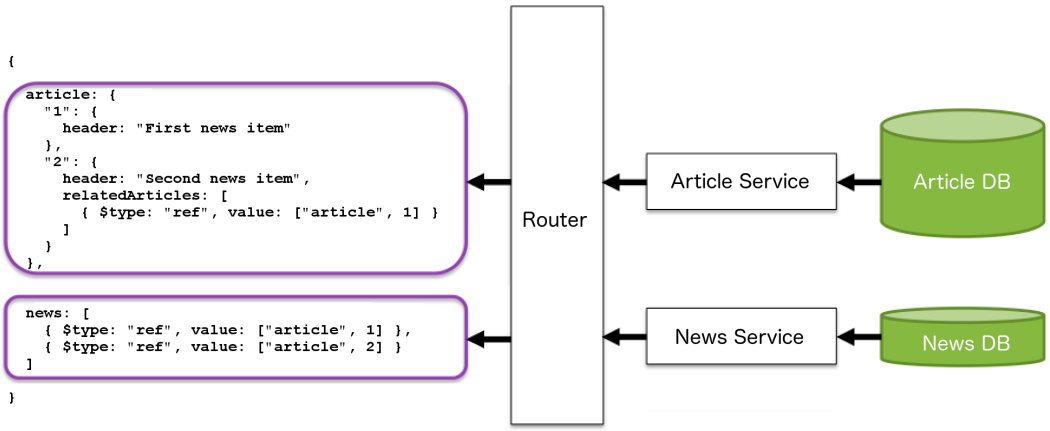
Figure 2.4. A graph containing news articles with related articles.

```
{
  article: {
    "1": {
      header: "Falcor released publicly"
    },
    "2": {
      header: "Improvements in Falcor version 0.1.16",
      relatedArticles: [
        { $type: "ref", value: ["article", 1] }
      ]
    }
  },
  news: [
    { $type: "ref", value: ["article", 1] },
    { $type: "ref", value: ["article", 2] }
  ]
}
```

their JSON paths. To do this matching, Falcor provides a router that is routing incoming request paths to the respective service or data store [21].

Figure 2.5 displays the mapping between the server's JSON object, the router and the respective services and data stores that provide data for building the JSON object. In this example, the article data is provided by the article service while the news list is fetched from the news service. While the client made a request to fetch the news list with headers and related articles, the router is building a JSON graph response using multiple data sources behind the scenes.

Figure 2.5. An example of a Falcor router for serving article data.



2.6.4 Optimizations

Falcor is designed to handle communication with the server transparently. To improve the efficiency of data fetching, it does the following optimizations [24]:

- **Caching**

  Previously fetched values are kept in an in-memory cache. Sub-sequent requests for the same data can be retrieved directly from the cache instead of over the network.

- **Batching**

  It is possible to collect multiple smaller requests into a single batch request. This may improve performance if the network overhead is high.

- **Request deduping**

  If there is an outgoing request that has not yet finished and another request is made with the same path, the second request is omitted and the response from the first request will be reused to answer the second request. This ensures no unnecessary requests are made and will remove the need for coordination among the presentation layer views which are requesting data.

2.7 GraphQL

GraphQL is a query language designed for describing data requirements of client applications, defined in a RFC specification [11] that is still under development. This section will introduce two of the main concepts in GraphQL, the type system and querying.

2.7.1 Type system

The GraphQL type system describes what types of objects can be returned by a GraphQL server and is used to check if queries are valid. This description of the server's capabilities - what types it supports - is called the GraphQL schema [11, Sec. 3]. Figure 2.6 shows a simple type modeling an article with an id and a header, both of the String data type.

Figure 2.6. A simple GraphQL type named Article.

```
1 type Article {
2   id: String
3   header: String
4 }
```

When types share common fields, they can share common interfaces. In figure 2.7, EntertainmentArticle and SportsArticle share the interface Article. This enables an article to have related articles of both types. Observe the exclamation mark after the type of the id field. This indicates that the id field is mandatory and must not return the value null [25].

Figure 2.7. Articles of different types modeled using a common interface. Including nested objects.

```
interface Article {
  id: String!
  header: String
  relatedArticles: [Article]
}

type EntertainmentArticle : Article {
  id: String!
  header: String
  relatedArticles: [Article]
  presenter: String
}

type SportsArticle : Article {
  id: String!
  header: String
  relatedArticles: [Article]
  judge: String
}
```

When defining the GraphQL schema, a root for queries should be defined using a type named Query. Figure 2.8 defines an operation that can be executed named article that is accepting an id and is returning an object of the type Article.

Figure 2.8. The Query type, defining an entry point for queries getting articles.

```
type Query {
  article(id: String!) : Article
}
```

2.7.2 Querying

A GraphQL query is declaring what data should be fetched from a GraphQL server. Figure 2.9 displays a query for fetching the header of an article with id "1" and a potential response is given in figure 2.10. Note that this query is using the same operation defined in figure 2.8.

A key functionality of GraphQL is the ability to nest queries. Figure 2.11 shows a query for fetching two articles with headers and publish dates and the headers

Figure 2.9. A query for an article with id "1".

```
query {
  article(id: "1") {
    header
  }
}
```

Figure 2.10. A possible response from executing the query of figure 2.9.

```
{
  "data": {
    "article": {
      "header" : "Constructing GraphQL queries"
    }
  }
}
```

Figure 2.11. A query for articles with id "1". and "2". Fetching headers and publish date of nested related articles.

```
query {
  article1: article(id: "1") {
    header
    publishDate
  }
  article2: article(id: "2") {
    header
    publishDate
    relatedArticles {
      header
      publishDate
    }
  }
}
```

and publish dates of their related articles. Note "article1" and "article2" before the article query. They are defining aliases to be used when a response is returned; the returned articles will be stored under those keys [26]. (See figure 2.12.)

Queries in GraphQL can be composed out of fragments; reusable query subsections [11, Sec. 2.8]. In figure 2.13 a fragment named "articleFragment" is defined. It is including the fields "header" and "publishDate" and is added to the query by using the dot operator (...). After executing the query the result will be the same as figure 2.12, but with less duplication of fields in the query.

Figure 2.12. A possible response from executing the query of figure 2.11.

```
{
  "article1": {
    "header": "Constructing GraphQL queries",
    "publishDate": "2016-04-01"
  },
  "article2": {
    "header": "Optimizing GraphQL queries",
    "publishDate": "2016-04-02"
    "relatedArticles": [
      {
        "header": "Constructing GraphQL queries",
        "publishDate": "2016-04-01"
      }
    ]
  }
}
```

Figure 2.13. A query for articles with id "1". and "2". Fetching headers and publish date of nested related articles using a fragment.

```
query {
  article1: article(id: "1") {
    ...articleFragment
  }
  article2:  article(id: "2") {
    ...articleFragment
    relatedArticles {
      ...articleFragment
    }
  }
}

fragment articleFragment on Article {
  header
  publishDate
}
```

2.8 Relay

Relay is an open source framework provided by Facebook for building data-driven React[2] applications. It is a client library for consuming GraphQL services that is introducing optimizations for data fetching and providing a consistent interface for declaring data requirements of React components.

2.8.1 Fetching data for views

Just as React splits the user interface into reusable components, so is Relay. Each component declare which data it needs and developers can focus on what should be displayed rather than how and when it should be fetched [28].

Data requirements are defined on containers. A container is a holder for a React component and a GraphQL fragment that is defining the data requirements of the component. The container itself is managing the data fetching and surrounding logic, without interfering with the internal state of the component [29].

Just as React components can be combined to build complex applications, so can GraphQL fragments. Parent containers are responsible for composing the fragment for their children [29]. In parallel with building the view-tree, the components are also building a query-tree out of GraphQL fragments, specifying what data needs to be fetched for the entire page [28]. When reaching the root of the query-tree, the data requirements for all children are aggregated in a single query, which can be sent to the server in a single request.

Relay restricts components so they may only access the data they explicitly asked for. This is called *data masking* and will ensure that there are no unexpected data dependencies among components [28], hopefully reducing bugs.

2.8.2 Caching

Fetching data repeatedly can be speeded up by using a client cache. The Relay documentation [30] provides an example where the user is navigating from a list of items to a detailed view of an item and back to the list. Without caching the list would be re-fetched when the user returns to the list, causing a delay from using the network. With client caching the network round trip from navigating back to the list could be skipped, and the response would be returned immediately.

In data modeled with GraphQL it is common that responses overlap. The response from a request for a list of items could contain the same item as the response from

[2]React [27] is a library for building web user interfaces in a decomposable way, out of reusable components.

requesting a single item. If caching is based simply on the query used for fetching the data, the items in the cache could be different if they were modified between fetches [30]. This is the same problem that were observed when developers at Netflix designed Falcor, as described in section 2.6.2.

To solve the cache consistency problem, the hierarchical responses of GraphQL queries are flattened into a collection of records that are stored in a map from id to record. Each record consists of a map from field names to field values or possibly links to other records. Links are special types that reference entries, defined from the root of the map [30]. Figure 2.14 shows an example where an article with a header and an author named Bob is returned from a GraphQL query.

Figure 2.14. A response from executing a GraphQL query.

```
query: {
  article: {
    id: 1,
    header: "Constructing GraphQL queries",
    author: {
      id: 1,
      name: "Bob"
    }
  }
}
```

When cached, the response is flattened to the representation in 2.15. Note that the author is referenced from the article by a link.

Figure 2.15. A flattened response from executing a GraphQL query.

```
Map {
  article_1: Map {
    id: 1,
    header: "Constructing GraphQL queries",
    author: Link(author_1),
  },
  author_1: Map {
    id: 1,
    name: "Bob"
  }
}
```

When writing to the cache the original response is traversed and flattened records with unique ids are created and inserted into the map. Reading from the cache is done by traversing the query and resolving the fields, where links are traversed. Using this tactic, when results overlap each record is only stored once [30] and duplication is no longer causing stale data in the cache.

2.9 Related work

Zhao *et al.* [31] studied the causes of long delays in web browsing on smartphones. They discovered that the main bottleneck at the time was the computational power of the devices rather than the mobile network. To decrease the loading time and power consumption of web browsing they proposed using a virtual machine based proxy. The mobile device would make requests to the proxy which will make requests to the respective web servers, process the responses and send them back to the mobile device. The proxy would be responsible for executing all consecutive HTTP requests and client side scripts, display the web page on a "virtual" screen and send a screen copy back to the client. When evaluating the delay of using the system to access 20 popular web pages, they achieved 80 % reduced delay and 45 % lower power consumption compared to regular web browsing.

This work was relevant as it investigated how introducing an intermediary for fetching web content affects the performance of browsing in mobile devices. However, the work is different from this degree project because it focuses on web page rendering rather than downloading hypertext data. Also, the authors state that the main focus is decreasing power consumption on mobile devices and reducing delay, while optimizing the amount of network traffic and data transferred is secondary.

Huang *et al.* [32] analysed data transfer size, latency and battery usage of mobile mashup applications invoking multiple web APIs. They observed excessive number of requests, dependencies between requests introducing the need of sequential execution to get the required data, and unnecessary data in the responses. To address the problems that were decreasing performance, they proposed a proxy system that acts as an intermediary between the mobile client application and the web APIs. The mobile application would make requests to the proxy with instructions on what data to fetch specified in a query language, API Query Language (AQL). Multiple AQL instructions are contained in a single request to decrease the number of requests. The proxy would make the requests to the respective web APIs and remove unnecessary data in the responses based on the AQL instructions before returning the aggregated data to the client. In their evaluation they verified that data transfer size, latency and energy usage could be reduced. The data transfer size could be reduced significantly, up to 88 % in their experiments. Latency was reduced by 9 % when invoking a single endpoint, but up to at least 30 % when endpoints needed sequential invocations. The latency reduction remained low as the majority of the time was used for invoking the web APIs, while a smaller portion was used for the wireless transmission between the mobile client and the proxy.

This work was relevant to this degree project as it investigated aggregation of web API requests and responses in an intermediary with the goals of reducing latency, data transfer sizes and energy consumption. The proposed system show great similarities with the goals of Falcor and Relay+GraphQL, with the difference that the focus is on aggregating web API requests to third parties. This work focused on

data fetching within the same domain, therefore the time used invoking services may be less significant.

Several non-academic sources describing experiences and discussing the value propositions of using Falcor and Relay+GraphQL were available online. Susiripala [33] described their initial impressions of Relay+GraphQL, proposed improvements to mutations and highlighted the lack of publish/subscribe support. They also discussed some of the similarities and differences between Relay+GraphQL and Meteor [34]. Faassen [35] discussed how GraphQL could be modeled using REST endpoints. Jones [36] described problems in increased code complexity introduced by using Relay+GraphQL in their React applications. Chenkie [37] pointed out the increased need for boilerplate code when using both Falcor and Relay+GraphQL. Corcos [38] provided a great description of the value proposition of Falcor and Relay+GraphQL through a chat application example. Crawford [39] highlighted the hard requirement of using Relay, the need of a GraphQL schema and GraphQL server. It is not possible to use Relay for communication with legacy REST services.

Most of these sources concerned implementation details and simplicity, but not none of them discussed performance. Performance was the main focus of this work.

Chapter 3

Method

This chapter introduces the research methods used in this degree project and describes the process for the execution of the study. First, research methodology is introduced and the selection of methods for this degree project is described and discussed. Then the hypotheses are stated.

As described in section 1.5, this project is divided in three phases: literature study, experiment design and experiment implementation and execution. (See figure 3.1.) These phases are described further in the upcoming sections 3.1, 4 and 5.

Figure 3.1. Overview of the research phases.

3.1 Literature study

In quantitative research, researchers are discouraged to rush into beginning the empirical part of the study. A thorough literature study exploring previous research should be conducted beforehand. The literature study serves several important functions for the researchers. It gives means for the researchers to explore the frontiers of their respective fields and find out what has already been studied, avoiding replication. It places the proposed research in perspective with previous research, identifying how the study may contribute in a meaningful way and extend knowledge. It also helps the researchers in adjusting the research questions so they are not too wide or too vague, and provides knowledge that is useful when interpreting the results of the study and their significance [40, pp. 62-63].

This degree project was initiated with a literature search concluding that no previously published research evaluating technologies for declarative data fetching using the frameworks Falcor and Relay+GraphQL could be found.

Searches were initiated using the search query "Falcor", which resulted in hundreds of results in Google Scholar [41]. However, all top results were unrelated and therefore the search query was considered too broad. The search proceeded by iterating the search query to "Falcor AND Netflix" because the framework and its creator is likely to be mentioned together. This returned only a few results of which all were unrelated. Finally, "Falcor AND JavaScript" was used as a search query because Falcor is written in JavaScript and thus the two terms are likely to appear together. However, the results only included the term JavaScript because of crawling errors. When only including papers after 2014, a year before official release of Falcor, the number of results were adequately low to examine but they were all unrelated studies, mostly in the social and physiological fields.

A similar process was used when searching for previous work on Relay+GraphQL. The search term "Relay" was considered too generic, but as GraphQL is a requirement for using Relay they should appear together. However, the search for "Relay AND GraphQL" did not yield any related results on Google Scholar. When searching for "GraphQL" alone, it appears to be the name of yet another graph query language, other than the one studied in this project. Even when expanding the search query to "GraphQL AND Facebook", only unrelated graph research was found.

The searches were repeated in KTHB Primo [42], IEEE Xplore [43] and ScienceDirect [44] with similar results. The recency of the frameworks is likely contributing to the lack of resources found.

The literature study continued with researching important concepts of data fetching and the parameters that are affecting its performance. REST, a technology for data fetching that is more well established was studied, as it was an important comparison technology in the study. Knowledge of how Falcor and Relay+GraphQL works was acquired, along with implementation details. This was important for the experiment design, implementation and execution phases of the project as it provided the researcher with essential background knowledge. Finally, related work that were presenting and evaluating similar technologies were searched for and insights from non-academic sources were studied. The material was mainly collected from searching Google Scholar, KTHB Primo, article databases like IEEE Xplore and ScienceDirect, and the documentation of the respective frameworks.

3.2 Research method selection

Håkansson [45] provides a model guiding the selection of research methods. To reach the goals and results of a degree project, a strategy for conducting the research is required and needs to be implemented during the course of the project. To guide the selection of research methodology a layered model is introduced, showing what methodologies are compatible with each other. During selection of research methods, each layer in the model should be investigated before continuing to the next one. The layers and the selection of methodologies for this project follows:

- **Quantitative and qualitative research methods**

  The two main categories for research methods are quantitative and qualitative research methods, distinguishing if the research concerns numerical or non-numerical data. The research questions of this project concerns performance which typically is answered using quantitative data, and therefore quantitative research methods were selected. From this point in the research method selection, only the quantitative elements of the model were considered.

  The *quantitative* approach supports experiments and tests to verify or falsify theories and hypotheses. Hypotheses must be measurable with quantifiable data that typically require large data sets and use statistics to provide valid results [45].

- **Philosophical assumptions**

  Philosophical assumptions steers the research by providing assumptions about valid and appropriate research methods [45]. Because the research questions of this project required experiments to be answered and concerned performance of computer science the positivist assumption was used.

  Positivism assumes that the reality is objective and independent of the observer and their instruments. Researchers test theories to improve understanding of phenomenons. The positivist assumption it is usually used in projects of experimental character and is suitable for testing performance within computer systems [45].

