Master Thesis

im Rahmen des
Universitätslehrganges „Geographical Information Science & Systems“
(UNIGIS MSc) am Interfakultären Fachbereich für GeoInformatik (Z_GIS)
der Paris Lodron-Universität Salzburg

zum Thema

”Client-side Visualisation of Scientific Raster Data Using WebGL and Open-Source Web Mapping Technologies“

vorgelegt von

Bernhard Baumrock, BA
1423598, UNIGIS MSc Jahrgang 2015

Zur Erlangung des Grades
„Master of Science (Geographical Information Science & Systems) – MSc(GIS)”

Wien, 21.02.2018

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 21.02.2018

_____

Bernhard Baumrock, BA

Acknowledgement

This thesis was written in 2017/2018 at EOX IT Services GmbH in Vienna. First of all, I would like to thank **Mag. Joachim Ungar**, who was my first point of contact for all questions regarding my thesis. I was really struggling finding a topic for my thesis until I met Joachim and he told me about EOX and what they are doing. I doubt it would have been possible to finish my study without him. Thank you!

Next, I want to thank **Mag. Fabian Schindler** and **Mag. Daniel Santillan** for helping me with the implementation of the prototype and for having answers to all my questions regarding geotiff.js and plotty.

Last but not least, I want to thank **Dr. Gerhard Triebnig** for having me in their team while I was working on my thesis, for letting me join the EO Open Science conference at the European Space Agency in 2017 and for all the support regarding this thesis.

Furthermore, I want to thank everybody of the UNIGIS team for their support during my study, especially **Mag. Julia Moser** for answering all my administrative questions and **Dr. Gudrun Wallentin** for supervising this thesis.

Abstract

Web mapping has come a long way from its beginnings in 1993. Browsers get more powerful and modern web technologies allow highly performant operations to be done on the client-side using the processing power of local graphic cards. This thesis discusses the possibilities of utilizing these resources in web mapping applications for displaying complex scientific raster data, improving the overall user experience while reducing server load.

The developed prototype demonstrates feasibility of such a highly performant client concept for managing GeoTIFF data within browsers. It has been implemented using WebGL technology and integrates various open source software libraries among which are OpenLayers, geotiff.js and plotty.

A comparison of this (load wise) client-focused prototype has been performed against traditional server-focused service setups (WMS, WMTS) by measuring defined key performance indicators (timings and amount of data transferred). Available network bandwidth between client and server has been taken into consideration as well. A benchmark tool has been custom-built and used for deriving the quantitative benchmark results.

The results show that the network speed is the main determining parameter for interaction timings, meaning that the new client-side processing method makes the user experience better (faster) when the network speed is high. On the other hand, it is also shown that it can make sense to use this method at slow network speeds, fetching data only once and then interacting with this data solely on the client-side (having an offline usage scenario).

The data transferred between the server and the client is most likely less using traditional methods with good compression (JPG tiles). This is also the reason why the network speed is such an important factor.

Keywords: Web mapping, WebGL, GeoTIFF, OpenLayers, geotiff.js, plotty, performance, benchmark, WMS, WMTS

Table of Contents

List of Figures

Abbreviations

OGC	Open Geospatial Consortium
WMS	Web Map Service
WMTS	Web Map Tile Service
WCS	Web Coverage Service
AJAX	Asynchronous JavaScript and XML
DOM	Document Object Model
KPI	Key Performance Indicator
RT	Request Time
ORT	Overall Request Time
DT	Data Transferred
ODT	Overall Data Transferred
NS	Network Speed
TF	Tile Filesize
PTS	Processing Time on Server
PTC	Processing Time on Client
NOT	Number Of Tiles
NOI	Number Of Interactions
kB	Kilobyte
MB	Megabyte
NDVI	Normalized Difference Vegetation Index
GLI	Green Leaf Index
UI	User Input / User Interface
TTFB	Time To First Byte

1 Introduction

1.1 Motivation

Web-mapping started soon after the emergence of the world wide web with the introduction of the Xerox PARC Map Viewer in 1993 [1]. A little more than ten years later Google Maps, OpenStreetMap, Google Earth and OpenLayers came up (2004/2005) [2]. In recent years one can observe a shift from raster to vector web maps – both Google and Apple transitioned their maps and also MapBox (founded in 2010) announced the use of Vector Tiles in 2013 [3]. The idea is that vector tiles are applying the strengths of tiling – developed for caching, scaling and serving map imagery rapidly – to vector data, making it possible to get different styles of the same map (same data) depending on user input or user settings [4]. What is already possible for regular (street) web maps could also be a great possibility for the visualisation of complex scientific raster data.

1.2 Literature Overview

The evolution of web mapping lead to the implementation of several mature web service standards, making it possible to have distributed systems that work well together. As in this thesis the objective has been to evolve selected aspects of web mapping and to make heavy use of related web service protocols, a condensed overview of the most relevant software technology and interoperability standards which were investigated during the thesis in detail is provided in the following.

The OGC (Open Geospatial Consortium) is the organisation that creates and manages those standards and describes itself on their website as follows [5]:

> The OGC is an international not for profit organization committed to making quality open standards for the global geospatial community. These standards are made through a consensus process and are freely available for anyone to use to improve sharing of the world's geospatial data. OGC standards are used in a wide variety of domains including Environment, Defense, Health, Agriculture, Meteorology, Sustainable Development and many more.

The OGC standards baseline comprises more than 30 standards – for this thesis the most important ones are the Web Map Service (WMS), the Web Map Tile Service (WMTS) and the Web Coverage Service (WCS).

1.2.1 Web Map Service (WMS)

As of writing this thesis the current WMS standard by OGC is version 1.3.0 published in 2006 and described as follows [6]:

> The OpenGIS® Web Map Service Interface Standard (WMS) provides a simple HTTP interface for requesting geo-registered map images from one or more distributed geospatial databases. A WMS request defines the geographic layer(s) and area of interest to be processed. The response to the request is one or more geo-registered map images (returned as JPEG, PNG, etc) that can be displayed in a browser application. The interface also supports the ability to specify whether the returned images should be transparent so that layers from multiple servers can be combined or not.

On the WMS introduction page a sample WMS request can be found [7]:

```
http://metaspatial.net/cgi-bin/ogc-wms.xml
 ?VERSION=1.3.0
 &REQUEST=GetMap
 &SERVICE=WMS
 &LAYERS=DTM,Overview,Raster_250K,
  Topography,nationalparks,Infrastructure,Places
 &STYLES=,,,,,,
 &CRS=EPSG:27700
 &BBOX=424735.97883597884,96026.98412698413,
  467064.02116402116,127773.01587301587
 &WIDTH=400
 &HEIGHT=300
 &FORMAT=image/png
 &BGCOLOR=0xffffff
 &TRANSPARENT=TRUE
 &EXCEPTIONS=XML
```

Figure 1: GetMap response of a sample WMS request

Changing any of the parameters in the example request would result in a different map. This has some advantages but also some major disadvantages and will be discussed in the next chapter about WMTS.

1.2.2 Web Map Tile Service (WMTS)

The first and latest WMTS standard was published in 2010. The standard shares many concepts with the WMS standard. The difference is very well explained in the foreword to the standard's specification [8]:

> **WMS** focuses on rendering **custom maps** and is an ideal solution for dynamic data or custom styled maps (combined with the OGC Style Layer Descriptor (SLD) standard). **WMTS** trades the flexibility of custom map rendering for the scalability possible by serving of **static** data (base maps) where the bounding box and scales have been constrained to discrete **tiles**. The fixed set of tiles allows for the implementation of a WMTS service using a web server that simply returns existing files. The fixed set of tiles also enables the use of standard network mechanisms for scalability such as distributed cache systems.

