Client-Server Communications Efficiency in GIS/NIS Applications

An evaluation of communications protocols and serialization formats

KASHMIR KLINGESTEDT
Client-Server
Communications Efficiency in
GIS/NIS Applications

An evaluation of communications
protocols and serialization formats

KASHMIR KLINGESTEDT

Master in Computer Science
Date: August 16, 2018
Supervisor: Håkan Lane (KTH), Robert Olofsson (Digpro)
Examiner: Elena Troubitsyna
Swedish title: Kommunikationseffektivitet mellan klient och server i
GIS/NIS-applikationer
School of Electrical Engineering and Computer Science
Abstract

Geographic Information Systems and Network Information Systems are important tools for our society, used for handling geographic spatial data and large information networks. It is therefore important to make sure such tools are of high quality. GIS/NIS applications typically deal with a lot of data, possibly resulting in heavy loads of network traffic. This work aims to evaluate two different communications protocols and serialization formats for client-server communications efficiency in GIS/NIS applications. Specifically, these are HTTP/1.1, HTTP/2, Java Object Serialization and Google’s Protocol Buffers. They were each implemented directly into a commercial GIS/NIS environment and evaluated by measuring two signature server calls in the system. Metrics that were examined are call duration, HTTP overhead size and HTTP payload size. The results suggest that HTTP/2 and Google’s Protocol Buffers outperform HTTP/1.1 and Java Object Serialization respectively. An 87% decrease in HTTP overhead size was achieved when switching from HTTP/1.1 to HTTP/2. The HTTP payload size is also shown to decrease with the use of Protocol Buffers rather than Java Object Serialization, especially for communications where data consist of many different object types. Concerning call duration, the results suggest that the choice of communications protocol is more significant than the choice of serialization format for communications containing little data, while the opposite is true for communications containing much data.
Sammanfattning

Contents

1 Introduction 1
  1.1 Problem Statement .................................. 2
  1.2 Scope .............................................. 2
  1.3 Purpose ............................................ 2

2 Background 4
  2.1 Geographic Information Systems ................. 4
  2.2 Network Information Systems ..................... 4
  2.3 The dpSpatial Base Platform ..................... 5
  2.4 The Hypertext Transfer Protocol ................. 6
    2.4.1 HTTP/1.1 ..................................... 6
    2.4.2 HTTP/2 ..................................... 7
  2.5 Data Serialization ................................ 7
    2.5.1 Java Object Serialization ....................... 8
    2.5.2 Protocol Buffers ................................ 8
  2.6 Related Work .................................. 9
    2.6.1 Are HTTP/2 Servers Ready Yet? ............... 9
    2.6.2 Is HTTP/2 Really Faster Than HTTP/1.1? ....... 9
    2.6.3 Is the Web HTTP/2 Yet? ...................... 10
    2.6.4 Native Web Communication Protocols and Their
          Effects on the Performance of Web Services and
          Systems ........................................ 11
    2.6.5 A Comparison of Data Serialization Formats For
          Optimal Efficiency on a Mobile Platform .......... 11
    2.6.6 Performance Evaluation of Object Serialization
          Libraries in XML, JSON and Binary Formats .... 12

3 Methodology 13
  3.1 Selected Server Calls ............................ 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>GetGeometryComponents</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2</td>
<td>LockObject</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementations</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1</td>
<td>HTTP Client</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2</td>
<td>ProtoBuf</td>
<td>16</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Server Configurations</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Metrics</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Java Microbenchmark Harness</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Wireshark</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Results</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Call Duration</td>
<td>19</td>
</tr>
<tr>
<td>4.1.1</td>
<td>LockObject</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2</td>
<td>GetGeometryComponents</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>HTTP Overhead Size</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>HTTP Payload Size</td>
<td>23</td>
</tr>
<tr>
<td>4.3.1</td>
<td>LockObject</td>
<td>23</td>
</tr>
<tr>
<td>4.3.2</td>
<td>GetGeometryComponents</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Discussion</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Call Duration</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2</td>
<td>HTTP/1.1 vs. HTTP/2</td>
<td>26</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Java Object Serialization vs. ProtoBuf</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Sources of Error</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>The Bigger Picture</td>
<td>28</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Ethics and Sustainability</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Future Research</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>32</td>
</tr>
<tr>
<td>A</td>
<td>Setup of a client-server connection in dpSpatial using a URL-Connection</td>
<td>34</td>
</tr>
<tr>
<td>B</td>
<td>LockObject.proto</td>
<td>35</td>
</tr>
<tr>
<td>C</td>
<td>GetGeometryComponents.proto</td>
<td>36</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Important decision making and planning in our society rely on the availability and analysis of information. It is therefore important to find ways to store and manage different kinds of information efficiently. Geographic Information Systems (GIS) and Network Information Systems (NIS) are systems made for handling geographic spatial data and large information networks. Such systems make it possible to store, manipulate, analyze and visualize data, and they can be combined to create powerful tools.

The applications and uses of GIS and NIS are manifold. Astronomy, economics, environment, urban planning, telecommunications, transportation and weather are just a few examples of areas where such systems are applicable [9]. As such, the development and potentiation of GIS/NIS applications is meaningful for our society as a whole.