- **Research methods**

  Research methods provide a framework for executing research tasks. They specify how research is initiated, carried out and completed [45]. The experimental method was chosen for this project because the goal of the project was to study how the choice of technologies for data fetching affected performance. The choice of technology was seen as the independent variable that was manipulated and performance was the dependent variable that was studied.

  The *experimental* research method is used to study causes and effects, relationships between variables and how manipulating variables affects the result.

This method is often used when investigating performance of software systems [45].

Other alternative research methods included descriptive, fundamental and applied research. However, the descriptive method aims at studying phenomenons, but not their causes. To follow up the results and discuss their causes could be important for the validity of the study because the experiment implementation is open for the researchers interpretation of what is an optimal implementation. The fundamental and applied methods are aimed at generating new theories or technologies and because this was not the goal of the project, they were not relevant to apply.

- **Research approaches**

  Research approaches are used for deciding what is true or false. The most common approaches are the inductive, deductive and abductive approaches [45]. The deductive approach was used as it fits well with the experimental method, where theories and hypotheses are experimentally evaluated to reach conclusions about how changing variables effects the result.

  The *deductive* approach tests theories in order to verify or falsify hypotheses, most of the time using quantitative methods and large data sets. The hypotheses must be measurable and expressing the expected outcome, a generalization based on the collected data [45].

  The *inductive* and *abductive* approaches could have been used when analyzing the results of the experiments, trying to come up with theories of why the systems performed the way they did. However, this was not the main objective of the project.

- **Research strategy / design**

  Research strategies or methodologies are guidelines for how to conduct research including organization, planning and design [45]. The experimental research strategy was used in combination with the case study in this project. This combination was selected because it laid a foundation for providing generalizable experimental data, while at the same time providing data that is representative for real world applications. The reader is asked not to confuse the use of "case study" in this report with the qualitative counterpart.

  Using the *Experimental* research strategy, all factors that affect the results of experiments are controlled. Just as the experimental research method, the strategy verifies or falsifies hypotheses and gives insight in relationships between variables. The amount of data collected is often large and analysed using statistics [45].

  A *Case study* is investigating a phenomenon in real world scenarios, where the boundary between the phenomenon and the context is not clear. This

strategy can be used in both quantitative and qualitative studies [45].

- **Data collection**

  Data collection methods are used when collecting data during research. Some common data collection methods include experiments, questionnaires, case studies and observations [45]. During this project, a set of experiments were designed and used for collecting performance data from the data fetching technologies in order to verify or falsify the hypotheses. The experiments are described further in chapter 4.

- **Data analysis**

  To support decision making based on the collected data, methods for inspecting, cleaning, transforming and modeling data are used. In quantitative research, statistics and computational mathematics are typically used. *Statistics* is used when calculating results for a sample and the significance of the results, while *computational mathematics* can be used for numerical methods, modeling and simulation [45]. Statistical methods were used for data analysis during this project, both to visualize how reliable the measurements were and to determine whether the results were statistically significant.

- **Quality assurance**

  To verify the quality of quantitative research work, validity, reliability, replicability and ethics should be discussed. Validity concerns if the experiments are measuring the correct thing, reliability refers to consistency in the results, replicability is the possibility of another researcher repeating the work and reaching the same results and ethics are concerning moral principles for conducting research [45]. These subjects were considered in every step of the experiment design, implementation and execution and are discussed throughout the report where appropriate.

3.3 The Experimental method

Because the experimental method was chosen as the primary research method for the project it was necessary to explore it further. Experimental research can be summarized with three main characteristics [40, p. 266]:

- An independent variable is manipulated.
- All other variables that affect the dependent variable are kept constant.
- The effect of manipulating the independent variable is observed.

3.3.1 Control of variables

The essence of the experimental method is the control of variables. The researcher must eliminate other possible explanations by controlling the influence of irrelevant variables. Only then the research can make conclusions about causality between the independent and dependent variables. The experimental method rests on two main assumptions of variables [40, p. 267]:

- **The law of the single independent variable**

  If two situations are equal in every aspect except for a single variable that is added or removed in one of the situations, the difference in outcome can be attributed to that variable.

- **The law of the single significant variable**

  If two situations are not equal but it can be shown that no variable other than the independent variable affects the outcome or if other significant variables are kept equal, any difference between the situations when introducing or removing an independent variable can be attributed to that variable.

For this project, before designing experiments to verify or falsify the hypotheses, the first step taken was to identify the independent and dependent variables.

- **The independent variable** that was manipulated during the experiments was the choice of technology for data fetching.
- **The dependent variables** are latency and data volume in regard to transfer size and number of requests. From the transfer size, other insights about the data usage could be derived, like HTTP overhead.

3.3.2 Extraneous variables

Variables that are not among the independent variables of the experiment but will affect the dependent variables are called *extraneous variables*. To draw conclusions of relationships between independent and dependent variables, the possibility of extraneous variables must be eliminated. This challenge is referred to as *confounding*, the mixing of extraneous variables and independent variables so that their effects on the dependent variable can not be separated. Eliminating confounding by controlling the effect of extraneous variables lets the researcher rule out possible explanations of the observed changes [40, p. 268].

The goal when creating the experiments was to make the setups as equal to each other as possible so **the law of the single independent variable** or **the law of the single significant variable** could be applied to the results.

Some of the measures taken to ensure that the REST, Falcor and Relay+GraphQL situations were as equal as possible follows:

- The REST API was using Gzip [46] for data compression, therefore both the Falcor and GraphQL servers were also setup to use it.
- The REST API was caching responses from the backend data source. Therefore the GraphQL and Falcor servers also implemented caching. Before running the experiments, the caches were pre-populated because the latency introduced by fetching the data from the data source to the API would be the same for all three technologies.
- The REST, Falcor and GraphQL servers were hosted at the same location. Using this approach would decrease the impact network conditions had on the results.
- Client side caching was inactivated to make sure the measured value is the performance of the framework rather than the caching technology. This represented the worst case performance, as if all requests would lead to a miss in the cache.

3.3.3 Validity

Researchers must be sure that the conclusions drawn from experiments are valid. To assist in deducing valid insights from experiments, three types of validity was considered [40, pp. 271-272]:

- **Internal validity** concerns that conclusions about the relationship between independent and dependent variable is actually causal and not caused by some unmeasured extraneous factor [40]. The measures taken to decrease the risk of unknown extraneous variables were described in section 3.3.2 and throughout the report where appropriate.
- **Statistical conclusion validity** refers to the proper use of statistics when inferring relationships between variables. For example, if the conclusion is drawn by a true relationship or due to chance [40]. In this report, results are presented together with their respective error bounds and only changes of statistical significance are considered when drawing conclusions.
- **External validity** refers to how the findings of the study can be generalized to other settings, other than those tested in the experiments [40]. When designing the experiments, efforts were made to make the outcomes generalizable. A test case including parallel data fetching may be representative for other parallel data flows, but not for sequential flows. Test cases were created to include as many types of data flows as possible.

3.4 Hypothesis formulation

A powerful tool in the confirmatory phase of quantitative research is the usage of hypotheses, the researcher's predictions of the outcome of the study. They encourage the gathering of information and previous research required to make well-grounded predictions of the outcome of the study, stimulates the research by providing direction, a reporting framework and relational statements that are testable [40, pp. 82-83, 96].

Hypotheses can be derived both inductively and deductively. In inductive hypothesis formulation they are formed as a generalization of observed relationships between variables. The researcher makes an observation, notices trends or possible relationships and then formulates a hypothesis about the explanation of the observation. In deductive hypothesis formulation, the hypothesis are derived from theory by deductive reasoning. This results in hypotheses that are more general and are grounded in the existing body of knowledge [40, pp. 84-86].

There are three types of hypotheses [40, pp. 91-92]:

- The **research hypothesis** is derived inductively or deductively and is describing the expected outcome of the research.
- The **null hypothesis** states the negation of the research hypothesis, that there is no relationship between the variables that are studied. This hypothesis is used when determining if the outcome of the study is of chance or a real difference by using statistical methods. If the experiments indicate that the null hypothesis is unlikely to be true, it can be rejected and the researcher can conclude that the relationship introduced in the research hypothesis may be true.
- An **alternative hypothesis** is not the research hypothesis or the null hypothesis but states a different prediction of the relationship between variables. For example, if the research hypothesis predicts an increase in variable A and the null hypothesis that A will remain the same, an alternative hypothesis could be that A will decrease.

Ary *et al.* [40, pp. 86-90] provides criteria to be fulfilled by usable hypotheses. Hypotheses should state the relationship between variables, they must be testable, not contradicting well-established knowledge, and stated as simple and concise as possible. Before experimentally testing the proposed hypotheses, they should be evaluated against these criteria.

3.4.1 Hypothesis testing

After a hypothesis is stated and evaluated, it can be tested. Hypothesis testing involves the following steps [40, pp. 92-93]:

- State the relationships between variables that should be observed if the research hypothesis is true.
- State the null hypothesis.
- Select a research method that will be able to show if the predicted relationship exists or not.
- Collect empirical data.
- Using statistics, calculate the probability that the results are of chance when the null hypothesis is true.
- If the probability is low, there are sufficient evidence to reject the null hypothesis.

A rejection of the null hypothesis may provide enough evidence to support the research hypothesis.

3.5 Hypotheses

Hypotheses for changes in performance were derived deductively from theory. They are grounded in the knowledge that the frameworks implement a combination of techniques that are known to affect performance positively and therefore the frameworks should have the same properties.

All hypotheses about changes in performance are comparisons between using the frameworks and the REST approach that is currently deployed.

The research hypotheses were:

- Using Falcor will reduce the latency of data fetching in the studied client application.
- Using Falcor will reduce the data volume and the number of requests of data fetching in the studied client application.
- Using Relay+GraphQL will reduce the latency of data fetching in the studied client application.
- Using Relay+GraphQL will reduce the data volume and the number of requests of data fetching in the studied client application.

The null hypothesis is thus that no changes in latency, data volume or number of requests can be observed when using the frameworks.

An alternative hypothesis could be that latency is increased when using Falcor or Relay+GraphQL. This hypothesis is justified by the potential that introducing

an additional intermediary would introduce additional delays in communication or because introducing a query language may require heavier computation.

These hypotheses were considered passing the hypothesis validation introduced in section 3.4 because latency and data volume are properties that can be tested using experiments generating quantitative, numerical data. They do not contradict any previous research and they are concisely presented in one sentence each.

Chapter 4

Experiment design

The experiment design phase began with researching appropriate performance metrics and how they can be measured. This was done by studying previous research that conducted similar experiments with the goal of establishing performance in terms of latency and data transfer size. Thereafter additional factors of REST, Falcor and Relay+GraphQL which could effect the previously identified metrics was discussed, resulting in an extended motivation for the hypotheses.

Continuing the experiment design, the test application was studied further, latency and data experiments were designed, test cases were constructed and data sets to use in the experiments were specified.

4.1 Performance indicators

Coulouris *et al.* [12, p. 63] describes the following performance characteristics of communication in distributed systems:

- **Latency** is the delay between the beginning of a message transmission on the sender side to the beginning of the receiving on the recipient side. This delay includes the actual transmission delay, delay in network access and the delay imposed by the sender and receiver operating systems.
- **Bandwidth** is the amount of information that can be sent over a network in a set time. When multiple users are using the same network, they share the bandwidth.
- **Jitter** is the variation in time used to deliver messages over the network.

The full time required for transferring a message over the network is thus given by latency plus size of the message divided by data transfer rate, as of equation

4.1. The data transfer rate is the speed data can be transferred at between two computers once the transmission has begun [12, p. 83].

$$Message\ transmission\ time = latency + \frac{message\ length}{data\ transfer\ rate} \tag{4.1}$$

Adams [47] is discussing performance of web services and claims that performance of web applications often is measured by how quickly it responds to requests. This definition of performance relates to the message transmission time defined by Coulouris *et al.* [12, p. 83] so that the response is two times the transmission time plus the time used by the server to generate the response. However, Adams [47] also propose that a thorough evaluation of performance should include the effects of multiple simultaneous requests, latency in responding to requests, scalability and effects of load. Granularity of web services and the effects of serialization options are also discussed in terms of performance.

4.2 Measurement techniques

There are two metrics that are of primary concern in this work, latency and data volume. Data volume consists of two parts, the size of the transferred data and the number of requests.

By using statistical methods for calculating variance and standard deviation the amount of *jitter* can be measured. However, as variance in the measurements can not easily be attributed to the performance of the frameworks and likely is caused by extraneous factors like network conditions, jitter is not one of the main measurements in this study.

4.2.1 Latency

Latency of web services is often measured by the response times, that is time from the request is issued by the client and until the response is fully received. Zhou *et al.* [48] is describing how to design truly RESTful APIs for Software-Defined-Networks (SDNs) and in their work latency is measured by response times. In their experiments they sent random requests to the API at 500 ms intervals over an hour and recorded the average response times in 5 minute intervals. As the requests typically had response times of below 20 ms it can be assumed that the API is not congested. Huang *et al.* [32] is also using the same approach to measure average response times; they are measuring without load.

Li *et al.* [49] is measuring latency a bit differently. They are also using the average response times as their main metric, but tested how different workloads impacted

latency. This was achieved by varying the number of clients that were making requests to the server.

Because the technologies are used for data fetching in user facing systems, the latency of the entire software stack is interesting to measure. This means that measurements of latency should include the time from the fetching is initiated in the client, the request is sent to the server, handled by the server and until the response reaches the presentation layer of the client application. However, this leads to an increased risk of introducing extraneous variables that needs to be controlled.

4.2.2 Data volume

Data volume can be measured at the same time as latency by looking at the transfer size of the packets. This is exactly what Huang *et al.* [32] did by using a packet sniffer. Using this approach will also make it possible to measure the ratio of HTTP overhead by comparing the packet payload with the full packet size. The data volume is measured in bytes.

The number of requests can be measured by simply counting the requests.

4.3 Initial discussion of data fetching alternatives

This section is dedicated to an initial discussion about how the technologies for data fetching may affect performance in the test application. It can also be seen as an extended motivation for the hypotheses stated in section 3.5.

4.3.1 REST

When applying the REST architectural style as a whole it ”emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems” [2].

However, applying all constraints may have an impact on performance, which was observed by Zhou *et al.* [48]. In their initial implementation of a REST API they used fixed URLs. When they switched to the hypertext driven approach the average number of remote calls increased from 1,22 to 3,98 for completing an operation. When caching the hypertext links they could reduce the average number of requests to 1,70. Although not part of the evaluation, it is assumed that multiple round trips for completing an operation would lead to increased delays. In the case of Zhou *et al.* [48], they observed response times below 20 ms which they considered satisfactory and did not elaborate on.

Zhou *et al.* [48] also highlights one of the strengths of the REST architectural style. When using the hypertext driven approach, they successfully migrated the URL namespace transparently. This was achieved by only exposing a single fixed URL in the clients. This URL remained the same while the hyperlinks provided by its responses changed. If the clients used fixed URLs, these requests would have been invalid the API would respond with HTTP response code 404 (Not found) instead.

It is worth mentioning that in the work of Zhou *et al.* [48] the clients making requests are located within the same data center as the API server. Therefore latencies can remain low even though the average number of requests for performing an operation increase. In the case of Aftonbladet, the delay introduced by doing additional sequential requests over a mobile network would likely introduce a major decrease in performance.

4.3.2 Falcor and Relay+GraphQL

The predictions of performance of Falcor and Relay+GraphQL are derived from properties of the technologies and the concepts they are implementing. It is known that caching is a commonly deployed solution for reducing the average response times [12, pp. 49-50], therefore a framework implementing caching should also have that benefit.

Both frameworks allow aggregation of requests from the client to the server and thus the resolving of sequential requests is moved to the server. This may benefit latency assuming that the round trip times from the server to datasources are lower than from the client to the data sources. The prediction is that this is especially visible when sequential requests are made to internally hosted resources because communication within the same data center is assumed to be quicker than external communication. A note to be made is that when using this approach partial responses may not be available and response times are bounded by the slowest request.

On the server side, one way Falcor distinguish from GraphQL is that GraphQL implements a query language while Falcor does not. This is discussed in by Jafar Husein [50], an architect of Falcor. Having the power of a query language however comes with a computational cost and developers should ask themselves if they need that power. This provides a case for measuring differences in computation times on the server side.

In the discussion [50], they also talk about how Falcor and GraphQL differs in expressiveness. For example, Falcor does not support open ended queries. Evaluating the set of features the frameworks provide was however not the main focus of this work.

4.4 Test application

Relay is a client library for consuming GraphQL services in React applications, therefore the choice of application to study was limited to React applications. Also, as both Falcor and Relay+GraphQL have reference implementations written in JavaScript, a web application was the best choice of platform for the evaluation. Popular mobile platforms like Android and iOS do not have built-in support for executing JavaScript code and using a custom interpreter could affect the measured performance.

Some of the frameworks were implemented in multiple programming languages [51]. To make the comparison more fair, only the reference implementations were used. This ensured that implementations were written in the same language and could be executed in the same environment.

This study is based on a real world use case from Schibsted's product Aftonbladet, one of Sweden's major newspapers. The application that was studied in this project was a page within a React web application for presenting news by Aftonbladet. This page contained the currently viewed article in full text, its statistics from social networks and a list of teasers for today's most read articles. It could be considered a typical news web application or even representative for any modular multimedia web application.