The concept of predefined tiles enables the tiles to be pre-generated and cached which speeds up map delivery and makes it a lot more scalable. The tiles are organized in zoom

levels, where each zoom level is a matrix of tiles (tile matrix). These tile matrices are themselves organized in a structure which is called tile pyramid:

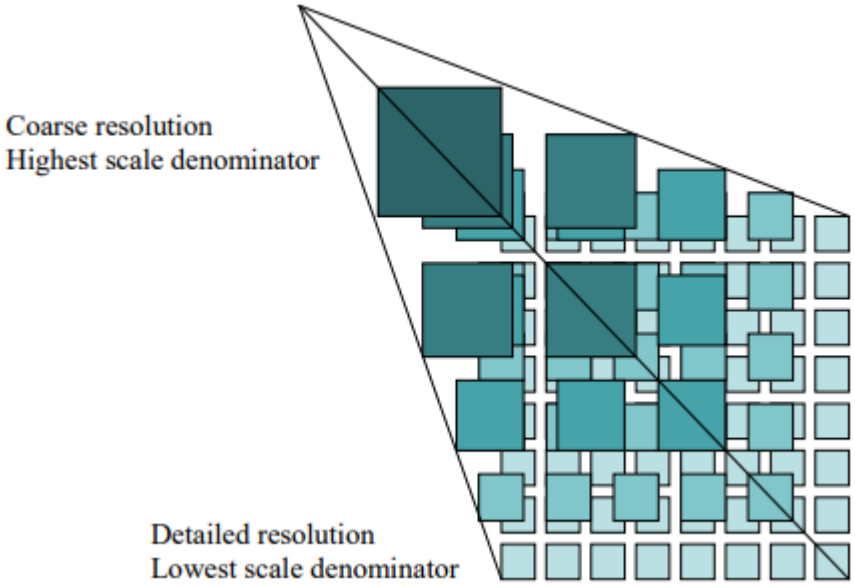

Figure 2: WMTS image tile concept [8]

For each zoom level of the map there is one tile matrix in the pyramid. These tiles can either be created on demand (just like any regular WMS request) or cached (prerendered) like mentioned in the introduction to the WMTS standard's specification [8]:

> The RESTful pattern provides the ability to set up conformant WMTS servers simply. If all the images are prerendered, a WMTS server could even be created using no image processing logic at all but relying only on a normal web server to return the static ServiceMetadata XML document and provide the image tile files. This is important for deployment purposes as many Internet service providers (especially the free ones) allow web pages and static content hosting but do not allow using CGI, ASP, or more advanced applications for security reasons. The RESTful approach therefore enables small organizations to provide geographic data using readily available services or simple web server configurations. This approach also scales dramatically since the issues of serving fixed resources in high volumes have been continuously tackled over the past decades. Finally, this approach can benefit from network scaling effects since the images are considered by the HTTP protocol to be standard web resources and network providers can leverage their existing technologies to improve the flow of those resources to requesting clients.

Since this approach is quite common among a variety of service providers such as OpenStreetMap there is a special profile only for this purpose, called “WMTS Simple”.

1.2.2.1 WMTS Simple Profile

The description of the WMTS Simple Profile by OGC is as follows [9]:

> The Web Map Tile Service (WMTS) Simple profile defines restrictions that limit the flexibility in implementing a WMTS instance. Adding additional requirements has the goal of simplifying the creation of services and clients. By implementing this profile, clients can more easily combine data coming from different services including from other WMTS instances and even from some tile implementations that are not OGC WMTS based, such as some current distributions of OSM. In fact, most of these tiling services are implicitly following most of the WMTS requirements. Many current WMTS services that implement this profile will have to undergo some changes on how tiles are exposed, and a client that is compatible with WMTS 1.0 will be immediately compatible with this profile. The aim is to align the WMTS standard to other popular tile initiatives which are less flexible but widely adopted.

Using this profile, it is not necessary to setup a fully functional mapserver – the files just need to be placed in the right folders, get requested on the client-side by the web map and are served on the server-side by the webserver (Apache, NGINX).

1.2.3 Web Coverage Service (WCS)

The latest WCS standard was published in 2012 as version 2.0.1. The introduction to the specification explains the purpose of the standard and the main difference to WMS [10]:

> A WCS provides access to coverage data in forms that are useful for client-side rendering, as input into scientific models, and for other clients. The WCS may be compared to the OGC Web Feature Service (WFS) and the Web Map Service (WMS). As WMS and WFS service instances, a WCS allows clients to choose portions of a server's information holdings based on spatial constraints and other query criteria.
> Unlike WMS, which returns spatial data to be portrayed as static maps (rendered as pictures by the server), the Web Coverage Service provides available

> data together with their detailed descriptions; defines a rich syntax for requests against these data; and returns data with its original semantics (instead of pictures) which may be interpreted, extrapolated, etc., and not just portrayed.

The crucial detail here is that WCS returns data with its original semantics that is meant to be rendered on the client. This means that the client can not only display this data but can also process this data before displaying. WCS can return a variety of different file formats, including GeoTIFF, netCDF, JPEG2000, GMLJP2 and many more.

1.2.4 OpenLayers

OpenLayers is a well-known JavaScript library for creating web maps. On their official website it is described as follows [11]:

> OpenLayers makes it easy to put a dynamic map in any web page. It can display map tiles, vector data and markers loaded from any source. OpenLayers has been developed to further the use of geographic information of all kinds. It is completely free, Open Source JavaScript, released under the 2-clause BSD License (also known as the FreeBSD).

In this thesis version 4.6.4 was used that can be downloaded from GitHub under the following link: https://github.com/openlayers/openlayers/tree/v4.6.4

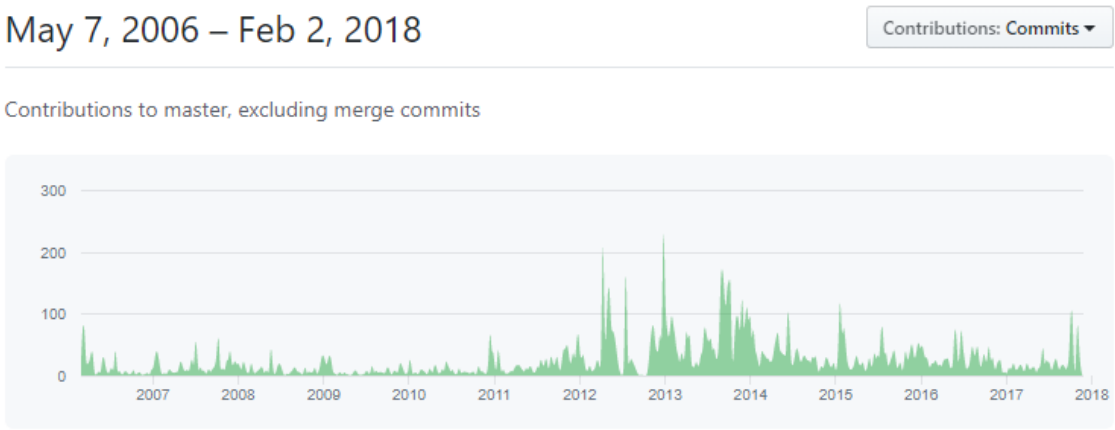

Figure 3: GitHub commits for OpenLayers

OpenLayers is hosted on GitHub since 2006 and as of writing this thesis has 22.574 commits, 174 releases and 207 contributors. Another well-known JavaScript library for

webmapping is Leaflet [12], having 6.528 commits since 2010, 38 releases (current version is 1.3.1) and 556 contributors [13]. The prototype in chapter 3 is built on OpenLayers and there was no reason for this decision other than that the company supporting this thesis is already working with OpenLayers in several applications. Though, the method used should be adoptable for other libraries like Leaflet as well.