GIS/NIS applications typically deal with a lot of data. In a client-server perspective, this means a lot of data is being transmitted back and forth. In order to make such applications of high quality, it is therefore important to make the client-server communications as efficient as possible. A key to accomplish this is the choice of communications protocol and data serialization format to be used by the application.

This thesis aims to evaluate two popular communications protocols and serialization formats in how they affect the client-server communications efficiency in GIS/NIS applications. The concerned protocols
are HTTP/1.1 and HTTP/2, and the serialization formats are Java Object Serialization and Google’s Protocol Buffers. Evaluation is done by implementing each protocol and format into an existing commercial GIS/NIS application and measuring call duration, HTTP overhead size and HTTP payload size of two signature server calls.

1.1 Problem Statement

How does the choice of communications protocol and serialization format affect the client-server communications efficiency in GIS/NIS applications, and which solution is the most suitable?

1.2 Scope

Many different communications protocols and serialization formats exist, but due to time limitations of this project solely two communications protocols and two serialization formats were selected for evaluation. These are HTTP/1.1, HTTP/2, Java Object Serialization and Google’s Protocol Buffers.

Even though the different protocols and formats are implemented directly into the source code of an existing commercial GIS/NIS application, no finished product is produced by this work. The focus here lies on evaluation, however the results may influence the future development of the involved application.

1.3 Purpose

This work is relevant for anyone dealing with GIS/NIS applications. Such systems have existed for decades and are, as mentioned earlier, used in many different areas of our society. They are valuable tools for managing geographic networks of any kind. In many cases, such as with large networks, it would barely be possible to manage to maintain all data or to make sound decisions without them. As such, they are vital tools for our society to function efficiently and it is therefore important to keep developing and improving such systems to enhance
their functionality and quality, which is what this thesis aims to contribute to.
Chapter 2

Background

2.1 Geographic Information Systems

A Geographic Information System (GIS) is a system designed to handle geographic spatial data. Spatial data represent objects that have physical dimensions and take up space, and geographic spatial data occur on, in or above the planet Earth [16]. A GIS typically allows one to capture, store, manipulate, analyze, manage, and visualize such data. In the beginning, such systems were in the form of static paper maps, sometimes with a layered approach in an attempt to make them less static. Today, most GIS’s are in the form of dynamic digital maps, making operations much less cumbersome. The development of GIS’s continues as we go from fundamental analysis to more complex problem solving [10].

2.2 Network Information Systems

A Network Information System (NIS) is a system for managing networks, such as water supply- or telecommunications networks. Relevant data, attributes, and specific operations vary between different kinds of networks. For instance, a NIS for electricity networks could make it possible to modify voltage levels in a network, while this is not a relevant functionality for a water supply network. A NIS can also be built on top of a GIS to add a geographic context to the network.
2.3 The dpSpatial Base Platform

The dpSpatial base platform is a system developed by Digpro, which is the core of several GIS/NIS applications developed by the same company. An illustration of dpSpatial can be seen in Figure 2.1. It uses Oracle Spatial for storing, analyzing and presenting geographical data. A typical installation consists of a database server and one or more Tomcat web servers [6]. Larger installations often use a fail-over database server and a load balancer in front of the Tomcat servers, however these contextual parts are not considered in this project.

Client-server communications in dpSpatial are made using the communications protocol HTTP(S)/1.1 over one or more TCP connections. Transmitted data are serialized using Java Object Serialization. Communications between server and database in dpSpatial are not altered in any way during this project, as the client-server communications are in focus.

Connections in dpSpatial are handled with Java URLConnection objects. URLConnection is the superclass of all classes that represent a communications link between the application and a URL and can be used both to read from and to write to the resource referenced by the URL. How a client-server connection is set up in dpSpatial can be seen in Appendix A.

Figure 2.1: The dpSpatial base platform [6].
2.4 The Hypertext Transfer Protocol

In order for a client and server to understand each other, they must follow a shared set of rules defining the dialogue pattern. The rules are specified by a so-called application protocol, which is a communications protocol that operates in the application layer.

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypertext information systems [7]. It moves data between client and server in a request-response manner in the form of messages over a TCP connection. HTTP is a half-duplex protocol, meaning information can flow in both directions but not simultaneously. The most commonly used version is HTTP/1.1 ever since it was standardized in 1997. However, HTTP/2 was standardized in 2015 and is today supported by many web servers and browsers [20]. Both HTTP/1.1 and HTTP/2 are evaluated in this project and are described in more detail below.

2.4.1 HTTP/1.1

In HTTP/1.1 the client initiates a request message by establishing a TCP connection to a server, which then returns one or more response messages containing a status line and possibly the requested payload. Each message also contains some header fields holding information about the connection. An example of a request-response message exchange is represented in Figure 2.2.

HTTP/1.1 supports pipelining, meaning multiple requests can be sent over a single TCP connection without having to wait for the corresponding responses [7]. However, the server still has to respond to the request messages in the same order as they were received and cannot proceed with the next request until the whole response is delivered. This causes a problem known as head-of-line blocking. Clients that need to make many requests must therefore make use of multiple TCP connections in order to achieve concurrency and speed up the communications [1].
2.4.2 HTTP/2

HTTP/2 is the latest version of HTTP. It aims to solve some issues present in HTTP/1.1 and to improve the protocol’s efficiency. For instance, HTTP/2 enables more efficient use of network resources and reduced latency by introducing header field compression. It also addresses the issue of head-of-line blocking by supporting multiplexing, allowing concurrent request-response exchanges over the same TCP connection. Prioritization of requests is also supported, allowing more important ones to complete faster.