The data fetching tasks are summarized below:

- **Current article**

  This is the full representation of an article, containing both full text and metadata. This data is exposed through an internally hosted API.

- **Social statistics**

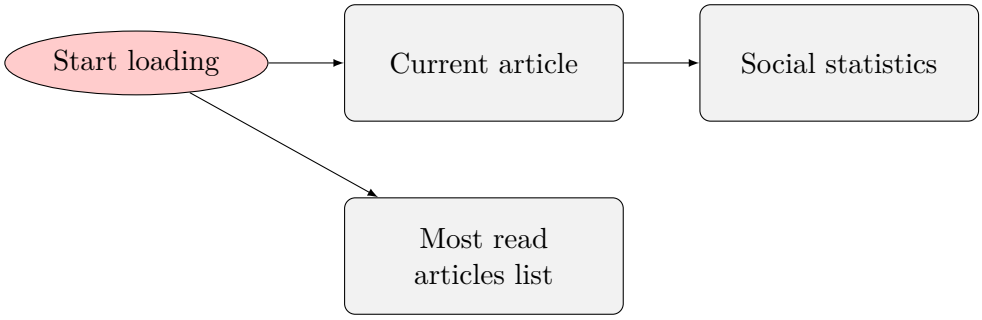
  This data is supplied by respective social network and currently only Facebook is supported. The request to Facebook's APIs require the full URL to the article, which is only available after fetching the article data. Therefore this request needs to be executed sequentially after the full text article loaded. Because of restrictions on how often the Facebook API can be invoked from the same application, requests needs to be cached internally.

- **Most read articles list**

  This is a list of the most read articles with their teaser information. The teaser information typically includes only the title, preamble and image of each article. This data set is accessed through an internally hosted API.

Figure 4.1 provides an overview of the data that was fetched by the web application. The oval block represents the entry point, when loading is initiated by the web browser.

Figure 4.1. Overview of data fetching in the studied application.



4.4.1 REST correctness

The internally hosted APIs for fetching article data are not truly RESTful because they violate several of the constraints of the specification by Fielding [2]. The following violations were found:

- **Exposing fixed resource URLs**

  The APIs are not hypertext driven, which introduces a tight coupling between the API server and the clients. This is one of the constraints that Fielding observed to be violated often in practice [52].

- **Not responding to content negotiation**

  The APIs are only providing JSON data, even when it is indicated that the requester wishes to get a response in XML. This was tested by modifying the HTTP-header "Accept" and setting it to "application/xml".

The impact of violating or complying with the constraint that REST APIs must be hypertext driven was discussed in section 4.3.1. It turned out there exists evidence that ignoring the constraint may have a positive impact on performance in terms of latency and number of requests. However, complying makes the system easier to maintain and less error prone by a looser coupling between client and server.

4.5 Latency experiments

The latency experiment was designed so latency was measured in the client application. Just before the data fetching request is sent, a timestamp is recorded using the built-in JavaScript method **new Date().getTime()** and stored in memory. When the response is received, another timestamp is recorded. The duration of the request-response interaction is given by the difference between the two timestamps. After the duration is calculated, it is stored persistently to be retrieved for later analysis.

Recording the timestamp is assumed to have no side effects or affect the performance of data fetching. It is also assumed that this is the case for saving the duration to persistent storage. However, as a precaution an additional delay of 500 milliseconds was added between saving the previous result and initiating the next request.

The latency experiment was repeated 10 times for each entry in the data set in order to reach statistically significant results with sufficiently small error bounds. The repetition count was derived experimentally by running the latency experiment for 5, 10 and 20 repetitions. For 10 repetitions, assuming normal distribution, the 95 % confidence interval for the entire data set was usually less than 3 % of the response time which was considered sufficient.

4.6 Data volume experiments

As the frameworks hide some metadata used by the frameworks themselves from the client, it is interesting to look into the raw transfer size. The initial plan was to use the packet sniffer Wireshark [53] to get this data. However, an easier alternative was found and adopted. From the developer console in Google Chrome, all network interactions could be exported in HTTP Archive (HAR) [54] format. This included all information needed to measure data volume and the metrics provided in the HAR was consistent with measurements by Wireshark.

Because the data volume is assumed not to change between subsequent requests for the same content, there is no need to record data volume as many times as the latency was recorded. However, because the frameworks may introduce optimizations for subsequent requests, they were recorded multiple times for one article to establish the behavior of the framework and thereafter only once for each entry in the data set. After an initial investigation it was concluded that transfer sizes remain the same for consecutive requests when local client caching was turned off.

The number of requests was counted using the data provided in the HAR file.

4.7 Test cases

The three pieces of data fetched by the test application were combined to construct test cases. The main objective when creating the test cases was to provide enough test cases to answer how the data fetching methods performed for multiple types of data flows.

Test cases were created for fetching a single piece of data, sequential data fetching flows and parallel flows. The sequential and parallel data flows were considered "building blocks" that could be combined to create more complex flows. To test

how large the impact of reducing the amount of data returned by the services were, some test cases used queries and paths introducing optimizations (filtering).

The test cases are summarized below:

- **Full article**

  Fetching the full text article using a fat query (a query for all fields possible) for the article data. This established the baseline for performance while fetching the exact same piece of data from the different services.

- **Optimized article**

  Fetching the full text article with the optimized ImageAsset described in section 5.3.2. This showed how filtering data and thus decreasing the size of the response, while introducing slightly heavier server logic, affected performance.

- **Full article + teasers**

  Fetching the full text article and teasers in parallel using fat queries for both the article and teaser data. This showed how parallel data fetching affected performance.

- **Optimized article + optimized teasers**

  Fetching the full text article and teasers, both with the ImageAsset optimization. This showed how parallel data fetching in combination with data filtering affected performance.

- **Full article + social data**

  Fetching the full text article and the social data sequentially. This measured performance of sequential data fetching using fat queries for both the article data and the social data.

- **Optimized article + social data**

  Fetching the full text article with the ImageAsset optimization and the social data sequentially. This showed how sequential data fetching while reducing the amount of data fetched affected performance.

- **Full article + teasers + social data**

  Fetching the full text article and teasers in parallel while fetching the social data sequentially after the full text article finished loading. This showed performance of sequential and parallel data fetching in combination using fat queries for all three data pieces. This was in fact the largest possible query of the implementation used in the experiments.

- **Optimized article + optimized teasers + social data**

  Fetching the full text article and teasers with the ImageAsset optimization in parallel while the social data is fetched in sequence with the full text article. This showed how performance was affected by decreasing the amount of data sent using both parallel and sequential data flows.

- **Optimized components only**

  Fetching the minimal information needed to render an article with the ImageAsset optimization. This showed how more aggressive data filtering affected performance.

- **Title only**

  Fetching only the title of an article. This data was fetched using a single path in Falcor and close to the simplest possible query in GraphQL. This was assumed to show the smallest response time possible for Falcor and GraphQL while still fetching useful data.

4.8 Experiment data sets

The experiments were executed on 50 different articles provided by a service returning the latest articles published on Aftonbladet's website. The articles were from a continuous selection of this list, and were picked without inspecting their contents. The article identifiers used in the experiments are available in appendix A.1.

This data set was chosen as it consists of real data that represents the common use case of the actual application, while providing variation.

Chapter 5

Experiment implementation

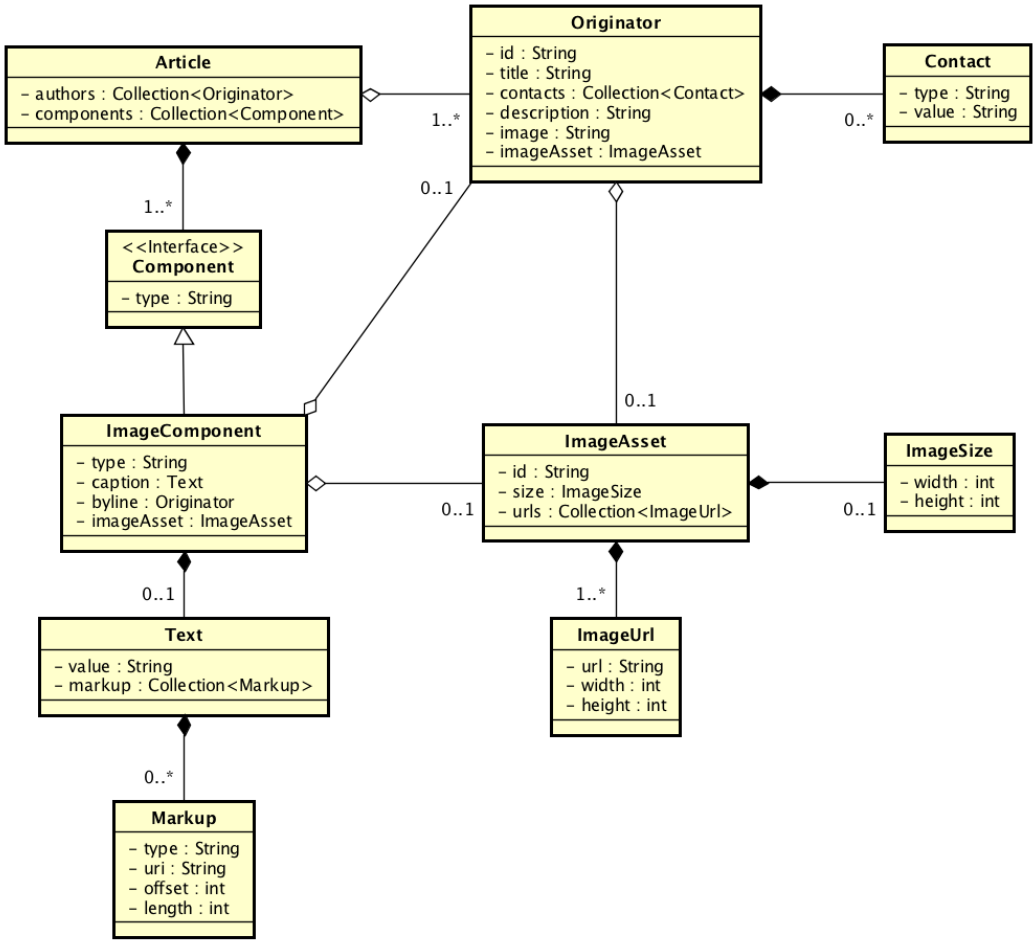
In this chapter the experiment implementation is presented, including prerequisites like modeling the data so it can be exposed through Falcor and GraphQL. First a subset of the data model is presented, which will be used as an example during the entire chapter. Then the implementations using Falcor and Relay+GraphQL are described using the example data model, the experiment setup is presented and the test environment is specified.

5.1 Article data model

The full text article was the most important and complex data type used by the test application. The full specification of the data format includes what content to present to the reader in the client applications, how it should be styled, and additional metadata. However, the full specification was too extensive to present as a whole in this report, therefore this section is based on a selected sub-section. Figure 5.1 is showing an UML diagram over the partial full text article that is considered an example in this report.

To give the reader an understanding of how extensive the full article specification was it should be mentioned that the Article class has an additional 13 fields of which 8 are complex data types. Also, the Component interface had an additional 11 child types. All these were modeled in Falcor and GraphQL, however they are not presented in the report because the selected subset should be able to highlight the key concepts without unnecessary repetition.

Figure 5.1. Partial schema for the full text article.



5.2 Modeling in Falcor

The Falcor backend service was built using a Falcor router. The JavaScript implementation of the router is holding a list of tuples consisting of a path and a function named *get* that is resolving the path and providing the actual data.

For the authors of an article, the id, title, description and image fields are resolved using the path in figure 5.2. Note that the line break was introduced only for fitting the path in the report and does not fill any purpose in Falcor.

Figure 5.2. The JSON path for fetching the id, title, description and image of an author of an article.

```
article[{keys:articleId}].authors[{integers:authorIndices}]
.['id','title','description','image']
```

When a request for the path is made, the router forwards it to the corresponding get function. Figure 5.3 shows the function resolving the path defined in 5.2. First, it will get the raw article data from an article service using the provided articleId (line 5). Then, for each author that were defined in the authorIndices (line 8) it will check what fields were requested in the path (line 9) and create tuples consisting of the path and value (line 10-13). These are then returned when the entire path is evaluated (line 17).

Figure 5.3. The function for resolving the id, title, description and image fields of an author.

```
1  get: function(pathSet) {
2      const articleId = pathSet.articleId;
3      const authorIndices = pathSet.authorIndices;
4
5      return getArticle(articleId).then(function(article) {
6          var pathValues = [];
7
8          forEachIfExists(article.authors, authorIndices, function(index, authorForIndex) {
9              forEachIfExists(authorForIndex, pathSet[4], function(key, valueForKey) {
10                 pathValues.push({
11                     path: ["article", articleId, "authors", index, key],
12                     value: valueForKey
13                 });
14             });
15         });
16
17         return pathValues;
18     });
19 }
```

Because path traversal was commonly implemented using forEach-loops with null-checks, a convenience function called forEachIfExists was created. (See figure 5.4.) This function is taking a collection of items, what indices to traverse and a callback. The callback function will be called for each collection item corresponding to an index. It is also implementing null-checks because Falcor does not handle paths resolving to null or "undefined" gracefully.

Figure 5.4. The helper function forEachIfExists.

```
1  function forEachIfExists(collection, indices, callback) {
2      if (collection && indices) {
3          indices.forEach(function(index) {
4              const itemForIndex = collection[index];
5              if (itemForIndex) {
6                  callback(index, itemForIndex);
7              }
8          });
9      }
10 }
```

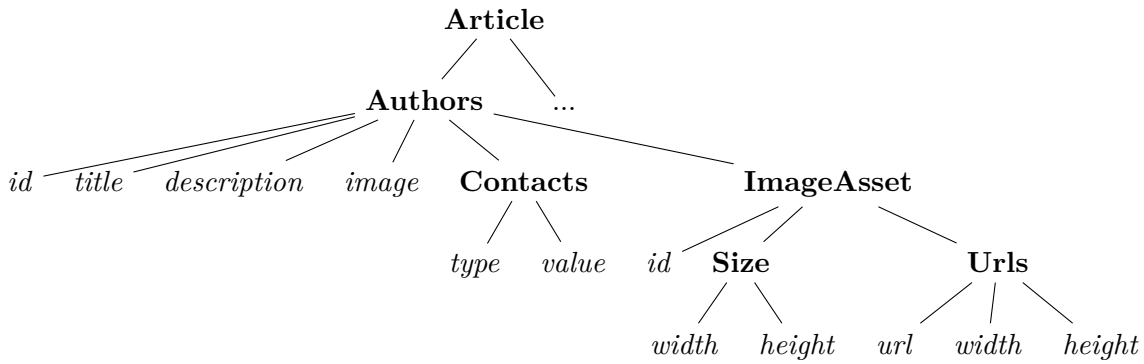
So far mapping paths to data was quite straight forward, however the author type consisted of multiple nested objects. To make them queryable through the router four additional paths had to be defined. (See figure 5.5.)

Figure 5.5. The JSON paths for fetching nested objects of the author type.

```
article[{keys:articleId}].authors[{integers:authorIndices}]
.contacts[{integers:contactIndices}].['type','value']

article[{keys:articleId}].authors[{integers:authorIndices}]
.imageAsset.['id']

article[{keys:articleId}].authors[{integers:authorIndices}]
.imageAsset.size.['width','height']

article[{keys:articleId}].authors[{integers:authorIndices}]
.imageAsset.urls[{integers:urlIndices}].['url','width','height']
```

A general rule for creating routes can be visualized with tree graphs. Figure 5.6 is showing a tree for the article and its Authors field. The rule is that one route is needed for each branch-node that have leaf-nodes for children. In the graph it can easily be seen that five branch-nodes have leaf-nodes as children, and therefore five routes were needed. (Those of figure 5.2 and 5.5.)

Figure 5.6. A tree representation of the author data.



Each path defines its own get function for resolving the path with a similar implementation as the one shown in 5.3. This caused a lot of code duplication, which was one thing GraphQL solved more smoothly.

The component type was initially modeled using the Falcor specific data type $atom. An atom is a wrapper for any unstructured JSON data and makes it possible to expose all components without adding paths for each specific component type. However, this makes the fields of the components non-filterable. In the end, the component was modeled to make all fields queryable and make the Falcor and GraphQL implementation as similar as possible.

5.2.1 Data formatting

The data provided by Falcor is returned in a format defined by the requested JSON path. For example, the path "article[123].authors[0].id" will be returned in a JSON document with the same structure as the keys. (See figure 5.7.)

Figure 5.7. A falcor response for the id of the first author of article 123.

```
{
    "jsonGraph": {
        "article": {
            "123": {
                "authors": {
                    "0": {
                        "id": "author1"
                    }
                }
            }
        }
    }
}
```

This makes responses predictable and makes local JSON data accessible using the exact same paths as remote data. However, the JSON document provided by Falcor could not be returned on the same format as the currently deployed REST endpoint, which would make a migration more complex.

Figure 5.8 is showing the format of the data from the REST endpoint with unrelated fields omitted. The main difference is the depth of the JSON object tree, where the first three levels of figure 5.7 are not present in the REST endpoint response. Also, the authors object is represented by a map in the Falcor response and as an array in the REST endpoint response.