OpenLayers supports the WMS and WMTS standards but it is not possible to display data generated by a WCS request (like GeoTIFF files). On the basic concepts page the possible layers are listed [14]:

> **Layer**
> A layer is a visual representation of data from a source. OpenLayers has three basic types of layers: ol.layer.Tile, ol.layer.Image and ol.layer.Vector.
> **ol.layer.Tile** is for layer sources that provide pre-rendered, tiled images in grids that are organized by zoom levels for specific resolutions.
> **ol.layer.Image** is for server rendered images that are available for arbitrary extents and resolutions.
> **ol.layer.Vector** is for vector data that is rendered client-side.

Displaying GeoTIFF tiles is not possible, because this is something in between those three options: GeoTIFF tiles are raster data but need to be rendered on the client-side.

1.2.5 Geotiff.js

geotiff.js is a JavaScript library for reading geospatial metadata and raw array data from a wide variety of different GeoTIFF file types directly in the browser. It is hosted on GitHub since 2015, has 197 commits in 17 releases, 6 contributors and is released under the MIT licence [15].

Figure 4: GitHub insights for geotiff.js

1.2.6 Plotty

Plotty is a JavaScript library for helping plot 2D data into a HTML5 canvas element using WebGL or JavaScript as a fall-back. It was published on GitHub under MIT license in 2015 and has had 61 commits, 12 versions and two contributors since then [16].

Figure 5: GitHub insights for plotty

1.3 Research Questions

The previous section showed state of the art regulations and technology and leads to the following research questions:

- Is it possible to combine the principles of WCS (providing subsets of the original data with original semantics) and WMTS (providing the data as tile pyramids for fast and efficient transmission to the client) using existing open source solutions?
- In which scenarios could the user benefit of using this technique regarding performance of the web map (response timings and data transfer)?
- Which are the main factors influencing the results?
- What are the benefits (opportunities) or drawbacks (limitations) of such an approach?

2 Method

The research was done in a hands-on manner implementing a web mapping prototype that is built on existing open source software and combines the essential parts to one new concept. Results were collected by defining important key performance indicators, measuring them by implementing a benchmarking setup and comparing them to existing workflows.

It is key to better understand the characteristics and differences of server- and client-side processing of web maps. Therefore, these different concepts are explained in the following sections to give further important background for explaining what has been achieved within the thesis.

2.1 Server- and Client-side Processing of Web Maps

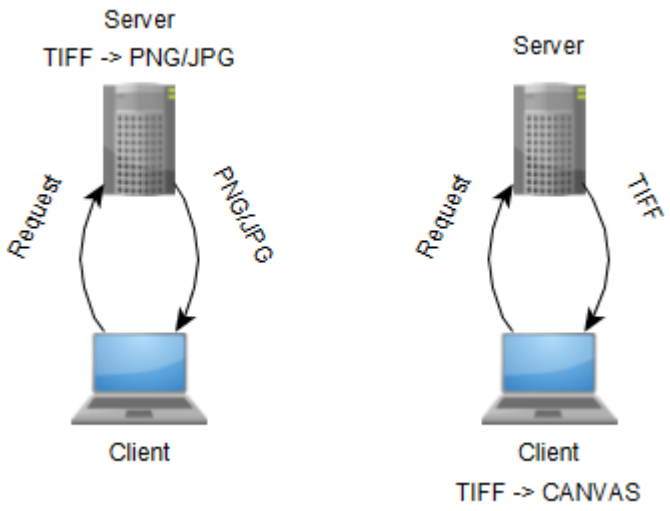

Figure 6: Basic illustration of server-side and client-side processing

Figure 6 shows the main difference between server-side and client-side processing in a simplified illustration. The important part is that on the left side (server-side processing) the client sends the request to a server and gets a PNG/JPG file back from that server. All the processing is done on the server-side. On the right side, the server returns a GeoTIFF file that is parsed and rendered on the client. Those two examples will be discussed in detail in the following two chapters.

2.1.1 Server-side Processing

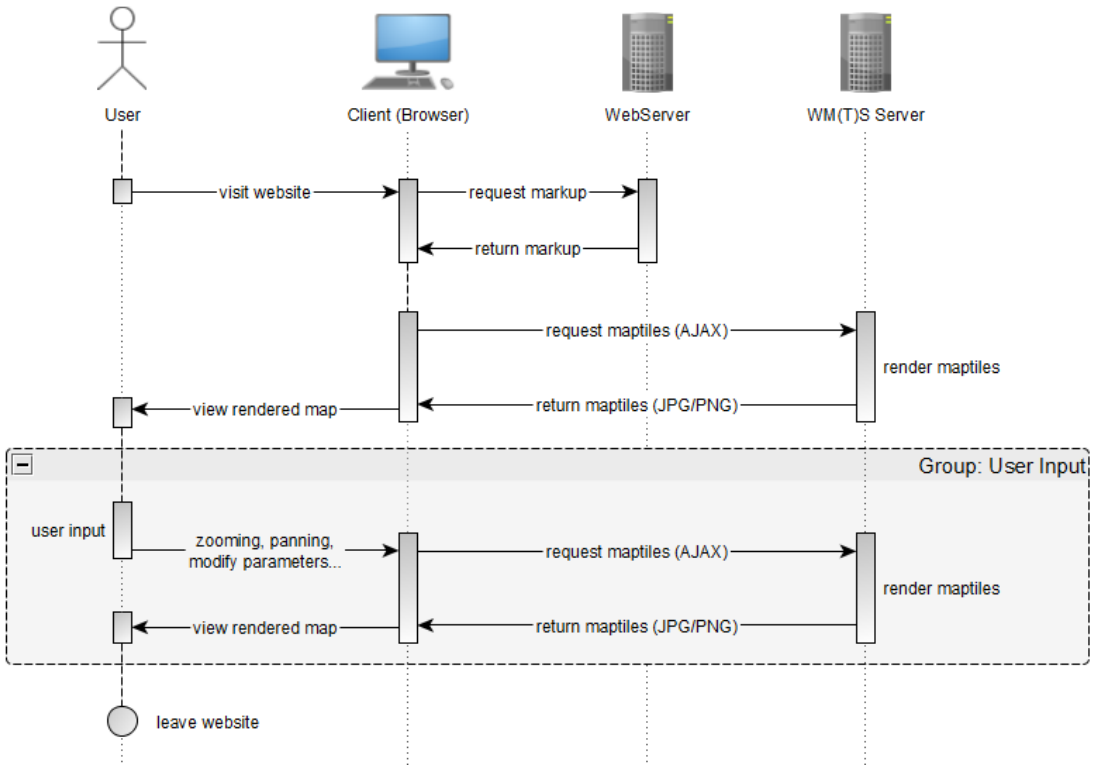

Figure 7: Sequence-diagram of server-side processing

This example of server-side processing is what happens on any regular WMS or WMTS request. A user visits a website with a web-map on it. First, the client requests the HTML markup of the requested site and returns it to the client. Then the web-map starts requesting all tiles of the current view through several AJAX requests from the WMTS server that is defined as the layer's source. The webserver and the WMTS server can technically be the same server, but in the diagram we keep them separate to have a better visualisation of what is going on where. The requested tiles get processed and rendered on the server-side and are returned as regular web images like JPG or PNG. The client's browser then only has to display these images.