2.5 Data Serialization

The process of translating data structures or object states into a format that can be stored or transmitted is what is known as serialization. The result is a series of bits that can later be reconstructed into an identical copy of the original structure or object according to the serialization format (i.e. deserialization). The serialization formats that are evaluated in this project are Java Object Serialization and Google’s Protocol Buffers.
2.5.1 Java Object Serialization

Most standard classes in Java support serialization of their objects. In order to enable Java Object Serialization for a class, one must make it implement either the Serializable or Externalizable interface. Implementing the Serializable interface will cause class objects to be serialized automatically using the standard implementations of the methods `writeExternal` and `readExternal`. When implementing the Externalizable interface, one must first implement these methods in the class.

When an object is serialized with Java Object Serialization a stream of bytes is produced, called an object stream. Some Java Object Stream classes are partly based on data stream classes, making Java object streams able to contain both primitive and object values [4].

Objects often contain references to other objects, which may in turn be referencing other objects. All referenced objects must be written to and read from the object stream along with the original object in order to maintain their relationships. The reference graph of an object is traversed recursively during both serialization and deserialization [3]. Figure 2.3 shows an illustration of this process.

![Figure 2.3: I/O of multiple referenced objects](image)

2.5.2 Protocol Buffers

Protocol Buffers (a.k.a. ProtoBuf) are a way of serializing structured data for use in e.g. communications protocols. They are developed by Google and were initially released to the public in 2008. The design
aims to be simple yet have high performance.

To use ProtoBuf, one must first specify how data should be structured by defining ProtoBuf message types in .proto files. Each such message is a small logical record containing a series of name-value pairs. The message format is relatively simple and allows for hierarchical structuring of data. Then, the ProtoBuf compiler is run on the .proto files, which generates data access classes providing access functions for each field and (de)serialization methods. [5]

2.6 Related Work

2.6.1 Are HTTP/2 Servers Ready Yet?

In a study by Jiang et al. [12] they investigate whether HTTP/2 servers have correctly realized the protocol’s new features and how the deployed servers use these features. To do this, they inspect six popular implementations of HTTP/2 servers and measure the top 1 million Alexa websites. They specifically look at six features of HTTP/2, being multiplexing, flow control, request priority, server push, header compression and ping.

Results show that not all implementations strictly follow RFC 7540 (i.e. the technical report of HTTP/2 [1]). Some features, like server push and prioritization of requests, have not been well implemented or fully utilized by websites. It is also stated that some features may be exploited by adversaries to launch DoS (Denial of Service) attacks.

2.6.2 Is HTTP/2 Really Faster Than HTTP/1.1?

A study by Saxce, Oprescu, and Chen [17] aims for a better understanding of HTTP/2 from a practical standpoint and to compare HTTP/2 to HTTP/1.1. For evaluation, they choose to compute the page load time (PLT), as they believe it is the most suitable metric for both the user and the content provider; the end user wants to browse the web in the fastest way and the content provider needs to increase its income (e.g. shopping, ads) by speeding up the content delivery to the user.
Comparison tests considered four of the main features of HTTP/2, being header compression, multiplexing, server push and prioritization. Results show that with HTTP/1.1 the PLT increases linearly with the number of requests, while HTTP/2 gives a constant PLT due to multiplexing (Figure 2.4). The study also states that HTTP/2 is mostly interesting for complex websites involving a significant number of requests, showing a decrease in PLT with an average of 20%. More simplistic websites with few dependencies will not benefit much from a switch to HTTP/2.

![Figure 2.4: Page load time vs. Number of requests, 100ms latency, 0% loss](image)

2.6.3 Is the Web HTTP/2 Yet?

In a study by Varvello et al. [20] it is investigated whether HTTP/2 will or should be the future of the web. They make use of a measurement platform that they built themselves [19], which monitors HTTP/2 adoption and performance across the Alexa top 1 million websites on a daily basis.

Results show that HTTP/2 does not currently succeed in serving a page using a single TCP connection, as 50% of websites using HTTP/2
today use at least 20 TCP connections. Also, most websites exhibit HTTP/1.1 practices like inlining and domain sharding into HTTP/2. Sharding causes HTTP/2 websites to use more TCP connections than necessary and inlining may reduce the utility of caching. The study also states that 80% of websites adopting HTTP/2 experience a decrease in PLT compared to HTTP/1.1, even when only partially using HTTP/2.

### 2.6.4 Native Web Communication Protocols and Their Effects on the Performance of Web Services and Systems

A study conducted by Naik et al. [15] investigates the impact of HTTP/2 on the overall performance of web services and systems from an end-user perspective. Six different experiments are performed to examine the protocol’s effects in reducing the round-trip time (RTT) and web latency.

Results show that support of HTTP/2 at the client side does not make any significant improvements in the overall performance of any websites, compared to HTTP/1.1. This suggests that overall performance is instead heavily dependent on other factors, such as website contents, server location, data transfer rate, number of intermediate nodes and traffic density.

### 2.6.5 A Comparison of Data Serialization Formats For Optimal Efficiency on a Mobile Platform

A study by Sumaray and Kami Makki [18] compares four different data serialization formats with the focus on serialized data size, serialization/deserialization speed and usability. Here, Google’s ProtoBuf is compared to XML, JSON, and Apache Thrift in a Java application run on an Android device. The application is developed by the authors themselves and tests make use of both text-heavy and number-heavy objects.