Figure 5.8. A subsection of an article from the REST endpoint, showing the format of author data.

```
{
  "authors": [
    {
      "id": "author1",
      ...
    },
    ...
  ],
  ...
}
```

5.2.2 Granularity of routes

Because Falcor does not support open ended queries [50], modeling a list of objects without a known length is problematic. One choice is to use the $atom data type and model the list as a block of unstructured JSON data. However, this makes the data fields non-filterable.

Another choice is to model the list as usual, and always query a significantly higher number of indices than it is realistic that the list contains. For example, if a list of components is usually somewhere between 1-20 elements long, but even though it is unlikely, it is possible that there exists an article consisting of 50 components. However, the risk that there exists an article with more than 1000 components is so small it can be considered impossible and therefore always requesting the 1000 first components would return all existing components.

A common pattern for requesting lists of unknown length is to expose a route that is returning the length of the list and then in a subsequent request fetch the actual list now when its size is known [55]. However, this is introducing a second round-trip which is one of the problems using Falcor aims to eliminate and using this approach would thus defeat one of the purposes of Falcor.

There is a tradeoff to be made in how granular the data can be modeled. Using the first approach, the data can only be accessed as a whole and thus a larger transfer size is expected unless all fields are required. Resorting to equation 4.1, it can be deduced that this would lead to an increased transfer time at the network layer. While data can be modeled more fine-grained using the second approach, it is unknown how querying for significantly more data than there exists affects performance. On line 8 of figure 5.3, it can be observed that N additional indices would cause the for-each loop to execute N more times, causing N more memory accesses to the authors array. It is unknown how the performance of Falcor's path validation is effected, but it turns out this approach introduced unnecessary data to be sent over the network. This is because when the Falcor client requests a piece of data that does not exist on the server, the server will respond with an empty atom telling the client that the data does not exist. This empty data container will be added to the cache and a repeated request can return this empty value from the cache instead of doing another round trip to the server.

In the experiments, the over-fetching approach was selected because both the Falcor and GraphQL implementations should have similar data filtering possibilities and introducing a second round trip was assumed to be a larger penalty than over-requesting data. The pre-defined lengths used to fetch lists can be found in appendix A.3.

5.3 Modeling in Relay+GraphQL

To create a backend service that is exposing the article data it first needs to be modeled as a GraphQL schema. This was done using the JavaScript reference implementation of GraphQL [56].

The entry point for fetching an article is defined in the GraphQL type named Query as part of the mandatory naming convention. Figure 5.9 shows the Query type, defining a top level operation called article. The operation returns data of the articleType and accepts an argument id of the GraphQLID type.

The data is fetched using the resolve function, which is getting an article by the defined id from an article service. The resolve function can be implemented on any field of a GraphQL object, or omitted if the field should be resolved following the JSON object path defined by the field name. This is one of the main differences from how routes in Falcor are implemented and makes modeling in GraphQL require a lot less code.

Some additional top level operations are omitted from figure 5.9 and marked with three dots (...). Future code snippets will use the same notation for marking omitted rows.

Figure 5.9. The GraphQL type named Query, the entry point for any query to the GraphQL server.

```
var queryType = GraphQLObjectType({
  name: 'Query',
  fields: {
    ...
    article: {
      type: articleType,
      args: {
        id: {
          type: GraphQLID
        }
      },
      resolve: function(_, args) {
        return getArticle(args.id);
      }
    },
    ...
  }
});
```

The articleType of figure 2.9 corresponds to the Article class defined in figure 5.1. The GraphQL implementation is shown in figure 5.10. The article has got a mandatory field (GraphQLNonNull) consisting of a list (GraphQLList) of objects of the Originator type.

Figure 5.10. The Article type.

```
var articleType = new GraphQLObjectType({
  name: 'Article',
  ...
  fields: {
    authors: {
      type: new GraphQLNonNull(new GraphQLList(originatorType))
    },
    ...
    components: {
      type: new GraphQLNonNull(new GraphQLList(componentInterface))
    },
    ...
  },
  ...
});
```

The components list is modeled in a similar way. They consist of a mandatory list of objects of the componentInterface, corresponding to the Component interface in figure 5.1. Using the interface makes it possible to serve multiple component types for the same field.

The ImageComponent type, seen in figure 5.11, is implementing the componentInterface and simply contains the fields of the specification. Fields are either of the simple data types like GraphQLString or complex data types like imageAssetType. The function isTypeOf is used to resolve objects implementing an interface to the specific types. This is required for any type that implements an interface.

5.3.1 Model expressiveness

In the full article specification there are fields modeled by maps without pre-defined keys. Exposing this kind of data directly is not possible in GraphQL. Some available solutions are to either model the map as a list of tuples including the key-value pairs or to create a custom resolve function that accepts the key that should be fetched.

This was the only exception where GraphQL could not provide JSON responses on the same format as the REST endpoint. In the experiments, the map was serialized and served as a whole using the GraphQLString type. This was similar to the Falcor implementation, where maps were exposed as atoms.

Figure 5.11. The ImageComponent type.

```
var imageComponentType = new GraphQLObjectType({
  name: 'ImageComponent',
  interfaces: [
    componentInterface
  ],
  fields: {
    type: {
      type: new GraphQLNonNull(GraphQLString)
    },
    caption: {
      type: textType
    },
    byline: {
      type: originatorType
    },
    imageAsset: {
      type: imageAssetType
    }
  },
  isTypeOf: (value) => value.type && value.type === 'image'
});
```

5.3.2 Further optimizations

The ImageAsset type contains a list of ImageUrl objects. The ImageUrl object is a container for an URL to the image and the width and height of the image. The REST endpoint will return a list of ImageUrls with widths ranging from 300 to 1000 pixels in 50 pixel increments. That is 15 ImageUrl objects, even though the client application in most cases already know exactly what image size it wants to use.

As an optimization a new field **url** with an argument **width** was introduced. This field had a custom resolve method traversing the original list and returning the ImageUrl with the width closest to the requested width. This means that the size of the url list can be reduced to 1/15th of the original size.

The same optimization was also introduced in the Falcor implementation.

5.4 Test application implementation

After exploring the implementation of the real news application further it was concluded that a migration to both Falcor and Relay+GraphQL would not be feasible

during the course of this project.

There were multiple reasons behind this decision. Firstly, the application was using Redux [57], a library for handling application state. There have been some debate on how Relay relates to Redux and whether they are compatible in the open source community [58, 59]. While exploring the options for client setup an integration between Redux, Relay and Falcor was considered too complex. Secondly, a complete re-write of the data flow of the application, moving away from the Redux solution, would likely have been even more time consuming.

Therefore, a separate application was created that mimicked the data requirements of the original application, but without the actual presentation layer. This was a React application consisting of three pages, one each for REST, Falcor and Relay+GraphQL. Each page was only presenting what data fetching method it was using and after the experiments finished it printed the experiment result data to be analysed. In the "componentDidMount"-function[1], data fetching was initiated using a function named loadAll.

One of the main objectives when designing the test application was to use the same notation for the REST, Falcor and Relay+GraphQL implementations whenever possible. The loadAll function is a recursive function that is taking a list of articleIds for the experiment and how many repetitions each article should be fetched for. Because of the asynchronous nature of JavaScript, using recursion was the preferred method to repeat a task after the previous task finished.

The loading using each data fetching method was configured to use a common test configuration to steer what data to fetch. This made test execution easier and less error prone as only one configuration file needed changing to execute experiments for all data fetching methods.

5.5 Experiment environment and setup

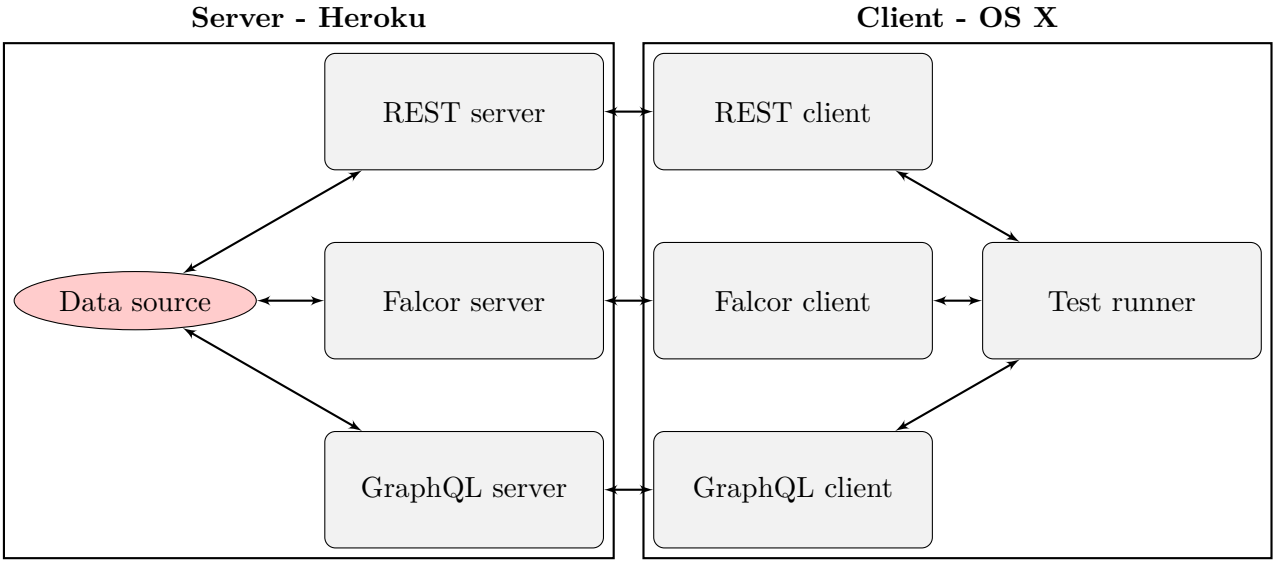
An overview of the setup used for executing the experiments is shown in figure 5.12.

The REST, Falcor and GraphQL servers were co-hosted as an application on Heroku [61]. The application used their "free" dyno, an application container with 512Mb RAM running the "cedar-14" (Ubuntu 14.04) stack. The European region was selected to minimize network latency.

One of the dangers of using a cloud provider for hosting is that performance may vary depending on how many tenants are using the cloud infrastructure. In an

[1] An application lifecycle callback invoked after the initial rendering of the React component. The react documentation suggests that integration with other JavaScript frameworks or data fetching is initiated in this method [60].

Figure 5.12. Overview of the setup used for executing the experiments.



attempt to minimize this impact, the test cases for each technology were executed in quick succession and the experiments for a test case were executed for all technologies before the next test case was used. The motivation for doing so is that the state of the cloud infrastructure and network is likely to be in the same condition as it was in the recent past. However, this is not guaranteed.

The choice of moving the server to a separate host and thus risk introducing additional extraneous variables were justified by two reasons. Firstly, it was more representative of a real world use case of the service where the server is located at a different host than the client. Secondly, when co-hosting both the client and server locally response times of less than 5 milliseconds were noted. This was close to the minimal measurable unit, 1 millisecond, and changes would be nearly indistinguishable. Using less powerful hardware and a slower network connection made performance differences more visible and easier to measure.

The test application was hosted locally on Mac OSX 10.10.5, with a 2,5 GHz quad core processor, 16 Gb RAM and a wireless connection (802.1X). Chrome version 49.0.2623.112 was used for loading the client application and executing the experiments. It was verified that the client never ran out of computing resources during the tests.

Both the server application and test application were using NodeJS version 4.2.0 and Node Package Manager (NPM) version 2.14.7. A full summary of NPM packages used and their versions are included in appendix A.2.

Chapter 6

Results

In this chapter the results from the experiments are reported.

When examining the collected data, it was concluded that it was not normally distributed. This was verified both graphically and by checking how many of the data points were within one, two and three standard deviations of the mean. In normally distributed data sets, approximately 68 % of the values are within one standard deviation, 95 % within two and and 99,7 % within three [62, p. 149]. This was not the case for the data collected in this study, which had a larger portion of the data points within one standard deviation, 76-95 %. The full result of the 68-95-99,7-test is shown in appendix A.5.

Because the collected material was not normally distributed, confidence intervals could not be applied. However, there are weaker statistical models available for interpreting the reliability of collected data. For example, Chebyshev's inequality [62, p. 130], guaranteeing that most values are near the mean for any probability distribution. More precisely, at most $1/k^2$ of the values can be more than k standard deviations away from the mean. This guarantees that at most 25 % of the readings are more than two standard deviations away form the mean. However, in reality the number of readings more than two standard deviations away from the mean was significantly lower. (See appendix A.5.)

In the following sections, the median value is given for latency, while the average value is given for transfer size. Details about the distribution of the results are available in the graphical representations and in appendix A.5. The full data transfer size results are available in appendix A.6. When referring to Relay in graphs, the combination of Relay+GraphQL is implied.

6.1 Reading box plots

Latency data in this chapter is represented as box (and whiskers) plots. Because there exists multiple common alternatives for what measurements to include in the plot, the ones used in this report are listed below:

- The bottom of the box marks the first quartile.
- The top of the box marks the third quartile.
- The line inside the box marks the median value.
- The bottom whisker marks the 9th percentile.
- The top whisker marks the 91st percentile.
- Outliers (data points outside the whiskers) are omitted from the plot.

The reason outliers were omitted was firstly because they decrease the readability of the graphs by affecting the length of the Y axis considerably. Secondly, they were considered more likely to be caused by measurement errors related to networking or the underlying cloud infrastructure, rather than functionality of the actual data fetching technologies.

The reader should be aware that outliers were only omitted from the box plots. No outliers were removed from the raw data or for other statistical analysis.

6.2 Reading bar graphs

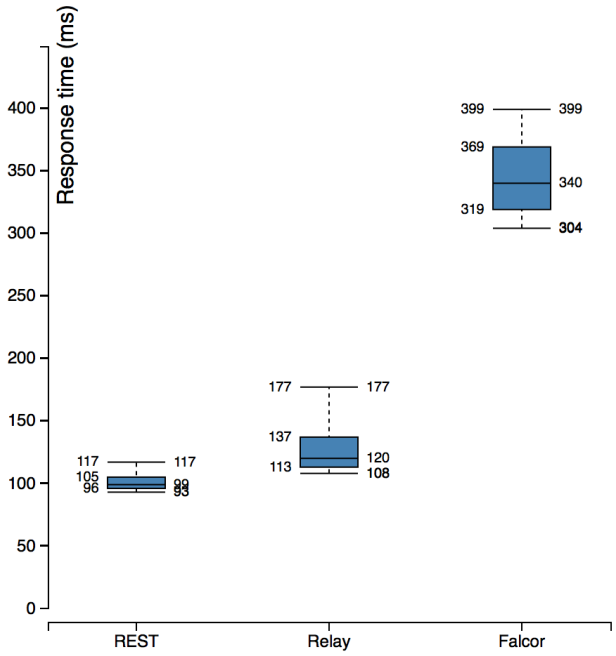
Transfer size data in this chapter are reported using bar graphs where the bars marks the mean value. Each bar includes error bars, representing the standard deviation in both positive and negative direction. The length of the error bars are thus two times the standard deviation.

6.3 Full article experiment

Figure 6.1 is showing the results for latency of fetching the full text article using the three data fetching technologies. The REST endpoint returned quickest, with a median response time of 99 ms. Relay+GraphQL showed an increased response time, 120 ms (+18 %), and using Falcor increased the response time to 340 ms (+208 %).

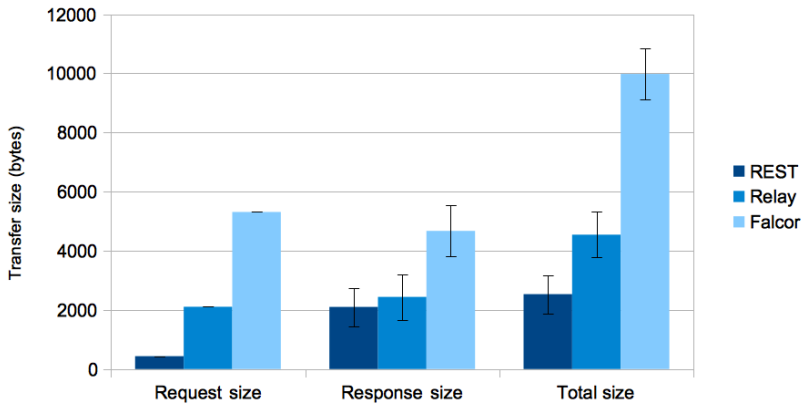
Figure 6.2 is showing the resulting transfer sizes. The REST endpoint had the smallest request and response sizes, resulting in a lowest total transfer size, on average 2530 bytes. The Relay+GraphQL implementation and the REST endpoint

Figure 6.1. Latency of fetching the full article using REST, Relay and Falcor.



showed a similar response size, but the request size was larger for Relay+GraphQL. This resulted in an increased total transfer size, 4541 bytes (+80 %). Falcor had significantly larger request and response sizes than the other two methods, 9978 bytes (+294 %).

Figure 6.2. Transfer size of fetching the full article using REST, Relay and Falcor.