This process happens on every single user interaction. Every pan, every zoom and also every modification of parameters that influence the final rendering leads to a new request to the server. This repeatable nature is illustrated by grouping those actions as "Group: User Input" in the diagram.

2.1.2 Client-side Processing

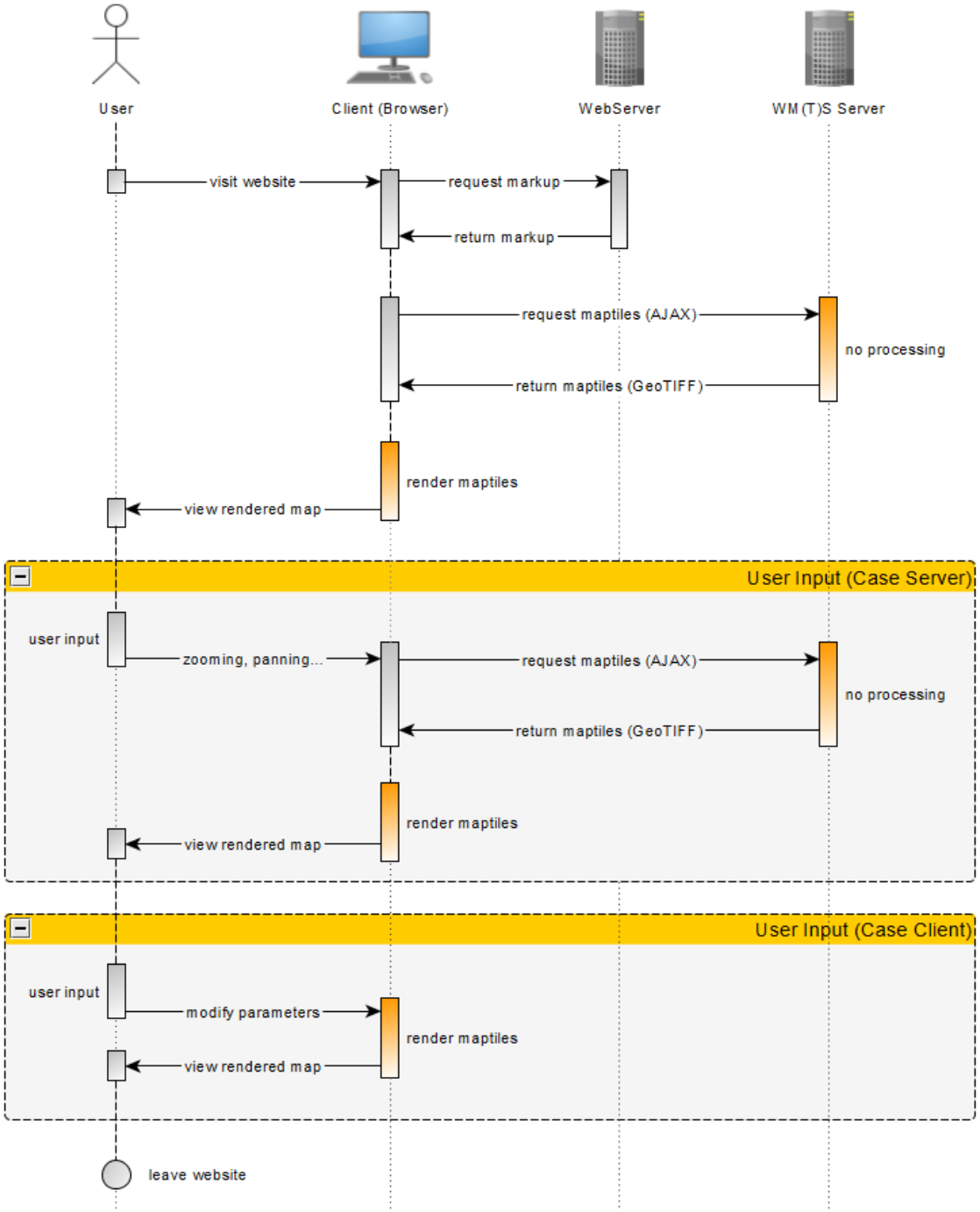

Figure 8: Sequence-diagram of client-side processing

Client-side processing is already possible for vector data but not for raster data. This thesis extends this concept and applies it to rendering of raster data in form of GeoTIFF as one possible result of a WCS request. This is a new concept and therefore not part of any standard. The principle is shown in Figure 8 and it is similar to the illustration in the previous chapter. The differences are highlighted in orange:

1. **There is no processing on the server-side.**

   Since the processing will be done on the client, the server does not have to do any processing for the requested tiles. Of course, this is only the case when the requested tiles do already exist as pre-processed GeoTIFF files on the server. Applying this caching technique on the server-side is only possible for client-side processing, because at server-side processing the server has to return tiles based on the user settings and therefore the cache would grow extremely large and would presumably make no sense.

2. **Tiles are rendered on the client and not on the server.**

   Since browsers don't allow to display GeoTIFF files directly and OpenLayers also has no "GeoTIFF support", it is necessary to take an extra step to parse the file and render it as an image based on the user's parameter settings. This is shown as "render maptiles" in the illustration.

3. **Two different cases of user input.**

   Other than in the previous case of server-side processing two different cases for user input can be distinguished:

   a. *User input that needs to request new data from the server.*

      Whenever the user zooms or pans to an area that he has not viewed before, the web-map library will request new data from the server. This is illustrated as "User Input (Case Server)" in the diagram.

   b. *User input that does NOT need to request new data from the server.*

      Whenever the user only modifies parameters that change the visual representation of the data but do not change the underlying data itself (for example changing the colour palette but not changing the map view or zoom), the web-map will *not* request any new data from the server and all the rendering will be done on the client-side. This is illustrated as "User Input (Case Client)" in the diagram.

2.2 Benchmarking

To be able to evaluate the new method key metrics are defined and afterwards compared for both methods. The goal of the new method is to improve the user experience of interacting with the web-map, so the user is our main target for the evaluation. This is important to mention, because if the goal was to decrease the load on the server, the focus would be totally different and therefore it would be necessary to define totally different key performance indicators (KPIs).

2.2.1 Key Performance Indicators

The main KPIs for this thesis – focusing on the client's user experience – are listed here and shown in two illustrations on the following pages:

- Request time = RT

  This describes the time needed from the user input until the map is rendered and fully displayed.

- **Overall request time = ORT**

  This describes the overall time needed for all user interactions and is the most important measurement for this thesis, because the user is our focus and the time that is needed for waiting should be as low as possible.

$$\mathrm{ORT} = \sum RT$$

- Data transferred = DT

  This describes the amount of data that is transferred from the server to the client during one user interaction.