Results show that ProtoBuf outperforms the other formats, however only by a small amount when compared to Thrift. XML performs the
worst in all categories by far. Regarding serialized data size, it is ap-
parent that classes consisting of more numbers than strings can be se-
rialized into small data sizes when using the binary formats (i.e. Thrift
and ProtoBuf). Regarding execution speed, serialization is costlier
than deserialization for all formats, taking more than double the amount
of time. Serialization using the binary formats take roughly half the
time required for JSON.

The authors state that XML should be avoided unless necessary, as
JSON is a superior text-based alternative. When serializing data for
storage purposes or when designing a new web service from the ground
up, it is advantageous to use one of the binary formats due to their su-
perior execution speeds and data sizes.

2.6.6 Performance Evaluation of Object Serialization
Libraries in XML, JSON and Binary Formats

In a study conducted by Maeda [14], twelve object serialization li-
braries are compared based on qualitative and quantitative aspects in
Java. Each library is used to serialize a common example to a file. The
size of the file and the processing times are measured during the exe-
cution.

Results show that the serialized data size produced by the built-in se-
rialization in Java is larger than the text-based formats, even though it
writes objects in binary format. This seems to be because the serialized
data contains type information. Java serialization is also one of the top
worst formats regarding execution time.

According to the study, ProtoBuf has the best serialization time and
is tied for best deserialization time, together with Apache Avro in bi-
nary and Apache Thrift in JSON. From qualitative aspects, the size of
binary-based serialized data is better than text-based alternatives, like
XML and JSON. The authors also state that a schema compiler is im-
portant for good performance, something that ProtoBuf and Apache
Thrift provide, while Java serialization and Apache Avro do not.
Chapter 3

Methodology

Carrying out this project mainly consisted of two phases: implementation and evaluation. During the implementation phase, the different communications protocols and serialization formats were implemented into the dpSpatial base platform for two different server calls. During the evaluation phase, the network traffic produced by each server call was monitored and benchmarks were run in order to measure the call duration. Relevant parts of the dpSpatial code base were studied in order to understand how the system was built, so that implementations and measurements could be made in a correct and fair manner.

3.1 Selected Server Calls

A server call in dpSpatial consists of the client requesting the server to perform some action, by which the server processes the request and responds with some data to the client. Different types of requests are on the server side processed by different Java servlets. A Java servlet is a Java program that extends the capabilities of a server, and a server call in dpSpatial is always made to a specific servlet.

There are many different server calls that can be made in dpSpatial. It is however not necessary to examine each one of these for the purpose of this thesis. It is sufficient to select some signature calls that are frequently being made when running a client, each representing a certain form of client-server request-response exchange. For this study, the two dpSpatial server calls GetGeometryComponents and LockOb-
ject were selected for measurements. The first mentioned is a call dealing with much data. It was selected as GIS/NIS applications generally deal with a lot of data and it therefore represents typical communications well. The last mentioned call instead deals with little data, and was selected as it is interesting to investigate the differences in the results based on the amount of data handled. The two calls are described in more detail below.

3.1.1 GetGeometryComponents

The GetGeometryComponents server call is made whenever the client needs to display a specific geographic area containing a number of geographic objects. This happens whenever a user changes the geographic view in any way, such as panning around in the map, zooming in or out, or enabling/disabling different map perspectives. These are commonly used operations when using a GIS application, meaning calls to GetGeometryComponents are frequently made in dpSpatial.

The client-server communications of this call generally consist of a small request made by the client, followed by a large response from the server. The request specifies coordinates of the area, scale, ID of the work set and possible object filters. The response contains all geographic objects in the geographic context specified by the request. In this project, 1340 geographic objects are requested and transmitted.

3.1.2 LockObject

Each geographic object in a dpSpatial application is either locked by some work set or unlocked. This is to ensure that several people are not modifying the same object at the same time. The LockObject server call is made when trying to lock a geographic object in the client. When working in a dpSpatial application, different geographic objects are being manipulated regularly, meaning calls to LockObject are frequently made.

The client-server communications of this call generally consist of a small request made by the client, followed by a small response from the server. The request specifies the type and ID of the object to be locked. The response contains a boolean value, specifying if the object
was indeed locked. In contrast to the GetGeometryComponents call, this is a simple call.

3.2 Implementations

To avoid having to enforce the two specific server calls described in section 3.1 from within an actual dpSpatial application, a program was built with the capability of making the calls independently. By doing this, it was simple to perform benchmarks accurately and control the flow of the system calls. The program takes three arguments: the server call to be made, the protocol that should be used for client-server communications and the serialization format to be used.

3.2.1 HTTP Client

The implementation of the HTTP client was made using OkHttp, which is an open-source HTTP and HTTP/2 client for Android and Java applications [11]. It also uses Okio for fast I/O and resizable buffers. The latest version of OkHttp at the time was v3.10.0. However, it did not allow exclusive use of HTTP/2, meaning there was no way to configure the system to only use HTTP/2 traffic. This was a problem as this functionality was needed to evaluate the protocol properly.