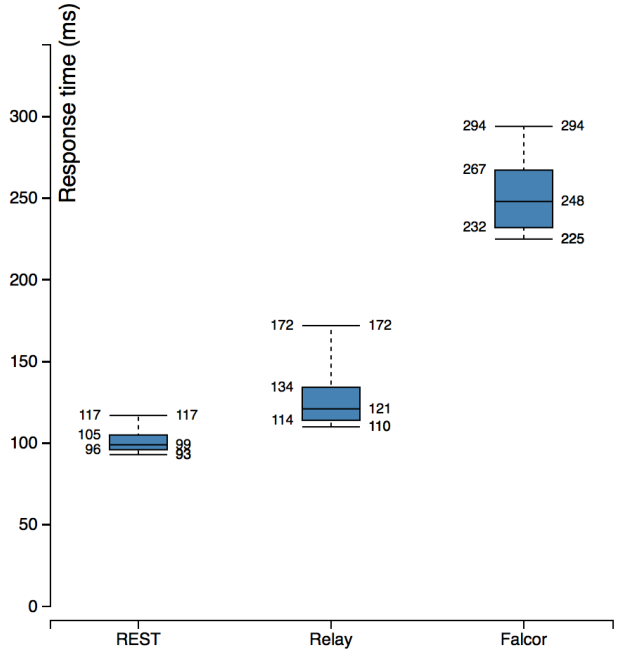


All three data fetching methods required one network request to fetch the data.

6.4 Optimized article experiment

Figure 6.3 is showing the results for the optimized article experiment. For this experiment, the same requests were made to the REST API as in the full article experiment presented in section 6.3. The REST API had a median response time of 99 ms. For Relay+GraphQL, an increased response time was recorded, 121 ms (+19 %). Falcor also showed an increased response time, 248 ms (+129 %). Falcor showed an improvement over the non-optimized variant used in the full text article experiment, but it still performed worse than the REST API.

Figure 6.3. Latency of fetching the optimized article using REST, Relay and Falcor.



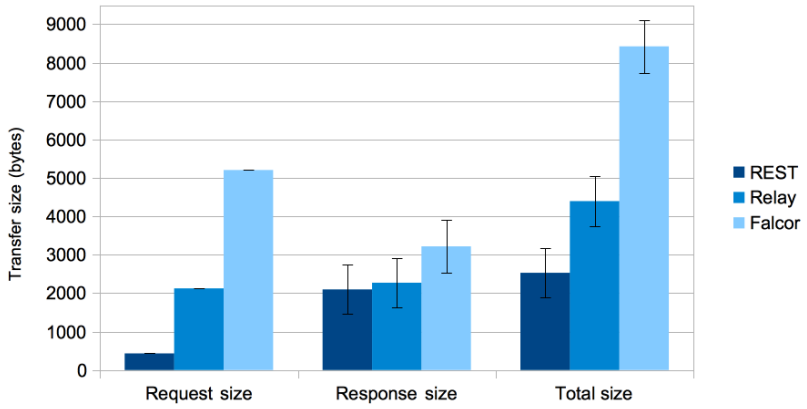
The results for transfer size shown in figure 6.4 are similar to the full text article experiment. The REST API was using the least amount of bytes, 2530 bytes on average, followed by Relay+GraphQL with 4396 bytes (+74 %) and Falcor with 8425 bytes (+233 %). However, both Relay+GraphQL and Falcor showed a slight improvement over the full text article experiment.

All three data fetching methods required one network request to fetch the data.

6.5 Full article + teasers

When fetching the full text article and teasers in parallel, the results in figure 6.5 are showing that Relay+GraphQL had the lowest response time. The REST API had

Figure 6.4. Transfer size of fetching the optimized article using REST, Relay and Falcor.



a median response time of 146 ms. Using Relay+GraphQL decreased the response time slightly to 141 ms (−3 %), while Falcor increased it to 343 ms (+132 %).

Figure 6.5. Latency of fetching the full article and teasers in parallel using REST, Relay and Falcor.

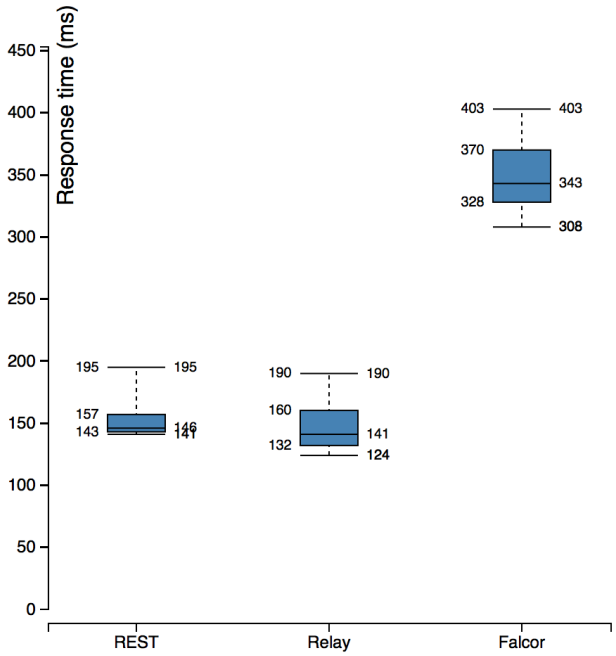
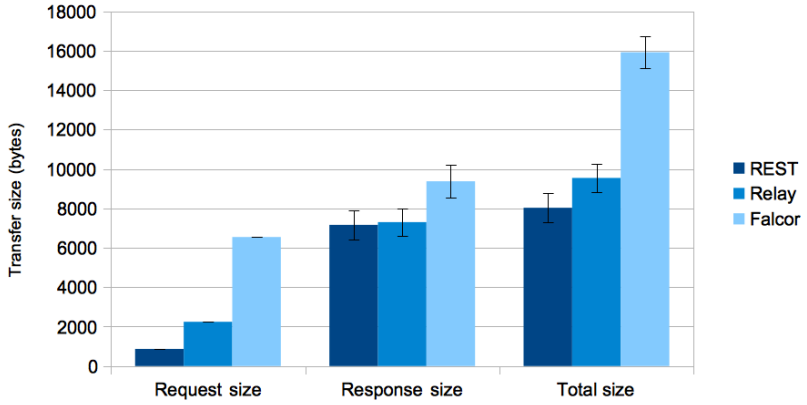


Figure 6.6 is showing the transfer sizes recorded while fetching the full text article and teasers in parallel. The REST API performed best with an average total trans-

fer size of 8026 bytes. Using Relay+GraphQL increased the total transfer size to 9541 bytes (+19 %), and Falcor increased it to 15915 bytes (+98 %). However, the response sizes of the REST API and Relay+GraphQL are nearing each other when introducing more network requests to the REST API.

Figure 6.6. Transfer size of fetching the full article and teasers in parallel using REST, Relay and Falcor.



The REST API used two network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

6.6 Optimized article + optimized teasers experiment

Figure 6.7 is showing the results of fetching the full text article and teasers in parallel, both with the ImageAsset optimization. Just as for the non-optimized case, Relay+GraphQL showed the lowest response time. The REST API (Exact same measurement as for the non-optimized case presented in section 6.5.) had a median response time of 146 ms. With the optimization, Relay+GraphQL reduced the response time to 122 ms (−16 %), while Falcor increased it to 260 ms (+77 %). This was a great improvement over the non-optimized case.

This was the first experiment where one of the frameworks outperformed the REST API in terms of transfer size. (See figure 6.8.) Both Relay+GraphQL and Falcor showed decreased response sizes and in total Relay+GraphQL outperformed the REST API. The REST API showed an average total transfer size of 8026 bytes and Relay+GraphQL showed an improvement, 7373 bytes (−8 %). However, the request size of Falcor caused the total size to be significantly larger than the alternatives, resulting in a transfer size of 12270 bytes (+53 %).

Figure 6.7. Latency of fetching the optimized article and optimized teasers in parallel using REST, Relay and Falcor.

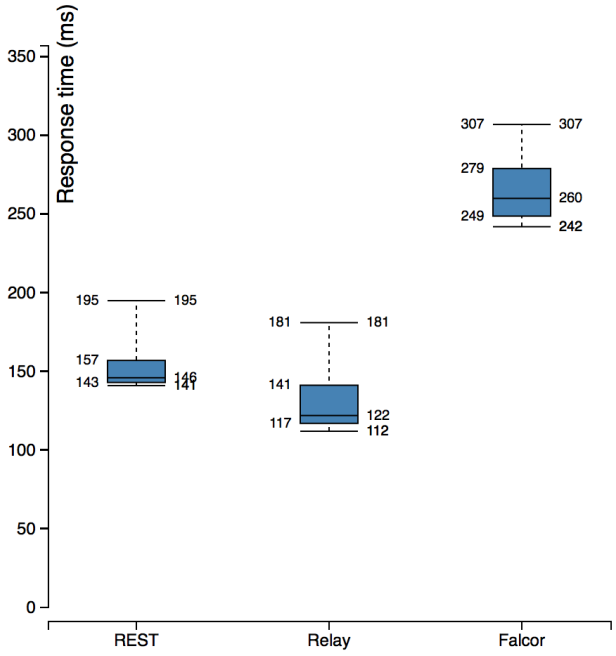
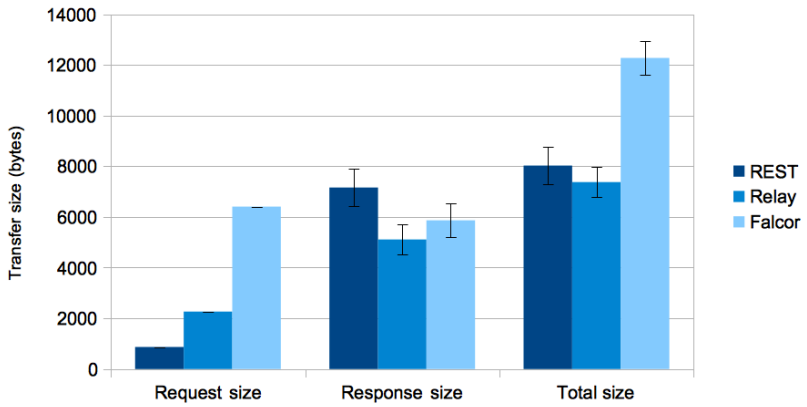


Figure 6.8. Transfer size of fetching the optimized article and optimized teasers in parallel using REST, Relay and Falcor.

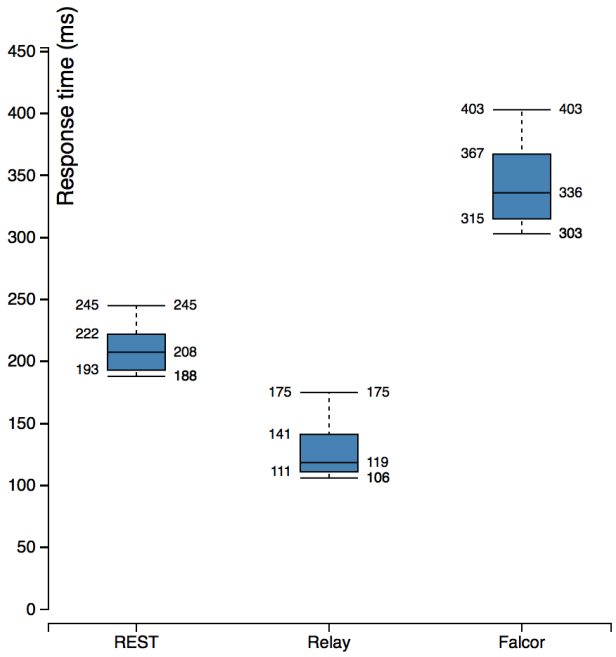


The REST API used two network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

6.7 Full article + social data experiment

The greatest improvements over the REST API were shown in experiments including sequential data fetching. As can be seen in figure 6.9, Relay+GraphQL performed best in the experiment where the full text article and social data were fetched sequentially. While the REST API returned in 208 ms on median, Relay+GraphQL decreased the response time to 119 ms (−44 %). Falcor still performed poorly with a 336 ms (+64 %) median response time.

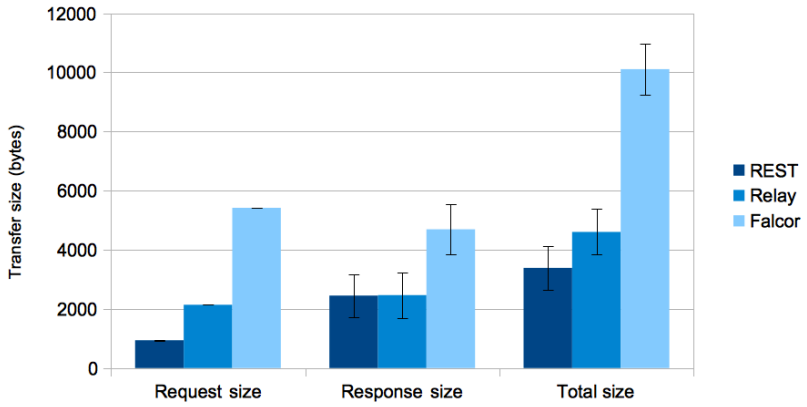
Figure 6.9. Latency of sequentially fetching the full article and social data using REST, Relay and Falcor.



The recorded transfer sizes are shown in figure 6.10. Using the REST API resulted in the smallest total transfer size with 3384 bytes on average. This was followed by Relay+GraphQL with 4600 bytes (+36 %) and Falcor with 10106 bytes (+199 %).

The REST API used two network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

Figure 6.10. Transfer size of sequentially fetching the full article and social data using REST, Relay and Falcor.



6.8 Optimized article + social data experiment

The results shown in figure 6.11, from the experiment where the optimized article was fetched in sequence with the social data, show similarities with the non-optimized variant. The REST API (Exact same as in the non-optimized variant, presented in section 6.7.) showed a median response time of 208 ms. Relay+GraphQL showed a decreased response time, 127 ms (−40 %), while Falcor increased the response time to 360 ms (+76 %).

This result differs from the other results when the optimized article format is used. In previous experiments, the response time of the optimized variant would be lower than the non-optimized variant, but these results showed the opposite.

Figure 6.12 is showing similar results for this experiment as the non-optimized counterpart. The REST API still provided the smallest transfer size with 3384 bytes on average. Using Relay+GraphQL resulted in an increase, 5658 bytes (+67 %), and Falcor increased the total transfer size to 10859 bytes (+221 %).

The REST API used two network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

6.9 Full article + teasers + social data experiment

Figure 6.13 is showing the results for fetching the full text article in parallel with the teasers and then fetching the social data in sequence with the article data. The REST API showed a median response time of 227 ms, Relay+GraphQL showed a decrease, 135 ms (−37 %), and Falcor showed an increase, 344 ms (+47 %).

Figure 6.11. Latency of sequentially fetching the optimized article and social data using REST, Relay and Falcor.

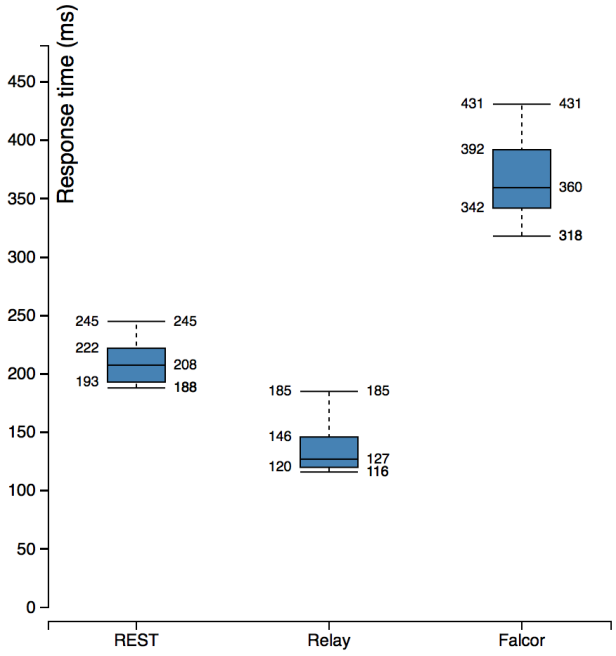


Figure 6.12. Transfer size of sequentially fetching the optimized article and social data using REST, Relay and Falcor.

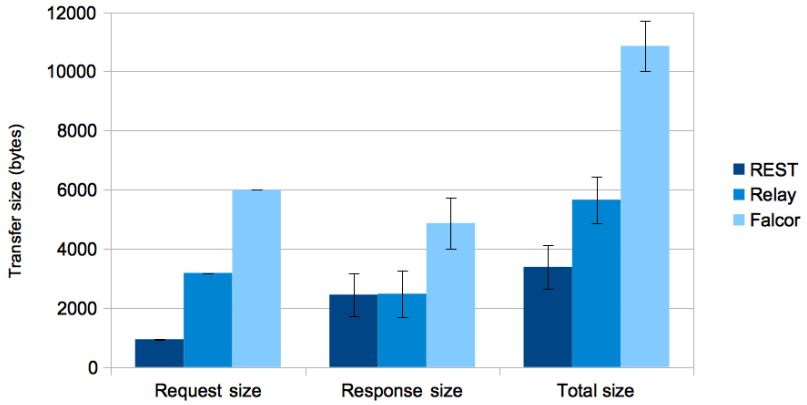
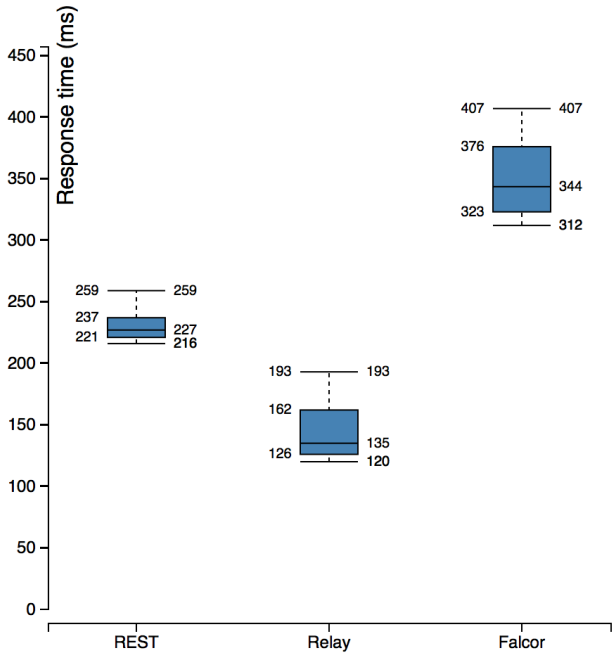


Figure 6.13. Latency of fetching the full article, teasers and social data using REST, Relay and Falcor.