- **Overall Data Transferred = ODT**

  This describes the amount of data that is transferred from the server to the client over all user interactions.

$$\mathrm{ODT} = \sum DT$$

These KPIs are illustrated in the following figure:

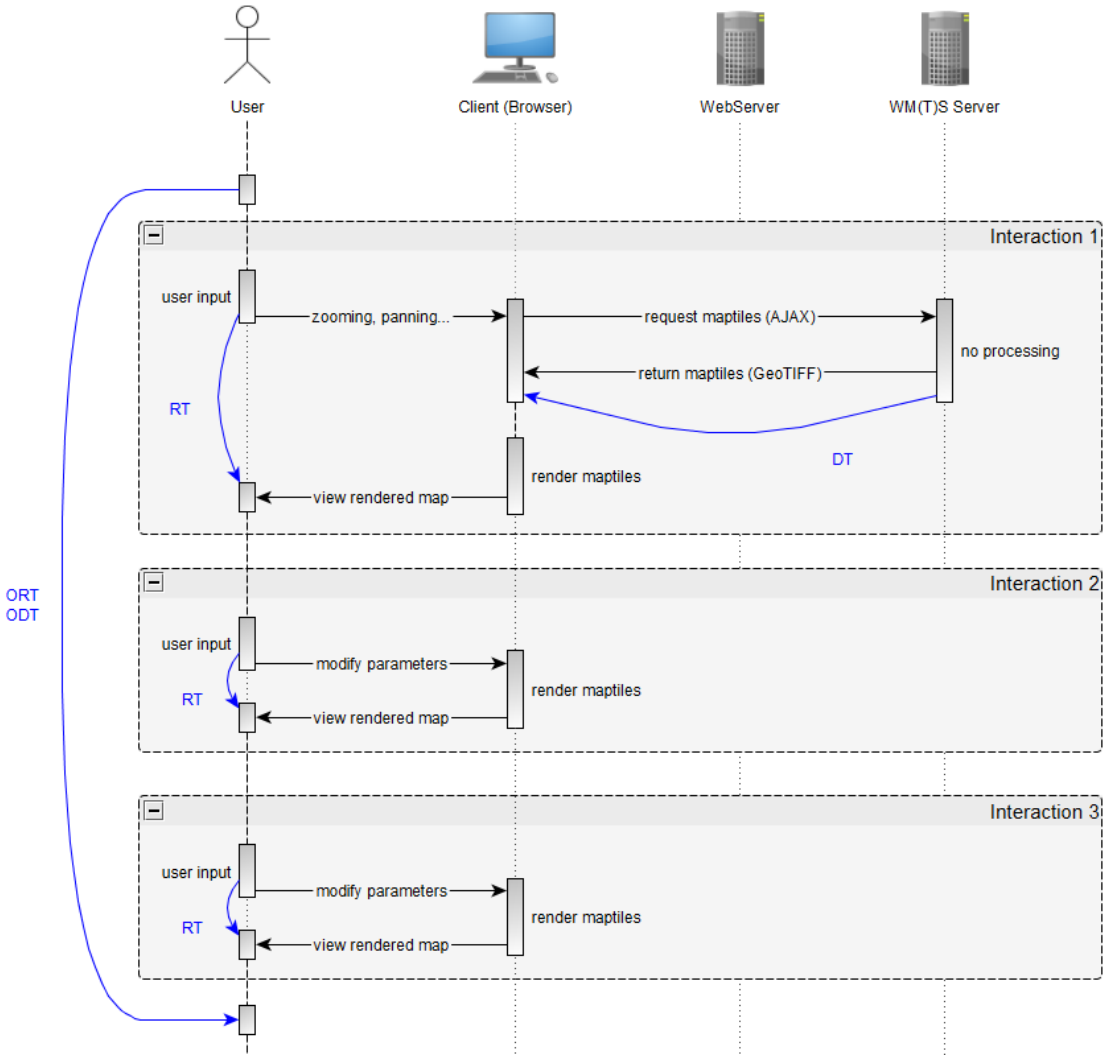

Figure 9: Main KPIs (client-side example)

The results in this example would be:

$$\text{ORT} = RT(Interaction1) + RT(Interaction2) + RT(Interaction3)$$

$$ODT = DT(Interaction1)$$

It is obvious that the ORT heavily depends on the number of user interactions. Since there is only one request that fetches data from the server the ODT equals the DT of user interaction 1. In interaction 2 and 3 the data is processed on the client so the ODT stays unchanged.

2.2.2 Variables Influencing the KPIs

Of course, the final results for the defined KPIs will heavily depend on the use case (user interactions) and also on the (hardware) setup of the whole application. Three main factors can be identified, illustrated in two figures and explained on the next page:

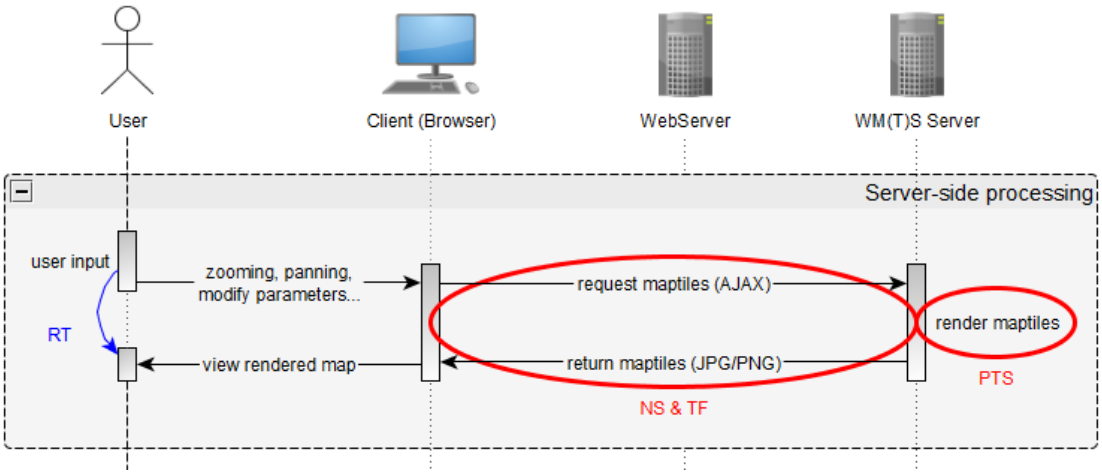

Figure 10: Variables Influencing the KPIs (Server-side Processing)