The source code of OkHttp was studied in order to find a solution to the problem. Looking at the head of the master branch, which then represented an unreleased state of OkHttp v3.11.0, it was found that exclusive use of HTTP/2 traffic had been added as a functionality. In order to use this needed addition, the master branch was pulled from git and built into a JAR file, which then replaced the earlier version of OkHttp for use in this project.

The implementation works by using an OkHttpClient object, which can be configured in several ways using a Builder. In this way, the protocol(s) that should be used can be specified, which in this case means specifying either HTTP/1.1 or HTTP/2. To make a request, a Request object must first be created. It may be configured using a Builder to specify headers, cache control, request type, URL and body. The call to the server is invoked by calling the method newCall of the OkHttpClient with the Request object as an argument. A Response object is
returned, containing the response from the server.

### 3.2.2 ProtoBuf

In order to use ProtoBuf, as explained in section 2.5.2, message types had to manually be defined in .proto files. Since measurements in this work only concern the two server calls LockObject and GetGeometryComponents, it was sufficient to define message types relevant for these calls. A proto file was created for each call, LockObject.proto and GetGeometryComponents.proto, which can be seen in Appendix B and Appendix C respectively. The ProtoBuf compiler was then run on the two files in order to generate the data access classes.

Handling of ProtoBuf had to be implemented both on the client-side and server-side. On the client-side, data to be transmitted in a request were first converted into ProtoBuf messages, and data received in the response from the server were converted back from ProtoBuf messages into Java objects. On the server-side, data received in a request from the client were converted from ProtoBuf messages into Java objects, and data to be transmitted in the response to the client were converted from Java objects into ProtoBuf messages.

### 3.2.3 Server Configurations

In order to make the Tomcat web server support HTTP/2 traffic, a specific Upgrade Protocol component had to be added to the Connector in the server.xml file:

```xml
```

### 3.3 Evaluation

In order to get as fair results as possible, some functionalities used in dpSpatial had to be disabled. Compression with gzip was sometimes used in dpSpatial communications, and was disabled for all communications in this project. Also, some classes concerned when making a GetGeometryComponents call were implemented with their own versions of the `writeExternal` and `readExternal` methods of Java Object Serialization. This was to adapt the serialization of these objects to vari-
ous circumstances in order to cut out as much data as possible. As this would have an effect on the network traffic load and make measurements less fair, Digpro’s own handling of Java Object Serialization in dpSpatial was fully disabled.

### 3.3.1 Metrics

The call duration of both the LockObject call and the GetGeometryComponents call is measured for all combinations of communications protocol and serialization format. The size of HTTP overhead is examined for HTTP/1.1 and HTTP/2. Also, the size of HTTP payload is examined for Java Object Serialization and ProtoBuf.

### 3.3.2 Java Microbenchmark Harness

Java Microbenchmark Harness (JMH) is a Java tool for building, running and analysing benchmarks written in Java and other languages [2]. JMH provides several benchmark configurations, such as the number of forks, warmup iterations, test mode and time unit. This functionality made it convenient to use JMH in this project for building and running benchmarks in a flexible way.

There are some alternative ways to build benchmarks using JMH. One way is to use annotations provided by JMH, which should then be used to annotate the methods to be measured. Another way is to use an Options object configured in a suitable way, which is then passed to a Runner object on which `run` is called.

Listing 3.1 depicts how JMH was used for measurements in dpSpatial. No forks were made, meaning benchmarks were run in a single thread. Three warm-up iterations and ten measurement iterations were run for each benchmark, where each iteration was set to run for 10 seconds. The mode was set to measure the average running time and to display the results in milliseconds.

**Listing 3.1: Usage of JMH for benchmarks in dpSpatial.**

```java
Options opt = new OptionsBuilder()
    .include (RunBenchmark.class.getSimpleName ())
    .forks (0)
    .warmupIterations (3)
```

3.3.3 Wireshark

Wireshark is a widely-used network protocol analyzer that allows the user to see what is happening on their network at a microscopic level [8]. It was used in this project to capture the network traffic during client-server communications in dpSpatial. Specifically, Wireshark was used to analyze the network traffic generated by the two server calls LockObject and GetGeometryComponents described in section 3.1.
Chapter 4

Results

In this chapter, the results gained from running the benchmarks on and monitoring the network traffic of the two dpSpatial calls are presented and explained. As stated in section 3.3.1, the metrics in focus are call duration, HTTP overhead size and payload size. Each metric is addressed separately from each other throughout this chapter. Figures in this chapter refer to Java Object Serialization as JOS and to ProtoBuf as PB, for the purpose of readability. Implications and significance of these results are discussed in Chapter 5.

4.1 Call Duration

The call duration of the two dpSpatial calls is presented here. As explained in section 3.3.1, the call duration is here defined as the time between sending the request from the client and having received the complete response from the server. Note that communications between the server and database are included in these results.

Figures 4.1 and 4.2 show the average call duration of the calls LockObject and GetGeometryComponents respectively. Each bar in the figures represents a specific combination of a communications protocol and serialization format, making a total of four combinations.

4.1.1 LockObject

Figure 4.1 shows the average call duration of the LockObject server call. Looking at the two leftmost bars, one can see that the variation
in call duration between using the different serialization formats with HTTP/1.1 is less than 1 ms, showing a slight decrease in call duration for ProtoBuf. Looking at the two rightmost bars, one can see that the same thing holds when applied to HTTP/2.