The data transfer size is summarized in figure 6.14. The REST API showed the lowest transfer size in total, 8879 bytes, followed by Relay+GraphQL with an increase to 9601 bytes (+8 %), and Falcor with 16043 bytes (+81 %). However, the response size of Relay+GraphQL was the lowest, due to decreased HTTP overhead. While the HTTP overhead data was sent three times to the REST API, it was only sent once to the GraphQL endpoint. This is the same type of improvement seen in section 6.5, in the experiment where the full article and teasers were fetched.

The REST API used three network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

6.10 Optimized article + optimized teasers + social data experiment

Figure 6.15 is showing the optimized version of the full article, teasers and social data experiment. The REST API (Exact same as in the non-optimized experiment presented in section 6.9.) showed a median response time of 227 ms. When comparing Relay+GraphQL with the REST API, a decrease to 124 ms (−42 %) was recorded. Falcor showed a decreased performance when compared to the REST API, with an increased response time, 273 ms (+18 %).

Figure 6.14. Transfer size of fetching the full article, teasers and social data using REST, Relay and Falcor.

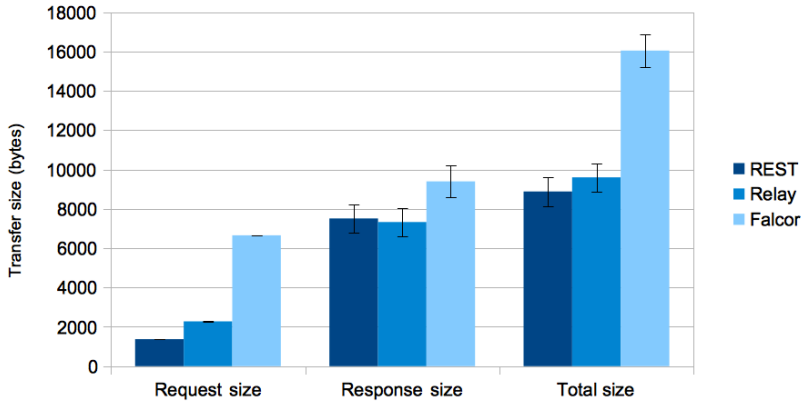
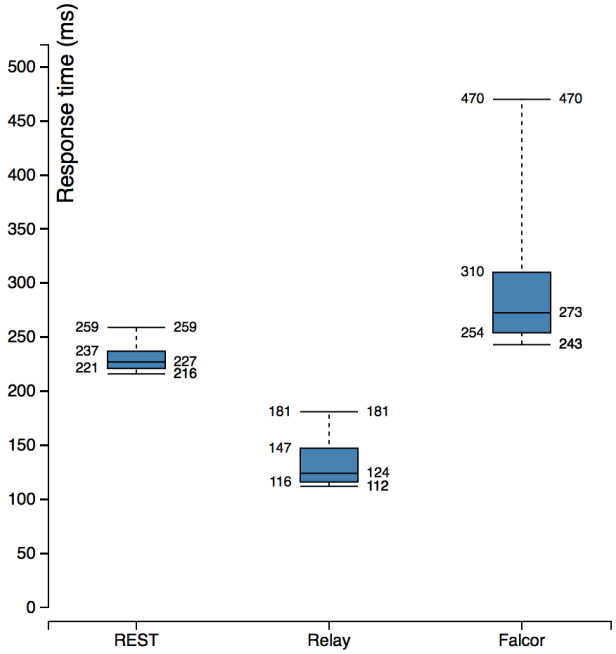
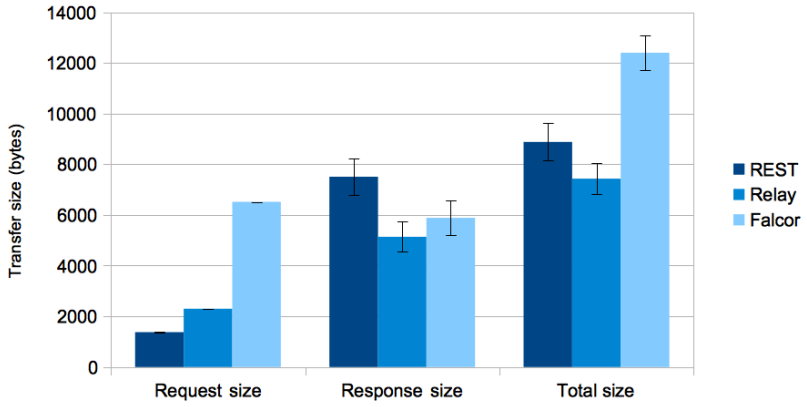


Figure 6.15. Latency of fetching the optimized article, optimized teasers and social data using REST, Relay and Falcor.



Just as for the other optimized experiment variants, the optimization efficiently decreased the response sizes of Relay+GraphQL and Falcor. This can be seen in figure 6.16. However, the large request size of Falcor made it the worst alternative for total transfer size, while Relay+GraphQL proved to be the best. In total, the REST API showed an an average transfer size of 8879 bytes, Relay+GraphQL showed a decrease, 7432 bytes (−16 %), and Falcor an increase to 12397 bytes (+40 %).

Figure 6.16. Transfer size of fetching the optimized article, optimized teasers and social data using REST, Relay and Falcor.



The REST API used three network requests to fetch the data, while Relay+GraphQL and Falcor only required one each.

6.11 Optimized components only experiment

The results from fetching the optimized components only using Relay+GraphQL and Falcor did not show any improvements in response time over the REST API. (See figure 6.17.) The REST API (Same as in the full article experiment presented in section 6.3.) had a median response time of 99 ms. Relay+GraphQL showed an increased response time, 109 ms (+9 %), while Falcor showed an increase to 188 ms (+77 %).

Figure 6.18 is also not showing any benefits of being able to fetch only the components of an article. While the response size can be decreased slightly by using Relay+GraphQL, the increased request size leads to a larger transfer size in total. The REST API used on average 2530 bytes, Relay+GraphQL showed an increase, 3487 bytes (+38 %) and Falcor increased the transfer size to 5761 bytes (+128 %).

All three data fetching methods required one network request to fetch the data.

Figure 6.17. Latency of fetching only the components of an article using REST, Relay and Falcor.

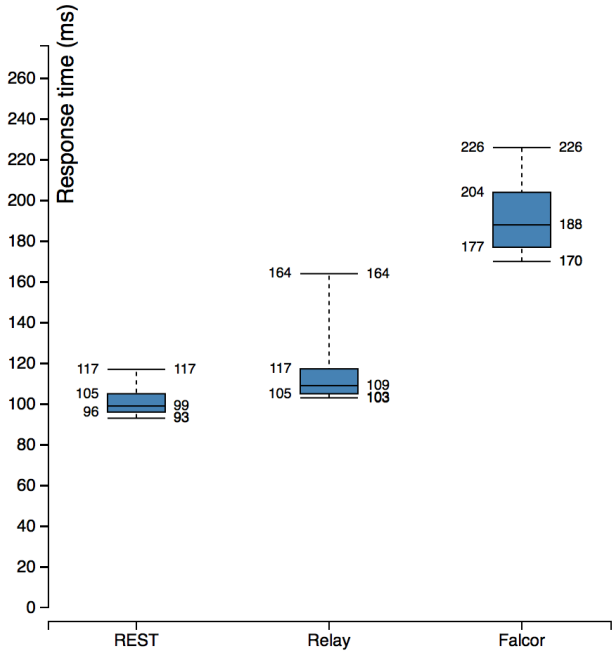
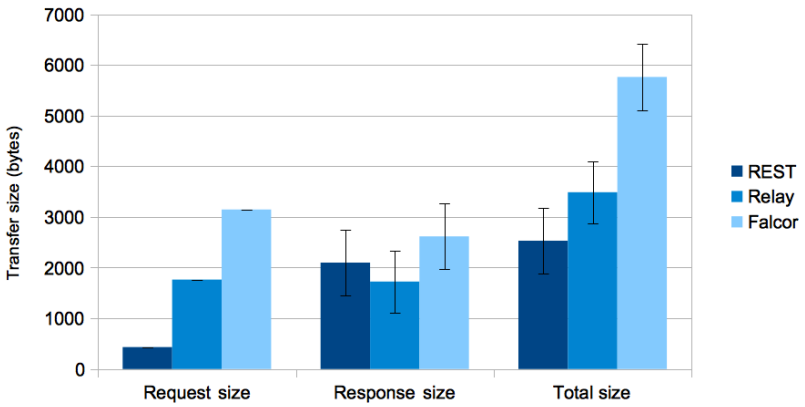


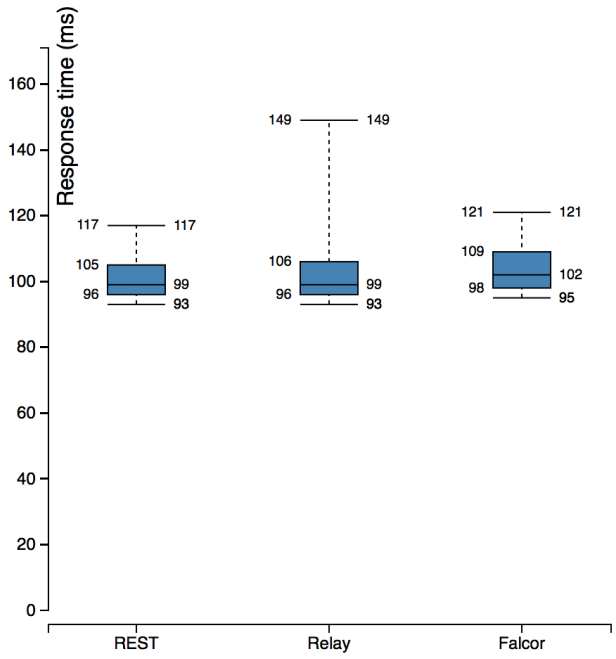
Figure 6.18. Transfer size of fetching only the components of an article using REST, Relay and Falcor.



6.12 Title only experiment

The results of the title only experiment presented in figure 6.19 revealed a similar performance for all three technologies. The REST API (Represented by the endpoint for fetching the full text article, presented in section 6.3.) showed a median response time of 99 ms. Relay+GraphQL showed the same response time, 99 ms (±0 %), and Falcor showed an increase to 102 ms (+3 %). Using Relay+GraphQL did not result in any significant change when compared to REST. Significance testing is discussed further in section 7.1.

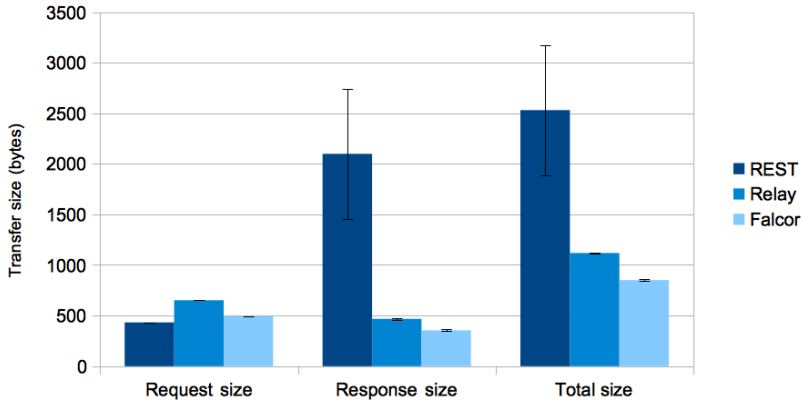
Figure 6.19. Latency of fetching only the title of an article using REST, Relay and Falcor.



The transfer sizes of the experiment are visualized in figure 6.20. In this experiment Falcor showed the smallest transfer size, due to both request and response sizes smaller than Relay+GraphQL. The REST API showed an average total transfer size of 2530 bytes, Relay+GraphQL showed a decrease, 1118 bytes (−56 %), and Falcor showed a decrease to 850 bytes (−66 %).

All three data fetching methods required one network request to fetch the data.

Figure 6.20. Transfer size of fetching only the title of an article using REST, Relay and Falcor.



Chapter 7

Discussion

In this chapter the results obtained from the experimental evaluation are discussed. First, the hypotheses are evaluated against the results followed by extended discussions about the latency and data volume results. Concerns about the Falcor implementation are then discussed before comments are made in regards to some of the community critique the frameworks received.

7.1 Testing the hypotheses

This section is dedicated to discussing the hypotheses in relation with the results. As a reminder, the null hypothesis stated there would not be any change to either latency, transfer size or the number of requests. The alternative hypothesis stated that there would be an increase instead of a decrease in respective measurement.

Because the collected material was not normally distributed, a weaker significance test was resorted to. A single sided sign test [62, pp. 337-339] was used to determine if the results showed significant changes. In a sign test, the values are compared pairwise and the sign of the difference between the values are noted. If the values are equal, the pair is omitted. Assuming there is no systematic difference between the data series, half of the signs should be positive and the other half negative. Thus, the result is an observation from the binomial distribution for the probability 1/2. By calculating the probability of repeating the results, the null hypothesis can be rejected if the probability is low.

For each test case two sign tests were performed. One test compared the results of using REST with the results of using Falcor, and one compared the results of using REST with the results of using Relay+GraphQL. In all test cases except the title only test case for Relay+GraphQL, statistically significant changes could be observed. To reject the null hypotheses, the significance level of 0.05 was used. The

full results of performing the sign tests for latency is shown in table A.7 and A.8. Executing the same tests for transfer sizes showed significant changes for all test cases.

The research hypotheses were the following:

- **Using Falcor will reduce the latency of data fetching in the studied client application.**

  The results showed that in none of the experiments, the latency was reduced using Falcor. As the results showed significant differences between Falcor and REST, the null hypothesis could be rejected in favor for the research or alternative hypotheses. Because the research hypothesis predicted a decrease in latency and the results showed an increase, the research hypothesis could also be rejected in favor of the alternative hypothesis. The alternative hypothesis predicted an increase in latency.

- **Using Falcor will reduce the data volume and the number of requests of data fetching in the studied client application.**

  All results showed statistically significant changes of total transfer size between the REST API and the Falcor implementation. This provided evidence that the null hypothesis can be rejected. However, for all but one experiment (the title only experiment) the total transfer size was increased. This suggested that the research hypothesis could be rejected in favor for the alternative hypothesis.

  In all experiments, Falcor made only one request regardless of how many requests to the REST API there were. This provided evidence in favor of the research hypothesis.

- **Using Relay+GraphQL will reduce the latency of data fetching in the studied client application.**

  When examining the results, it was observed that in all test cases but the title only experiment a significantly statistical change was present. This provided enough evidence to reject the null hypothesis. However, the evidence did not provide consensus whether the research hypothesis or the alternative hypothesis may be true; there existed statistically significant results providing evidence for both cases.

  The experiments that showed a decrease in response time were experiments corresponding to the combination of multiple REST API requests. This applied to fetching the article and teasers in parallel, the article and social data in sequence, the combination of the two and their optimized counterparts. All other experiments, corresponding to a single REST API request, resulted in increased response times.

- **Using Relay+GraphQL will reduce the data volume and the number of requests of data fetching in the studied client application.**

  The results showed significant changes in transfer size between the REST API and Relay+GraphQL implementation for all experiments. Therefore the null hypothesis could be rejected. Although, a majority of the test cases Relay+GraphQL showed higher transfer sizes than the REST API. This suggests that the alternative hypothesis is more likely to be true than the research hypothesis.

  In all experiments Relay+GraphQL made one request regardless of test case, providing evidence that the research hypothesis predicting a decrease in the number of requests may be true.

7.2 Latency

The results did not show any decrease in response time when using Falcor, but it did in certain cases when using Relay+GraphQL. These results were similar to what was found by Huang *et al.* [32], when they evaluated a similar solution for data fetching. The response time is mostly affected by the round trip time, while the transmission time is of a lesser significance when sending small text-based resources.

Huang *et al.* [32] also found that their server processing time was negligible and did not result in any significant overhead. This was not the case for either Falcor or Relay+GraphQL. Both frameworks introduced additional delays in comparison with the REST API which could be seen in, for example, the full text article experiment (figure 6.1). Even with the smaller query used in the title only experiment (see figure 6.19) the overhead was visible, yet close to zero.