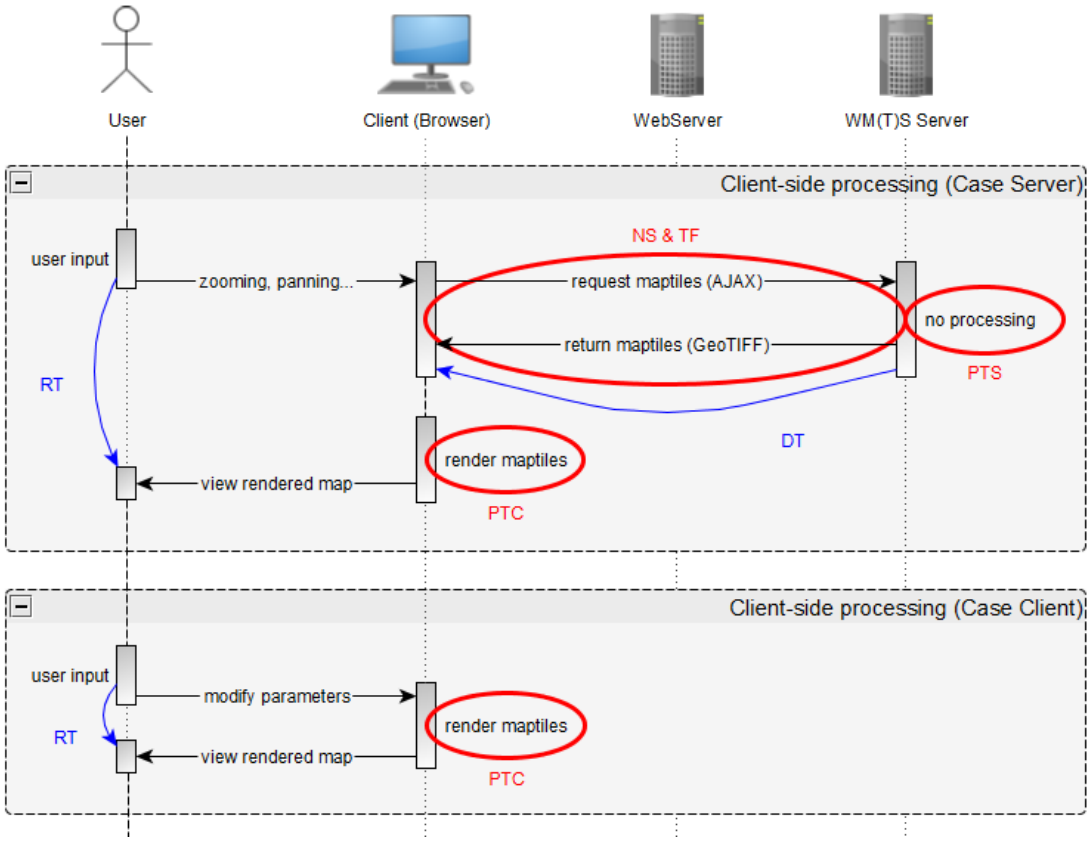

Figure 11: Variables influencing the KPIs (client-side processing)

The Variables are:

2.2.2.1 Network Speed (NS)

The slower the network the longer it will take to download data from the server to the client. This will affect every user interaction at server-side processing and all user interaction that fetch data from the server at client-side processing (“Case Server”).

Measurement: The network speed can easily be throttled through the browsers’ dev-tools. It is also easy to calculate the network speed by dividing the amount of data transferred by the time needed.

Effect: The slower the NS, the higher will be the RT.

2.2.2.2 Tile Filesize (TF)

The tile filesize will only vary between the two methods but be almost constant for each single tile of one methods. Tiles at client-side processing are GeoTIFF tiles and therefore usually a lot larger than JPG/PNG tiles at server-side processing.

Measurement: The tile filesize can easily be analysed by the operating system or by inspecting available web services via the browsers’ dev-tools.

Effect: The larger the tile filesize, the higher will be the RT.

2.2.2.3 Processing Time on the Server (PTS)

The processing time on the server is the time needed from receiving the request on the server until sending the response with the rendered JPG/PNG tile. This timeframe is a huge unknown and depends a lot on the server setup.

Measurement: This variable will not be measured. Instead the benchmarks will be done with different values and recommendations and findings will be discussed.

Effect: The longer it takes the server to render the tiles, the higher will be the RT.

2.2.2.4 Processing Time on the Client (PTC)

The PTC will depend on the client computer and browser. This variable is divided in two parts: First, the time needed for parsing the GeoTIFF and second, the time needed for plotting the date to the canvas. Parsing is only done once for each tile while rendering is done on any user input that changes the parameters.

Measurement: Run benchmarks from different computers and browsers.

Effect: The longer it takes the client to render the tiles, the higher will be the RT.

2.2.2.5 Number of Tiles (NOT)

Many variables occur on every tile load and therefore need to be multiplied by the number of tiles that were requested. The larger the map, the more tiles get requested, the bigger the impact of the described variables and KPIs.

The number of tiles requested does not only depend on the size of the map, but also on the user interaction itself. A user panning a 4x4 map to the side (meaning 4 tiles horizontally and 4 tiles vertically) could mean a request of 4 new tiles if the map was only slightly panned but could also mean a request of 16 or more tiles if the map was panned far to that direction.

2.2.2.6 ORT for Server-side Processing

In case of server-side processing the calculation is quite simple. ORT was defined as follows:

$$\mathrm{ORT} = \sum RT$$

Knowing all the variables for RT we can modify the formula:

$$\mathrm{RT} = NOT \cdot \left(\frac{TF}{NS} + PTS\right)$$

Basic example: A user requesting a map, each tile having 300kB, network speed 10.000kB/s and PTS as constant throughout this chapter of 50ms. The user does 3 interactions:

1. Display map (16 tiles)
2. Zoom in (16 tiles)

3. Three times: Modify parameters

$$\mathrm{RT}_1 = 16 \cdot \left(\frac{300}{10000} + 0{,}005\right)$$

$$\mathrm{RT}_2 = 16 \cdot \left(\frac{300}{10000} + 0{,}005\right)$$

$$\mathrm{RT}_3 = 3 \cdot 16 \cdot \left(\frac{300}{10000} + 0{,}005\right)$$

$$\boldsymbol{ORT_S = 0,56 + 0,56 + 1,68 = 2,8 \text{ seconds}}$$

Interaction	RT	ORT
RT1	0,56	0,56
RT2	0,56	1,12
RT3a	0,56	1,68
RT3b	0,56	2,24
RT3c	0,56	2,8

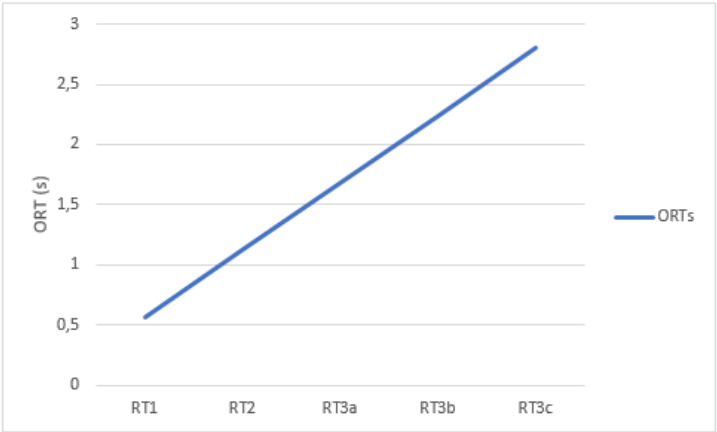

Figure 12: ORTs (server-side example)

2.2.2.7 ORT for Client-side Processing

The same example for client-side processing would be almost the same but resulting in larger tilesize (here assumed as 600kB per tile) and needing additional client-side processing time PTC (here assumed as 20ms per tile) for user interaction 1 and 2 (called RTs in the example, S standing for Server), but needing no network communication for user interactions 3 (called RTc in the example, C standing for Client):

$$\mathrm{RT_S} = \mathrm{NOT} \cdot \left(\frac{\mathrm{TF}}{\mathrm{NS}} + \mathrm{PTS} + \mathrm{PTC}\right) \qquad \mathrm{RT_C} = \mathrm{NOT} \cdot \mathrm{PTC}$$

$$\mathrm{RT}_1 = 16 \cdot \left(\frac{600}{10000} + 0{,}005 + 0{,}002\right)$$

$$\mathrm{RT}_2 = 16 \cdot \left(\frac{600}{10000} + 0{,}005 + 0{,}002\right)$$

$$\mathrm{RT}_3 = 3 \cdot 16 \cdot 0{,}002$$

$$\boldsymbol{ORT_C = 1,072 + 1,072 + 0,096 = 2,24 \text{ seconds}}$$

Interaction	RT	ORT
RT1	1,072	1,072
RT2	1,072	2,144
RT3a	0,032	2,176
RT3b	0,032	2,208
RT3c	0,032	2,24