Comparing the two leftmost bars with the two rightmost bars, the difference in call duration between using HTTP/1.1 and HTTP/2 can be observed. For HTTP/1.1 the call duration is about 50 ms and for HTTP/2 it is about 48 ms, meaning that the call duration is about 2 ms shorter with the use of HTTP/2.

All in all, Figure 4.1 shows that the call duration is affected more by the choice of communications protocol than the choice of serialization format for the LockObject call.

![Figure 4.1: Average call duration in milliseconds when calling LockObject for different combinations of communications protocols and serialization formats.](image-url)
4.1.2 GetGeometryComponents

Figure 4.2 shows the average call duration of the GetGeometryComponents server call. Comparing the bars representing the use of Java Object Serialization with the bars representing the use of ProtoBuf, one can see that the call duration is 4 ms shorter with ProtoBuf.

Comparing the two leftmost bars with the two rightmost bars, the difference in call duration between using HTTP/1.1 and HTTP/2 can be observed. It shows that the call duration is 1 ms shorter when using HTTP/2 instead of HTTP/1.1.

All in all, Figure 4.2 shows that the call duration is affected more by the choice of serialization format than the choice of communications protocol for the GetGeometryComponents call. This outcome is different from the results of the LockObject call, where the choice of communications protocol is the more determinant factor.

![Figure 4.2: Average call duration in milliseconds when calling GetGeometryComponents for different combinations of communications protocols and serialization formats.](image-url)
4.2 HTTP Overhead Size

The total HTTP overhead size for each of the two dpSpatial calls is presented here. Total overhead size here means the sum of the HTTP request overhead size and the HTTP response overhead size. As serialization only affects the HTTP payload and not the overhead, they are not relevant factors when measuring overhead sizes. The communications protocols are therefore in focus for this metric.

The results for both calls is shown in Figure 4.3. One can see that there is a clear difference in overhead size between the two protocols for both calls. The HTTP/2 overhead is 417 bytes smaller for LockObject and 396 bytes smaller for GetGeometryComponents, compared to the HTTP/1.1 overhead. On average, this shows an 87.2% decrease in overhead size with HTTP/2.

![Figure 4.3: Total HTTP overhead size in bytes of the request-response exchange between client and server for the dpSpatial calls LockObject and GetGeometryComponents using HTTP/1.1 and HTTP/2.](image-url)
4.3 HTTP Payload Size

The total HTTP payload size for each of the two dpSpatial calls is presented here. Total payload size here means the sum of the HTTP request payload size and the HTTP response payload size. The two serialization formats are in focus for this metric, as the payload size is not affected by the choice of communications protocol.

4.3.1 LockObject

Figure 4.4 shows the payload sizes of the LockObject call for the two serialization formats. It shows that there is a significant difference in payload size between the two formats. The use of ProtoBuf here results in a 126 bytes smaller payload size than that of Java Object Serialization, which is a 91.3% decrease.

![Figure 4.4: Total HTTP payload size in bytes of the request-response exchange between client and server when calling LockObject.]

4.3.2 GetGeometryComponents

Figure 4.5 shows the payload sizes of the GetGeometryComponents call for the two serialization formats. Note that the size is specified in kilobytes (kB) and not bytes. One can see that there is a difference
in payload size between the two formats. The difference is however less significant than for the LockObject call. The use of ProtoBuf here results in a 53 kB (= 53 000 bytes) smaller payload size than that of Java Object Serialization, which is a 4.4% decrease.

Figure 4.5: Total HTTP payload size in kilobytes of the request-response exchange between client and server when calling GetGeometryComponents.
Chapter 5

Discussion

In this chapter, the results presented in Chapter 4 are discussed. Reliability of gathered data and possible sources of error throughout this project are also considered. The significance of this work is reflected upon, and suggestions for future research related to this work are presented.

5.1 Performance

5.1.1 Call Duration

As mentioned in section 4.1, the response time of the LockObject call is mainly affected by the choice of communications protocol, while for the GetGeometryComponents call the choice of serialization format is the ruling factor. This most likely has to do with the sizes of the calls. Concerning the LockObject call, as explained in section 3.1.2, it is a simple call consisting of a small request followed by a small response. Since there are not much data being sent, a change of serialization format will barely affect the call time. What takes up most space in the HTTP traffic is the overhead, which is something that is affected by a switch from HTTP/1.1 to HTTP/2. Due to this, it is reasonable that the choice of communications protocol will be the ruling factor of the LockObject call.

Concerning the GetGeometryComponents call, as explained in section 3.1.1, it is a more complicated call where a lot of data is being transmitted. Because there is a large amount of data, it is rational that the
choice of serialization format will have some form of effect on the call duration. The data take up the majority of the HTTP traffic during this call, meaning effects on the overhead will not be as meaningful. As such, it is reasonable that the choice of serialization format will be the ruling factor of this call.

5.1.2 HTTP/1.1 vs. HTTP/2

As mentioned in section 4.2, there is about an 87% decrease in HTTP overhead when using HTTP/2 in contrast to HTTP/1.1. Most likely, this has to do with the fact that HTTP/2 uses header compression, which HTTP/1.1 does not support. Concerning the network traffic load, there seem to only be improvements when switching to HTTP/2 from HTTP/1.1. In the context of call duration, the use of HTTP/2 also seems to be the better choice.