The best improvement over the REST API was found when using Relay+GraphQL for fetching data that would require sequential requests to the REST API. This fits well with the results of Huang *et al.* [32], where they found the greatest improvements in response time for API requests with parameter dependencies. Parameter dependencies refers to when information from the response of a request is needed in a subsequent request. In this study, this was the case for fetching the social data.

Decreased response times when aggregating parallel requests were observed for Relay+GraphQL. This suggests that some overhead in the communication with the server was reduced, and the improvement is larger than the overhead introduced by the framework. However, any possible explanation is not available and parallel data flows was not studied by Huang *et al.* [32].

7.3 Transfer size

While Huang *et al.* [32] reported great improvements in transfer size when using their intermediary for aggregating API requests, this was rarely the case in this study. Only for a few test cases where aggressive filtering was applied, a decrease in transfer size was achieved. However, it is not clear how much the data was filtered in the work of Huang *et al.* [32]. Their original document sizes were larger than those used in this work and the use of compression could also have made the changes smaller in this work.

7.3.1 About request sizes

One insight from the experiments is about the importance of the request size for the total transfer size. When using the REST API, the request typically consists of a short URL and a few HTTP headers. With Relay+GraphQL, the URL remains short and the HTTP headers are nearly the same. However, the request is made using a HTTP POST with the query as payload. This means that only if the response size is reduced with more than the increase in request size, the total transfer size would be decreased. When handling quite small JSON documents and using compression for the responses, this was rarely the case.

7.4 Falcor implementation concerns

One of the major concerns with the results of the experiments was the increase in transfer size when using Falcor. An interesting observation was the amount of transfer size that was saved by using Gzip compression. In figure 7.1, it can be seen that the amount of bytes saved by compression is more than twentyfold for Falcor in comparison to REST and Relay+GraphQL.

Combining this with the transfer size (of figure 6.2) gives an original document size that is much larger for Falcor. Assuming the compression and decompression takes longer time for longer documents, this is one of the reasons behind the longer response times of Falcor. Also, it is assumed that repetitive data compress well and thus the resulting transfer size of Falcor is not too much larger than what was achieved with REST and Relay+GraphQL.

A suspicion for what caused this increase in response size is the empty atoms Falcor is sending when a value is requested, but does not exist. This was discussed in section 5.2.2 and is a consequence of how the solution for open ended queries was implemented. To find out how much this implementation affected the results, an experiment was conducted with the correct size of all lists predefined as of appendix A.4. Figure 7.2 is showing the results.

Figure 7.1. Transfer size saved by compression when fetching a full text article with REST, Relay and Falcor.

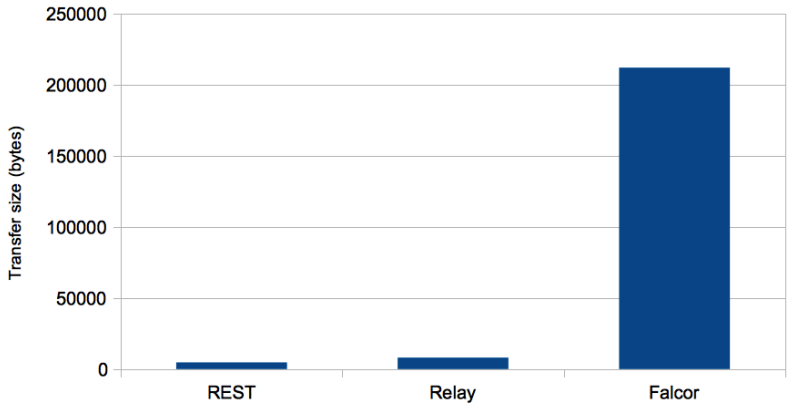
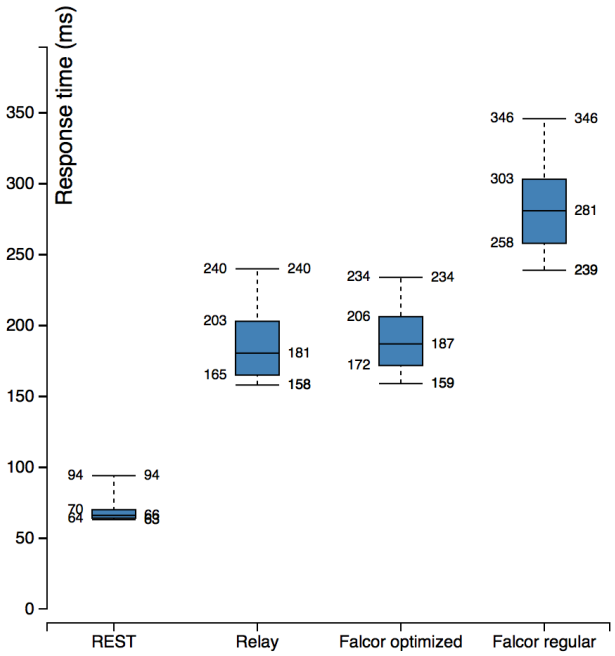


Figure 7.2. Latency of fetching a full text article with REST, Relay and Falcor (with and without optimized path).

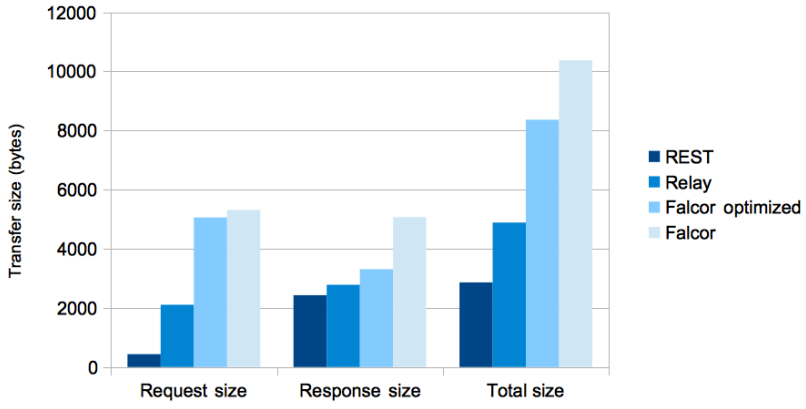


In this experiment, the article with id "22764268@abse" was fetched 100 times per data fetching method. A statistical significant difference between the response time of using an optimized and non-optimized Falcor path could be observed: a 99 ms decrease. The difference between Falcor and Relay+GraphQL is no longer of statistical significance.

This suggest that a fault in the experiment design and/or implementation caused the Falcor results to consistently show worse response times than Relay+GraphQL. However, it also means that the data used in the experiments (containing lists of unknown sizes) was not suitable for achieving good performance with Falcor while still exposing full filterability. Using atoms and removing the ability to filter data would likely have yielded performance similar to what was achieved in this experiment, but the comparison with Relay+GraphQL would be incomplete.

The transfer size of using the optimized Falcor path does still not beat REST or Relay+GraphQL. The results are shown in figure 7.3. While the transfer size was reduced by using an optimized path, and the transfer size saved by compression was reduced by almost 80 %, it was still considerably larger than the REST and Relay+GraphQL counterparts. The empty atoms are still a leading cause of the increased transfer size; all optional fields of the model are marked with empty atoms. Components of different types are not distinguished so every component contains all fields available in every type of component, resulting in a lot of empty fields. If Falcor implemented the abstraction of subclassing (available in GraphQL), this problem could potentially be eliminated.

Figure 7.3. Transfer size of fetching a full text article with REST, Relay and Falcor (with and without optimized path).



7.5 Community critique

There have been some community critique about features of the frameworks and also about implementation difficulties, some of which were briefly summarized in section 2.9.

Both Jones [36] and Chenkie [37] described their experiences from developing applications using Relay+GraphQL and Falcor. Their impression is that the frameworks greatly increase the amount of code needed to implement an application using the frameworks and that Relay introduce complex dependencies between React components.

In the experiment implementation, the amount of code needed was especially apparent for the Falcor router. Each route required a custom resolve function consisting of a lot of code that was similar, but not fully reusable between the routes. The Falcor server implementation consisted of approximately two times more lines of code than to the GraphQL implementation.

Also some peculiarities of Relay+GraphQL were encountered during the experiment implementation. For example, both the full text article and teasers could not be fetched by Relay when they were exposed directly in the Query-type. This was because the Relay root query could only have one child. As a workaround, an additional GraphQL object was created, wrapping the original Query-type (consisting of the full text article and teasers entry points.). This resulted in an additional level of depth in all queries and responses, contributing to an increased transfer size.

These implementation experiences suggests that some of the community critique may be justified.

7.6 Experiment data set analysis

For the experiments, a data set consisting of 50 articles, picked as a continuous selection from a list of the latest published articles was used. When analysing this data set in retrospect, it was concluded that the document sizes were on average smaller than the articles of a comparison data set. The comparison data set consisted of 500 articles, picked from the same list at a later time.

Table 7.1 is providing an overview of the document sizes of the data sets. The average article used in the experiments was approximately 25 % smaller than the average article in the comparison data set. The variation in size was also smaller, as can be seen by a lower standard deviation.

Using a smaller data set could affect the outcome of the transfer time measurements, as of equation 4.1. However, the difference in time spent during transmission would

Table 7.1. Document sizes of data sets used in experiments and for comparison. All measurements are given in bytes.

	Experiment data set	**Comparison data set**
Mean	4571	6156
Median	4538	6461
Stdev	2314	3406
Min	1681	1144
Max	12890	25994

be small as the transfer sizes were small and the bandwidth relatively high when executing the experiments.

Because all data fetching technologies are affected by the increased transfer sizes on the network, the comparison should still be valid in that regard. However, it is unknown whether an increased document size affected the computation times in any of the clients or servers.

Chapter 8

Conclusions and Future work

8.1 Summary

This thesis presented an evaluation of frameworks for declarative data fetching, Falcor and Relay+GraphQL. The main focus was on performance in terms of latency (response time) and network traffic, both in transfer size and number of requests. An implementation based on a real world news application was created for both frameworks and thereafter it was experimentally evaluated using a set of experiments. The results allowed for conclusions about performance of the frameworks in comparison with a corresponding REST API. Also conclusions about when it is beneficial to use the frameworks over a REST API was made.

8.2 Conclusions

The problem description in section 1.2 presented the following questions to be answered in this thesis:

- Which data fetching method, Falcor, Relay+GraphQL or REST APIs, will provide the fastest data fetching in terms of latency?
- Which data fetching method will cause the least amount of traffic over the network? Which will need the least number of requests and send the least amount of bytes?

Because the experimental results did not show any consensus for neither latency or transfer size, the questions can not be answered by naming a single technology. However, it was possible to identify during which circumstances what technologies performed best.

A rather obvious conclusion is that if it is possible to create a custom REST endpoint, this would be the most performant option both in terms of latency, transfer size and request count. Only when this is not feasible or benefits other than performance can be shown, declarative data fetching using the frameworks evaluated in this thesis should be considered.

The results showed no consensus whether Falcor, Relay+GraphQL or REST APIs provides the lowest latency. Falcor showed the highest latency in all experiments, with reservation for suboptimal design and implementation decisions. When comparing latency between Relay+GraphQL and the REST API, it was concluded that the REST APIs had lower latency when the data could be fetched with a single request to the REST API. When multiple requests were needed, regardless if they were executed in parallel or in sequence, Relay+GraphQL provided the lowest latency.

It was concluded that both Falcor and Relay+GraphQL could reduce the number of requests for parallel and sequential data flows and for single requests all technologies performed the same.

All experiments showed an increased transfer size for Falcor compared to the REST API while Relay+GraphQL showed an increased transfer size for a majority of the experiments. The case where transfer size could be decreased by using Relay+GraphQL was when the data from the REST API could be reduced with more than the increase in request size introduced by the framework. This highlighted one of the key points not considered in the initial motivation where only decreased response sizes were considered: the request size and response size are of equal importance. Often using Relay+GraphQL could decrease the response size by allowing data filtering, but this reduction was negated by sending a large query in the request.

8.3 Future work

This study was performed using experiments based on a real world news application. However, the test cases were still fabricated and not representing the production application to 100 %. Future work could present a case study from an actual migration and provide data from usage of the application by real users. It would also be interesting to study how the frameworks performed in combination with server side HTTP caching, which is commonly deployed in production applications.

Future work could also be more mobile focused. In this study, latencies were established using a wireless network (802.1X), but over a mobile network the proportions of latency and bandwidth are different and thus different results could be achieved. If the bandwidth is lowered, the transfer size has more impact on the total transfer

time. If the latency is changed, the penalty for sequential fetching may be different.

Another proposal for future work is to find a better solution for the problem with open ended queries in Falcor. In an improved version, paths could contain nested paths that are evaluated on the server side. Returning to the list of components of unknown size that was fetched in this project, a possible query could look like:

["article", "22764268@abse", "components", {"from": 0, "to":
["article", "22764268@abse", "components", "length"]}, "type"]

The end of the index range to fetch is given by a path (marked with bold text) returning the length of the list. If this sub-path could be evaluated on the server side before the rest of the path, a sequential flow can be avoided. This is not supported at the time of writing and the current solution would be to fetch the length of the list on the client, create the path with the correct range and then sequentially fetch the whole list.

Another proposed future improvement to Falcor that may be suitable is to implement a default path following mechanism similar to GraphQL's solution. In GraphQL, if no custom resolve function is defined it will resolve each field by using the JSON path of the field name on the parent object.

Because both Falcor, Relay and GraphQL are open source, anyone can develop them further. Also, a deeper analysis with insights from the source code is possible.

References

[1] I. Grigorik, *High Performance Browser Networking: WHAT every web developer should know about networking and web performance.* " O'Reilly Media, Inc.", 2013.

[2] R. T. Fielding, "Architectural styles and the design of network-based software architectures", University of California, Irvine, 2000.

[3] D. Jacobson *et al.*, *APIS: A strategy guide.* " O'Reilly Media, Inc.", 2011.

[4] H. Hamad *et al.*, "Performance Evaluation of RESTful Web Services for Mobile Devices.", *Int. Arab J. e-Technol.*, vol. 1, no. 3, pp. 72–78, 2010.

[5] D. Jacobson. (2012). Why REST Keeps Me Up At Night, ProgrammableWeb, [Online]. Available: `http://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15` [Accessed: 2nd Feb. 2016].

[6] @Scale. (2015). Building and Deploying Relay with Facebook, [Online]. Available: `https://www.youtube.com/watch?v=Pxdgu2XIAAg` [Accessed: 4th Feb. 2016].

[7] Schibsted. (n.d.). Schibsted Sverige - Schibsted, [Online]. Available: `http://www.schibsted.com/en/About-Schibsted/Schibsted-Sweden/` [Accessed: 4th Feb. 2016].

[8] KIA-Index and S. Annonsörer. (n.d.). KIA-index, KIA-index, [Online]. Available: `http://www.kiaindex.se/sok/?site_name=aftonbladet` [Accessed: 10th Feb. 2016].

[9] Netflix. (2015). Falcor: One Model Everywhere, Falcor, [Online]. Available: `https://netflix.github.io/falcor/index.html` [Accessed: 26th Dec. 2015].

[10] Facebook. (2015). Relay | A JavaScript framework for building data-driven React applications, [Online]. Available: `https://facebook.github.io/relay/` [Accessed: 26th Dec. 2015].

[11] Facebook. (2015). GraphQL, [Online]. Available: `http://facebook.github.io/graphql/` [Accessed: 12th Feb. 2016].

[12] G. F. Coulouris *et al.*, *Distributed systems: Concepts and design*. Pearson Education, 2005.

[13] C. Peltz, "Web services orchestration and choreography", *Computer*, vol. 36, no. 10, pp. 46–52, 2003, ISSN: 0018-9162. DOI: 10.1109/MC.2003.1236471.

[14] J. Yu *et al.*, "Understanding Mashup Development", *IEEE Internet Computing*, vol. 12, no. 5, pp. 44–52, 2008, ISSN: 1089-7801. DOI: 10.1109/MIC.2008.114.

[15] Oracle. (2010). Java Object Serialization Specification: Contents, [Online]. Available: https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html [Accessed: 11th Feb. 2016].

[16] Google. (2016). Protocol Buffers, Google Developers, [Online]. Available: https://developers.google.com/protocol-buffers/ [Accessed: 11th Feb. 2016].

[17] W3C. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition), [Online]. Available: https://www.w3.org/TR/REC-xml/ [Accessed: 11th Feb. 2016].

[18] T. Bray. (2014). The JavaScript Object Notation (JSON) Data Interchange Format, [Online]. Available: https://tools.ietf.org/html/rfc7159 [Accessed: 11th Feb. 2016].

[19] B. Gil and P. Trezentos, "Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones", in *Proceedings of the 2011 Workshop on Open Source and Design of Communication*, ser. OSDOC '11, New York, NY, USA: ACM, 2011, pp. 1–6, ISBN: 978-1-4503-0873-1. DOI: 10.1145/2016716.2016718.

[20] L. Richardson *et al.*, *RESTFUL WEB APIS*. " O'Reilly Media, Inc.", 2013.

[21] Netflix. (2015). Falcor: What is Falcor, Falcor, [Online]. Available: https://netflix.github.io/falcor/starter/what-is-falcor.html [Accessed: 18th Feb. 2016].

[22] Netflix. (2015). Falcor: Paths, Falcor, [Online]. Available: https://netflix.github.io/falcor/documentation/paths.html [Accessed: 19th Feb. 2016].