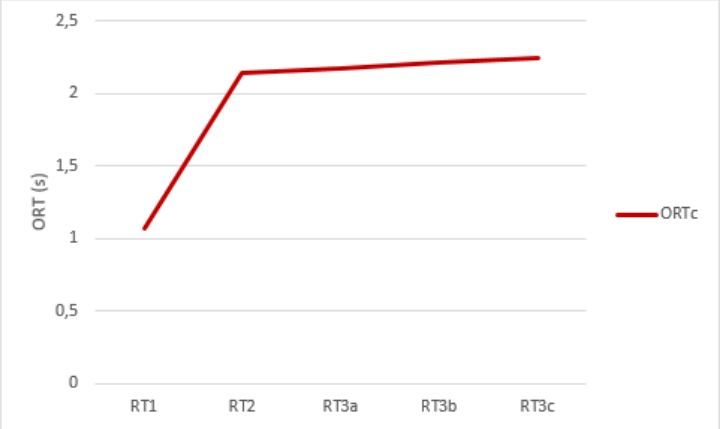

Figure 13: ORTc (client-side example)

This result shows that the overall request time (after all interactions) at client-side processing is slightly lower than at server-side processing:

Interaction	RT	ORTc
RT1	1,072	1,072
RT2	1,072	2,144
RT3a	0,032	2,176
RT3b	0,032	2,208
RT3c	0,032	2,24

Interaction	RT	ORTs
RT1	0,56	0,56
RT2	0,56	1,12
RT3a	0,56	1,68
RT3b	0,56	2,24
RT3c	0,56	2,8

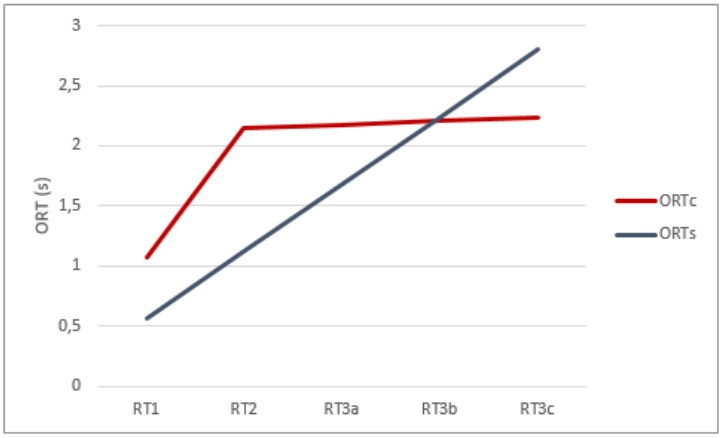

Figure 14: Break-even (ORT)

2.2.2.8 Break-Even (ODT)

The calculation of the ODT is even simpler and also shows a benefit for the client-side processing (remember though that this is fictional data and only shows the method of what we are going to do):

$$\mathbf{ODT_S} = 16 \cdot 300 + 16 \cdot 300 + 3 \cdot 16 \cdot 300 = \mathbf{24.000kB}$$

$$\mathbf{ODT_C} = 16 \cdot 600 + 16 \cdot 600 + 0 = \mathbf{19.200kB}$$

Interaction	DT	ODTc
RT1	9600	9600
RT2	9600	19200
RT3a	0	19200
RT3b	0	19200
RT3c	0	19200

Interaction	DT	ODTs
RT1	4800	4800
RT2	4800	9600
RT3a	4800	14400
RT3b	4800	19200
RT3c	4800	24000

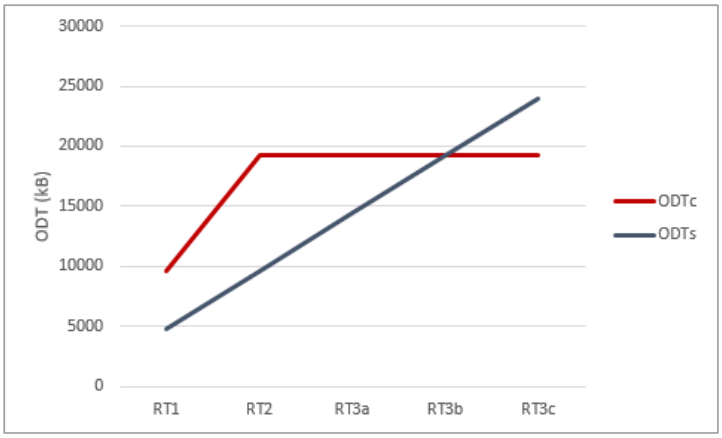

Figure 15: Break-even (ODT)

3 The Prototype

The prototype will build upon existing technologies shown in chapter 1.2 and extend the lacking part with a new library called “olGeoTiff” – a JavaScript class that modifies the behaviour of OpenLayers so it can display GeoTIFF files directly on the web-map in the browser.

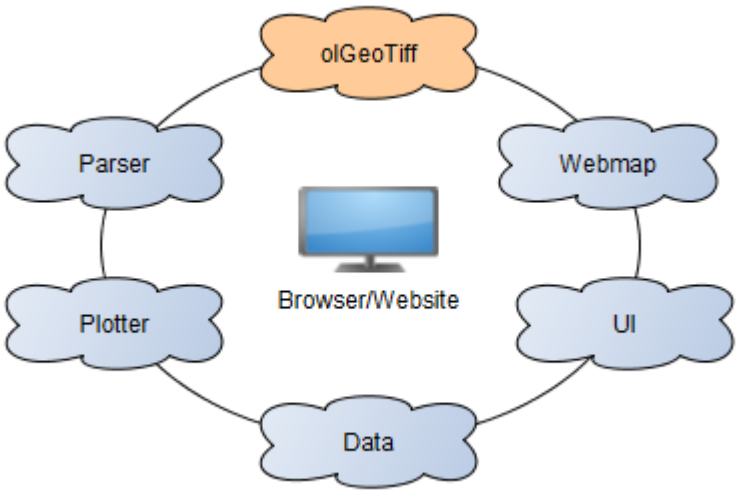

Figure 16: Components of the prototype

3.1 Use Case: Visualisation of Multiband Sentinel-2 Satellite Data

For the prototype and the benchmarking chapter a use case will be analysed that shows multiband satellite data of a Sentinel-2 pre-processed as GeoTIFF files [17]. The user will have the possibilities to zoom and pan around the map and to choose between different colour palettes and to change parameters for the visualization on the fly.