The conclusion that can be drawn here is that HTTP/2 should generally be the preferred choice over HTTP/1.1 when building GIS/NIS applications. However, in a real-world perspective, there are circumstances where an existing GIS/NIS application has been built that uses HTTP/1.1 for network communications. In such cases, it may be necessary to weigh the actual benefit that would be gained from converting to HTTP/2 against factors like the time and cost the switch would require. There is also the fact that HTTP/2 unarguably is the latest version of HTTP, which may be a reason in itself to choose it over HTTP/1.1; on a societal level, one could argue that it would contribute to the general development of the web.

5.1.3 Java Object Serialization vs. ProtoBuf

As mentioned in section 4.3, there is a decrease in HTTP payload size gained from using ProtoBuf instead of Java Object Serialization. However, comparing the results of the LockObject call and the GetGeometryComponents call, the proportional decrease differs substantially, being 91.3% and 4.4% respectively. A likely explanation to this has to do with how the two serialization formats work in practice.

The byte stream produced by Java Object Serialization will generally consist of a header with necessary information about the following
content, a class descriptor for each referenced object class and the actual data. With ProtoBuf, there are no such byte stream headers or class descriptors. As the GetGeometryComponents call mainly consists of a bunch of GeometryComponent objects, there are only a few different class descriptors present in the Java Object Serialization byte stream. The vast majority of the byte stream will therefore consist of the actual data, outweighing the byte stream header and class descriptors, which is likely why the difference in size between Java Object Serialization and ProtoBuf is not that great for this call. The Java Object Serialization byte stream of the LockObject call instead consists mainly of the header and class descriptors, meaning the difference in size will be proportionally greater for this call.

The results suggest that ProtoBuf should be the preferred choice over Java Object Serialization. There are however some factors that have not been mentioned during this project that should probably be taken into consideration when choosing serialization format. One is that Java Object Serialization requires more memory, because each object that has been serialized has to be remembered until the end of the process, in order to handle circular dependencies. Another factor is that Java Object Serialization is much simpler to use, as all that is needed to make a class support serialization is to make it implement the Java interface Serializable. To use ProtoBuf, message types have to be defined manually in .proto files, which may be quite extensive for big systems like GIS’s.

5.2 Sources of Error

Since the implementations and evaluations done in this project were made directly in the dpSpatial base platform, there is a risk of the environment affecting the results. As mentioned before, relevant parts of the code base were studied. There is however the possibility that some functionality of the system that affect the results have been overlooked. Nonetheless, the likelihood of this being the case is low, as a system administrator of dpSpatial has followed the complete process of this project. In order to be entirely certain that no affecting parts of dpSpatial have been overlooked, one would probably have to study and understand the entire code base.
It is also important to consider the fact that the communications taking place between the server and database are included in the measurements of call duration. Depending on how much of the call durations that consist of this, the call duration results may be more significant than they seem. For instance, if the server-database communications take up 40 ms of the LockObject call duration, the client-server communications would only take the total call duration subtracted by 40 ms. Looking at Figure 4.1, 40 ms is a large part of the call durations, meaning the differences in actual client-server communications time would be proportionally greater. However, even small time improvements may be meaningful if the concerned call is frequently made by the system.

During measurements, the Tomcat web server was run locally on the same machine as the client. It is possible that one would obtain more evident results if the server is run on another machine, or if a simulated network latency is applied.

5.3 The Bigger Picture

The findings of this work may be a contributing factor to the development and potentiation of GIS/NIS applications. As stated earlier, the applicability of GIS and NIS is extensive. For instance, in an article by GISGeography [9] over 1000 GIS applications and their uses are listed. Due to such systems being used in such a wide variety of areas, improvements of GIS/NIS applications may result in better decision making in our society as a whole.

This work does not bring any new functionality or purpose to GIS/NIS applications. It instead contributes to the efficiency of how such applications work internally, concerning network communications happening within the system. Such improvements may make the application more responsive and decrease the amount of network traffic generated by it. This means that operations are performed faster, which in a real-life perspective makes it possible to complete work dependent on GIS/NIS applications in less time.
Another aspect is that Oracle, the company behind Java, is planning to scrap Java Object Serialization due to existing security issues \cite{13}. Mark Reinhold, chief architect of the Java platform group at Oracle, states that it will be replaced with a small serialization framework of some kind. It is currently unknown for which release of Java this will be carried out or precisely how the new serialization framework will perform. Due to this, it is important to consider alternative serialization formats.

Additionally, as briefly reflected upon in section 5.1.2, choosing to use HTTP/2 instead of HTTP/1.1 may contribute to the general development of the web. HTTP/2 is the latest version of HTTP which addresses some issues present in HTTP/1.1, and also some other general improvements to the protocol as explained in section 2.4.2. It is clear that the purpose of the creation of HTTP/2 is to improve the overall performance of the web. By using HTTP/2 rather than HTTP/1.1, possible new issues may be identified that can be considered in future development of the web.

### 5.3.1 Ethics and Sustainability

During this project, no handling of personal or sensitive data has occurred. Also, use of GIS/NIS applications does not entail any unethical aspects. Therefore, there are no substantial ethical aspects of this thesis.

GIS/NIS applications are used in many important areas of our society and improvements to their efficiency may imply faster and more efficient decision making. For sustainability, it is possible that this can save energy and other resources in the long run.