[23] Netflix. (2015). Falcor: JSON Graph, Falcor, [Online]. Available: https://netflix.github.io/falcor/documentation/jsongraph.html [Accessed: 19th Feb. 2016].

[24] Netflix. (2015). Falcor: How Does Falcor Work?, Falcor, [Online]. Available: https://netflix.github.io/falcor/starter/how-does-falcor-work.html [Accessed: 18th Feb. 2016].

[25] Facebook. (2015). Type System |GraphQL, [Online]. Available: http://graphql.org/docs/typesystem/ [Accessed: 23rd Feb. 2016].

[26] Facebook. (2015). Queries | GraphQL, [Online]. Available: http://graphql.org/docs/queries/ [Accessed: 23rd Feb. 2016].

[27] Facebook. (2016). A JavaScript library for building user interfaces | React, [Online]. Available: https://facebook.github.io/react/index.html [Accessed: 19th Feb. 2016].

[28] Facebook. (2015). Thinking In Relay | Relay Docs, [Online]. Available: http://facebook.github.io/relay/docs/thinking-in-relay.html [Accessed: 22nd Feb. 2016].

[29] Facebook. (2015). Containers | Relay Docs, [Online]. Available: http://facebook.github.io/relay/docs/guides-containers.html [Accessed: 22nd Feb. 2016].

[30] Facebook. (2015). Thinking in GraphQL | Relay Docs, [Online]. Available: http://facebook.github.io/relay/docs/thinking-in-graphql.html [Accessed: 22nd Feb. 2016].

[31] B. Zhao *et al.*, "Reducing the Delay and Power Consumption of Web Browsing on Smartphones in 3G Networks", in *2011 31st International Conference on Distributed Computing Systems (ICDCS)*, 2011, pp. 413–422. DOI: 10.1109/ICDCS.2011.54.

[32] C.-C. Huang *et al.*, "Energy-efficient and cost-effective web API invocations with transfer size reduction for mobile mashup applications", *Wireless Networks*, vol. 20, no. 3, pp. 361–378, 2013, ISSN: 1022-0038, 1572-8196. DOI: 10.1007/s11276-013-0608-7.

[33] A. Susiripala. (2015). Initial Impressions on GraphQL & Relay, Kadira Blog, [Online]. Available: https://kadira.io/blog/graphql/initial-impression-on-relay-and-graphql [Accessed: 10th Mar. 2016].

[34] Meteor Development Group. (n.d.). Meteor, [Online]. Available: https://www.meteor.com/ [Accessed: 10th Mar. 2016].

[35] M. Faassen. (2015). GraphQL and REST, Secret Weblog, [Online]. Available: http://blog.startifact.com/posts/graphql-and-rest.html [Accessed: 10th Mar. 2016].

[36] T. Jones. (2015). Why I Quit Facebook Relay (for now), Medium, [Online]. Available: https://medium.com/@OverclockedTim/why-i-quit-facebook-relay-eeab0177f92f#.ueq7d6dk7 [Accessed: 10th Mar. 2016].

[37] R. Chenkie. (2016). Rise of the High Boilerplate Framework: A Look at Falcor and Relay, Auth0 - Blog, [Online]. Available: https://auth0.com/blog/2016/01/13/rise-of-the-high-boilerplate-framework-a-look-at-falcor-and-relay/ [Accessed: 10th Mar. 2016].

[38] C. Corcos. (2015). What's the Point of GraphQL and Falcor? Stateless applications and declarative resource requests, Medium, [Online]. Available: https:

//medium.com/@chetcorcos/what-s-the-point-of-graphql-and-falcor-cdd0f35960c0#.2unz7tbfk [Accessed: 10th Mar. 2016].

[39] C. Crawford. (2015). Facebook's Relay: A JavaScript Framework That Gives React a Larger Scope, The New Stack, [Online]. Available: http://thenewstack.io/facebooks-relay-javascript-framework-building-react-applications/ [Accessed: 10th Mar. 2016].

[40] D. Ary *et al.*, *Introduction to research in education*. Cengage Learning, 2013.

[41] Google. (n.d.). Google Scholar, [Online]. Available: https://scholar.google.se/ [Accessed: 4th Mar. 2016].

[42] KTH Library. (n.d.). KTHB Primo, [Online]. Available: http://kth-primo.hosted.exlibrisgroup.com/primo_library/libweb/action/search.do?vid=46KTH_VU1&prefLang=sv_SE [Accessed: 4th Mar. 2016].

[43] IEEE. (2016). IEEE Xplore Digital Library, [Online]. Available: http://ieeexplore.ieee.org/Xplore/home.jsp [Accessed: 4th Mar. 2016].

[44] Elsevier B.V. (2016). ScienceDirect.com | Science, health and medical journals, full text articles and books., [Online]. Available: http://www.sciencedirect.com/ [Accessed: 8th Apr. 2016].

[45] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects", presented at the The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July, CSREA Press U.S.A, 2013, pp. 67–73.

[46] Free Software Foundation. (n.d.). GNU Gzip, [Online]. Available: https://www.gnu.org/software/gzip/ [Accessed: 18th Apr. 2016].

[47] H. Adams. (2004). Best Practices for web services: Part 9, [Online]. Available: http://www.ibm.com/developerworks/library/ws-best9/#webserviceperformancebottlenecks [Accessed: 11th Mar. 2016].

[48] W. Zhou *et al.*, "REST API Design Patterns for SDN Northbound API", in *2014 28th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2014, pp. 358–365. DOI: 10.1109/WAINA.2014.153.

[49] L. Li *et al.*, "A REST Service Framework for Fine-Grained Resource Management in Container-Based Cloud", in *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, 2015, pp. 645–652. DOI: 10.1109/CLOUD.2015.91.

[50] AngularAir. (2015). Angular Air Episode 26: FalcorJS and Angular 2, [Online]. Available: https://www.youtube.com/watch?v=WL54eYbTJUw&feature=youtu.be&t=53m55s [Accessed: 31st Mar. 2016].

[51] GitHub. (2016). Chentsulin/awesome-graphql, GitHub, [Online]. Available: https://github.com/chentsulin/awesome-graphql [Accessed: 7th Mar. 2016].

[52] R. T. Fielding. (2008). REST APIs must be hypertext-driven - Untangled, [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [Accessed: 30th Mar. 2016].

[53] Wireshark Foundation. (n.d.). Wireshark - Go Deep., [Online]. Available: https://www.wireshark.org/ [Accessed: 13th Apr. 2016].

[54] W3C. (2012). HTTP Archive (HAR) format, [Online]. Available: https://w3c.github.io/web-performance/specs/HAR/Overview.html [Accessed: 5th May 2016].

[55] GitHub. (2015). Model get is resolving as undefined - Issue #583 - Netflix/falcor, GitHub, [Online]. Available: https://github.com/Netflix/falcor/issues/583 [Accessed: 3rd May 2016].

[56] GitHub. (2016). Graphql/graphql-js, GitHub, [Online]. Available: https://github.com/graphql/graphql-js [Accessed: 8th Apr. 2016].

[57] GitHub. (2016). Reactjs/redux, GitHub, [Online]. Available: https://github.com/reactjs/redux [Accessed: 2nd May 2016].

[58] GitHub. (2015). Relay and Redux - Issue #464 - reactjs/redux, GitHub, [Online]. Available: https://github.com/reactjs/redux/issues/464 [Accessed: 2nd May 2016].

[59] GitHub. (2015). Has Redux's Time Come and Gone - Issue #775 - reactjs/redux, GitHub, [Online]. Available: https://github.com/reactjs/redux/issues/775 [Accessed: 2nd May 2016].

[60] Facebook. (2016). Component Specs and Lifecycle | React, [Online]. Available: https://facebook.github.io/react/docs/component-specs.html [Accessed: 3rd May 2016].

[61] Salesforce. (2016). Cloud Application Platform | Heroku, [Online]. Available: https://www.heroku.com/home [Accessed: 9th May 2016].

[62] G. Blom *et al.*, *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 2005, ISBN: 91-44-02442-4.

Appendix A

Experiment specification

A.1 Article identifiers

The following article identifiers were used in the experiments. Note that they may not be available if the editorial staff at Aftonbladet unpublished the articles.

```
"22749731@abse",
"22764268@abse",
"22764205@abse",
"22764082@abse",
"22764128@abse",
"22764136@abse",
"22764172@abse",
"22767989@abse",
"22765990@abse",
"22767913@abse",
"22767770@abse",
"22765295@abse",
"22767598@abse",
"22767701@abse",
"22767064@abse",
"22767971@abse",
"22767911@abse",
"22767432@abse",
"22757289@abse",
"22767876@abse",
"22767605@abse",
"22767552@abse",
"22767824@abse",
"22767819@abse",
"22767791@abse",
"22767783@abse",
"22765719@abse",
```

```
28 "22767742@abse",
29 "22767512@abse",
30 "22767495@abse",
31 "22767590@abse",
32 "22767588@abse",
33 "22767513@abse",
34 "22767487@abse",
35 "22767481@abse",
36 "22767394@abse",
37 "22767365@abse",
38 "22767444@abse",
39 "22767407@abse",
40 "22767312@abse",
41 "22764762@abse",
42 "22767203@abse",
43 "22767342@abse",
44 "22767316@abse",
45 "22767302@abse",
46 "22766370@abse",
47 "22766361@abse",
48 "22766362@abse",
49 "22766323@abse",
50 "22767292@abse"
```

A.2 NPM packages and versions

The following NPM packages were used in the experiment implementation.

```
"compression": "^1.6.1", //For adding gzip to the server.
"cors": "^2.7.1", //For allowing cross-domain-requests.
"express": "4.13.4", //The http server
"express-graphql": "0.4.9", //The GraphQL integration with express.
"falcor": "^0.1.16", //Falcor frontend library
"falcor-express": "^0.1.2", //Falcor express integration
"falcor-http-datasource": "^0.1.3", //Falcor HTTP integration
"falcor-json-graph": "^1.1.7", //For building responses on the JSON graph format.
"falcor-router": "^0.4.0", //The Falcor router
"fs-web": "^1.0.1", //File system abstraction for web browsers
"graphql": "0.4.17", //GraphQL server library
"graphql-relay": "0.3.6", //Relay integration for GraphQL.
"isomorphic-fetch": "^2.2.1", //Used in client to fetch data from REST endpoints.
"react": "0.14.7", //React, framework for creating web apps.
"react-dom": "0.14.7", //Helper for React
"react-relay": "0.7.3", //The Relay client library.
"react-router": "^2.3.0", //Library for multi page applications with React.
"request-promise": "^2.0.1", //Used by servers for fetching data from data sources.
"webpack": "1.12.13", //Build system for client application.
"webpack-dev-server": "1.14.1" //Web server for client application.
```

A.3 Falcor index configuration

The following indices were used for lists in the Falcor experiment. In the implementation, paths were written with placeholders for keys, indices and other parameters. The test runner replaced the placeholders with the values specified in the code below.

```
path
  .replace("{keys:articleId}", "'" + articleId + "'")
  .replace("{integers:tagIndices}", "0..9")
  .replace("{integers:componentIndices}", "0..49")
  .replace("{integers:urlIndices}", "0..14")
  .replace("{integers:contactIndices}", "0..4")
  .replace("{integers:authorIndices}", "0..4")
  .replace("{integers:teaserIndices}", "0..9")
  .replace("{integers:channelIndices}", "0..4")
  .replace("{keys:imageWidths}", 500);
```

A.4 Falcor configuration with exact indices

The following indices were used for fetching lists of known length for the article with id "22764268@abse".

```
path.
  .replace("{keys:articleId}", "'" + articleId + "'")
  .replace("{integers:tagIndices}", "0..2")
  .replace("{integers:componentIndices}", "0..9")
  .replace("{integers:urlIndices}", "0..14")
  .replace("{integers:contactIndices}", "0")
  .replace("{integers:authorIndices}", "0")
  .replace("{integers:teaserIndices}", "0..9")
  .replace("{integers:channelIndices}", "0..4")
  .replace("{keys:imageWidths}", 500);
```

A.5 68-95-99,7 analysis

Table A.1, A.2 and A.3 are showing the standard deviation and how large portion of the observed latency readings were within one, two and three standard deviations of the mean.

A.6 Full transfer size results

Table A.4, A.5 and A.6 are showing the full transfer size results for all experiments.

Table A.1. Standard deviation and share of latency readings within 1, 2 and 3 standard deviations of the mean for REST.

	σ	$< \sigma$	$< 2\sigma$	$< 3\sigma$
Full article	17	0,928	0,956	0,966
Optimized article	17	0,928	0,956	0,966
Full article + teasers	26	0,824	0,964	0,998
Optimized article + optimized teasers	26	0,824	0,964	0,998
Full article + social data	31	0,9	0,968	0,99
Optimized article + social data	31	0,9	0,968	0,99
Full article + teasers + social data	23	0,884	0,964	0,98
Optimized article + optimized teasers + social data	23	0,884	0,964	0,98
Optimized components only	17	0,928	0,956	0,966
Title only	17	0,928	0,956	0,966

Table A.2. Standard deviation and share of latency readings within 1, 2 and 3 standard deviations of the mean for Falcor.

	σ	$< \sigma$	$< 2\sigma$	$< 3\sigma$
Full article	40	0,758	0,956	0,994
Optimized article	35	0,844	0,966	0,99
Full article + teasers	44	0,782	0,968	0,992
Optimized article + optimized teasers	32	0,846	0,968	0,992
Full article + social data	55	0,864	0,976	0,988
Optimized article + social data	51	0,784	0,97	0,98
Full article + teasers + social data	61	0,878	0,986	0,992
Optimized article + optimized teasers + social data	99	0,858	0,946	0,984
Optimized components only	25	0,808	0,968	0,986
Title only	22	0,948	0,972	0,984

Table A.3. Standard deviation and share of latency readings within 1, 2 and 3 standard deviations of the mean for Relay+GraphQL.

	σ	$< \sigma$	$< 2\sigma$	$< 3\sigma$
Full article	29	0,834	0,952	0,99
Optimized article	26	0,83	0,946	0,984
Full article + teasers	35	0,888	0,964	0,99
Optimized article + optimized teasers	30	0,83	0,958	0,984
Full article + social data	29	0,796	0,952	0,982
Optimized article + social data	27	0,814	0,942	0,984
Full article + teasers + social data	31	0,798	0,956	0,99
Optimized article + optimized teasers + social data	45	0,914	0,976	0,988
Optimized components only	27	0,854	0,946	0,974
Title only	25	0,852	0,942	0,98

Table A.4. Full transfer size results for REST, average, in bytes.

	Request size	Response size	Total size
Full article	431,954	2097,812	2529,766
Optimized article	431,954	2097,812	2529,766
Full article + teasers	867	7158,56	8025,56
Optimized article + optimized teasers	867	7158,56	8025,56
Full article + social data	937,22	2447,18	3384,4
Optimized article + social data	937,22	2447,18	3384,4
Full article + teasers + social data	1372,524	7506,42	8878,944
Optimized article + optimized teasers + social data	1372,524	7506,42	8878,944
Optimized components only	431,954	2097,812	2529,766
Title only	431,954	2097,812	2529,766

Table A.5. Full transfer size results for Relay+GraphQL, average, in bytes.

	Request size	Response size	Total size
Full article	2103,74	2437,51	4541,25
Optimized article	2124,74	2271,39	4396,13
Full article + teasers	2242,74	7298,36	9541,1
Optimized article + optimized teasers	2263,74	5109,45	7373,19
Full article + social data	2136,74	2463,35	4600,09
Optimized article + social data	3180,48	2477,75	5658,23
Full article + teasers + social data	2275,74	7324,97	9600,71
Optimized article + optimized teasers + social data	2296,74	5135,71	7432,45
Optimized components only	1762,74	1724,65	3487,39
Title only	650,74	466,98	1117,72

Table A.6. Full transfer size results for Falcor, average, in bytes.

	Request size	Response size	Total size
Full article	5308	4669,73	9977,73
Optimized article	5206,052	3218,57	8424,622
Full article + teasers	6542	9373,11	15915,11
Optimized article + optimized teasers	6406,052	5864,23	12270,282
Full article + social data	5416	4689,91	10105,91
Optimized article + social data	5991,052	4867,87	10858,922
Full article + teasers + social data	6650	9392,61	16042,61
Optimized article + optimized teasers + social data	6514	5883,07	12397,07
Optimized components only	3145,052	2615,7	5760,752
Title only	496	353,93	849,93

A.7 Significance testing

Table A.7 and A.8 are showing the full results of the sign tests for latency of Falcor and Relay+GraphQL.

Table A.7. Results of the sign test for latency, comparing REST and Falcor.

	p
Full article	0,000
Optimized article	0,000
Full article + teasers	0,000
Optimized article + optimized teasers	0,000
Full article + social data	0,000
Optimized article + social data	0,000
Full article + teasers + social data	0,000
Optimized article + optimized teasers + social data	0,000
Optimized components only	0,000
Title only	0,000

Table A.8. Results of the sign test for latency, comparing REST and Relay+GraphQL.

	p
Full article	0,000
Optimized article	0,000
Full article + teasers	0,000
Optimized article + optimized teasers	0,000
Full article + social data	0,000
Optimized article + social data	0,000
Full article + teasers + social data	0,000
Optimized article + optimized teasers + social data	0,000
Optimized components only	0,000
Title only	0,391

TRITA -ICT-EX-2016:92