### 5.4 Future Research

As the server-database communications were included in the calculations of call duration in this project, it is hard to determine just how significant the differences in call durations are for the client-server communications. In future research, the client-server communications and the server-database communications should be separated for more precise measurements.
In dpSpatial, there are few occurrences of concurrent client-server communications. Therefore, the multiplexing feature of HTTP/2 could not be utilized in this work. Concurrent communications are something that may however be present in other GIS/NIS environments, making multiplexing an interesting aspect to evaluate for such systems. Previous research has shown that there is a decrease in loading times on the web utilizing HTTP/2 with multiplexing [17][20], but similar research for GIS/NIS applications is something that should be explored.

Since Java Object Serialization may have to keep a large number of objects in memory, the memory usage for different serialization formats could be a relevant aspect to measure in future work. In addition to this, more communications protocols and serialization formats should be evaluated in the context of GIS/NIS applications.
Chapter 6

Conclusions

The work conducted in this project suggests that HTTP/2 outperforms HTTP/1.1 in several aspects. Results show an 87% decrease in HTTP overhead size when moving from HTTP/1.1 to HTTP/2. The benefit to be made in call duration seems to be greater when HTTP/2 is applied to small simple calls, rather than complex calls containing a large amount of data. It also proposes that Google’s Protocol Buffers outperform Java Object Serialization, looking at the sizes of produced byte streams gained from serializing data. However, the benefit of using ProtoBuf instead of Java Object Serialization is significantly greater when applied to data consisting of many different object types, rather than data consisting of only a few object types.

GIS/NIS applications usually deal with a lot of different server calls. Some are simple calls containing little data, others are complex calls dealing with a lot of data. Looking solely at performance, HTTP/2 and ProtoBuf would be the preferred choices over HTTP/1.1 and Java Object Serialization respectively. However, the actual benefit heavily depends on how the GIS/NIS application is built and what different kinds of server calls are being made. In a real-world perspective, one therefore has to weigh the benefit that may be gained against factors like economy, time and complexity of the implementation.
Bibliography


Appendix A

Setup of a client-server connection in dpSpatial using a URL-Connection

```java
private URLConnection setupConnection(String url, Integer workset) throws IOException {
    URLConnection urlc =
        new URL (url).openConnection ();
    urlc.setUseCaches (false);
    urlc.setDoOutput (true);
    urlc.setRequestProperty ("App", getApp () + ";" +
        getLang ());
    String sid = getSessionId ();
    if (sid != null)
        urlc.setRequestProperty ("DpsSession", sid);
    if (workset != null)
        urlc.setRequestProperty ("Workset", "" +
            workset.intValue ());
    if (gzip)
        urlc.setRequestProperty ("Accept-Encoding",
            "gzip");
    notifyServiceCalledListeners (url);
    return urlc;
}
```
Appendix B

LockObject.proto

```protobuf
syntax = "proto3";
package bios.jcommon;

option java_package = "bios.jcommon.proto";
option java_outer_classname = "LockObjectProto";

message ObjectId {
  int32 otype = 1;
  int32 oid = 2;
}

message GotLock {
  bool ok = 1;
}
```
Appendix C

GetGeometryComponents.proto

```protobuf
syntax = "proto3";
package bios.jcommon;

option java_package = "bios.jcommon.proto";
option java_outer_classname = "GetGeometryComponentsProto";

message CoordRectangle {
  double xmin = 1;
  double ymin = 2;
  double xmax = 3;
  double ymax = 4;
}

message GeometryComponentType {
  int32 otype = 1;
  int32 subtype = 2;
  int32 ctype = 3;
}

message ObjectSelection {
  int32 productId = 1;
  string coordsys = 2;
  repeated GeometryComponentType wanted = 3;
  repeated GeometryComponentType unwanted = 4;
}

message Request {
  CoordRectangle rect = 1;
  double scale = 2;
  ObjectSelection objsel = 3;
}
```
message ComponentId {
  int32 otype = 1;
  int32 oid = 2;
  int32 ctype = 3;
  int32 occurrence = 4;
}

message DGeometry {
  int32 gtype = 1;
  int32 srid = 2;
  repeated int32 elemInfo = 3;
  repeated double ordinates = 4;
}

message PlaceType {
  GeometryComponentType gct = 1;
  int32 state = 2;
  int32 dspFlag = 3;
}

message IndAttribute {
  int32 lineColor = 1;
  int32 fillColor = 2;
  int32 style = 3;
  int32 weight = 4;
  double weightMeter = 5;
  int32 fillStyle = 6;
  double spacing = 7;
  int32 zorder = 8;
}

// TextComponent is subclass of GeometryComponent.
message TextFields {
  int32 justification = 1;
  string text = 2;
  double height = 3;
}

// SymbolComponent is subclass of GeometryComponent.
message SymbolFields {
  DGeometry location = 1;
  string fontname = 2;
  string symbolnumber = 3;
  double rotation = 4;
  double height = 5;
  double width = 6;
message GeometryComponent {
  ComponentId cid = 1;
  int32 lock = 2;
  DGeometry geom = 3;
  bool modifiable = 4;
  bool deleted = 5;
  PlaceType pt = 6;
  IndAttribute ia = 7;
  oneof subFields {
    TextFields textFields = 8;
    SymbolFields symbolFields = 9;
  }
}