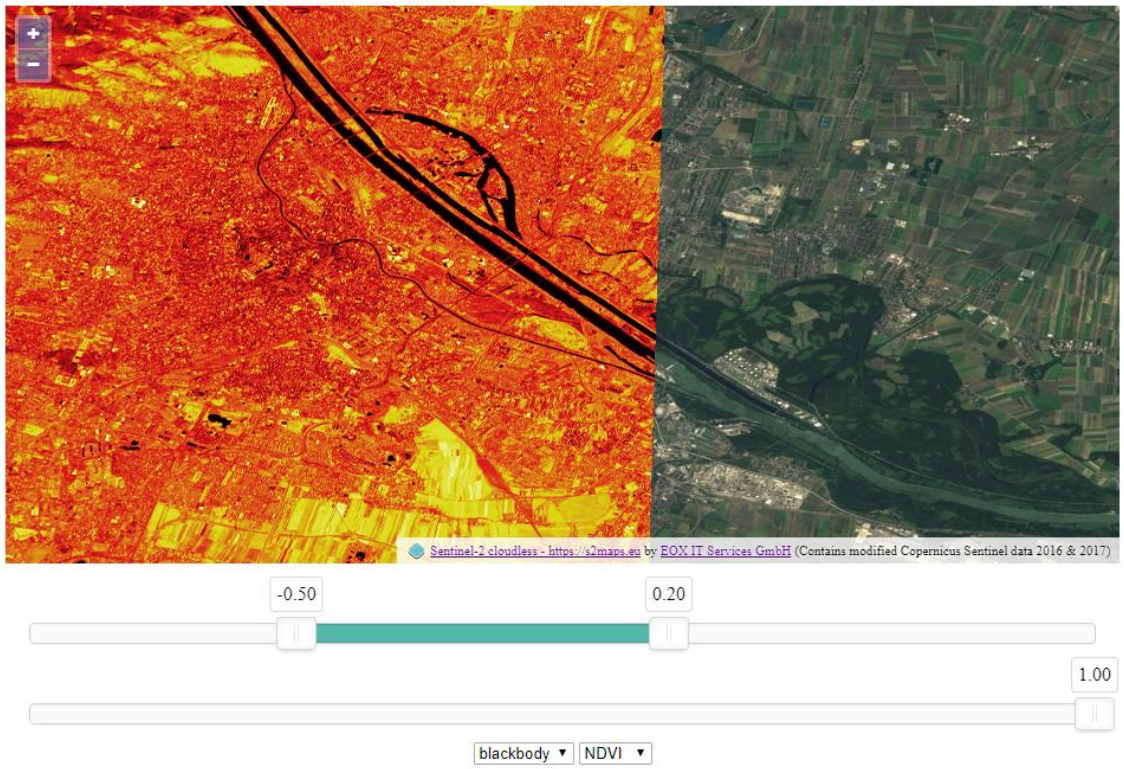

Figure 17: Sentinel-2 data rendered on an OpenLayers web-map

The map shows the S2-cloudless layer by EOX IT Services GmbH as baselayer [18] and as an overlay it shows an NDVI (Normalized Difference Vegetation Index) [19] calculated from our sample data and rendered with colour palette “blackbody”.

As seen in the figure above, the user has several ways to interact with the map:

- There is the web-map itself for panning and zooming.
- Underneath the map there are two sliders, one for setting the domain values for the colour palette and one for setting the opacity of the overlay layer.
- Underneath the sliders there are two select fields to change the colour palette and the type of index that gets calculated.

3.2 Sample Data

The sample data used for this thesis is Sentinel-2 Level 1C open data provided by the European Space Agency and the European Commission's Copernicus programme [20]:

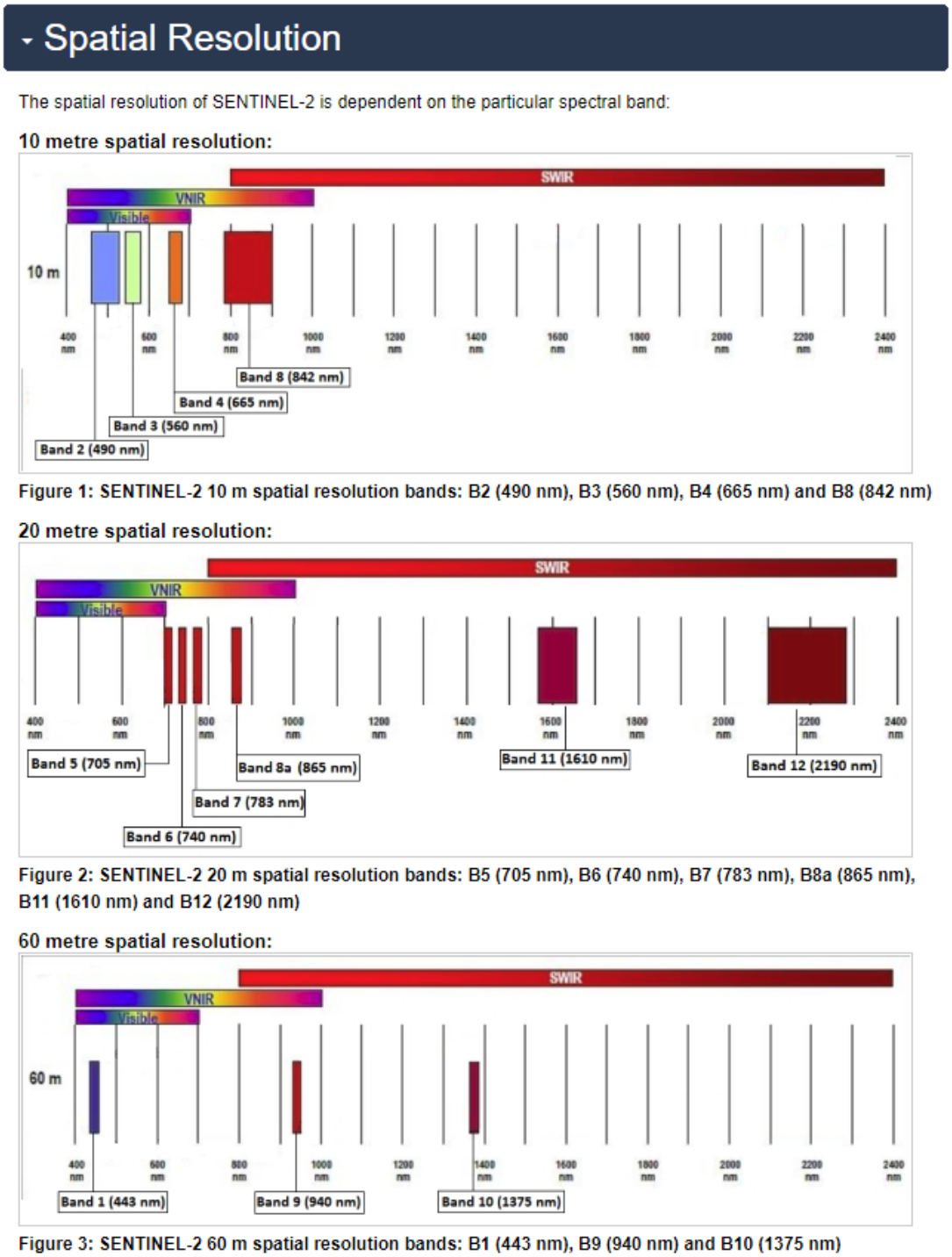

Figure 18: Spatial resolutions of Sentinal-2 spectral bands

This figure shows that Sentinel-2 data has 12 available bands at different resolutions. For our prototype we use the bands with the highest possible resolution of 10 meters: RGB – Red (Band 4), Green (Band 3), Blue (Band 2) and Near Infrared (Band 8).

This data was processed by EOX IT Services GmbH using mapchete [21] and creating GeoTIFF tile pyramids that can be requested just like PNG/JPG tile pyramids of a regular WMTS (Simple) request. The tilesize of the GeoTIFF tiles is heavily dependent on the number of bands that are stored in the tif and also the bit-depth of the data. The sample data used is a 16 bit GeoTIFF and has four different bands. The average tilesize of this dataset is around 346kB.

3.3 Implementation

3.3.1 Architecture

Referencing the sequence diagram from chapter 2.1.2, Figure 19 shows where and when the components do their jobs:

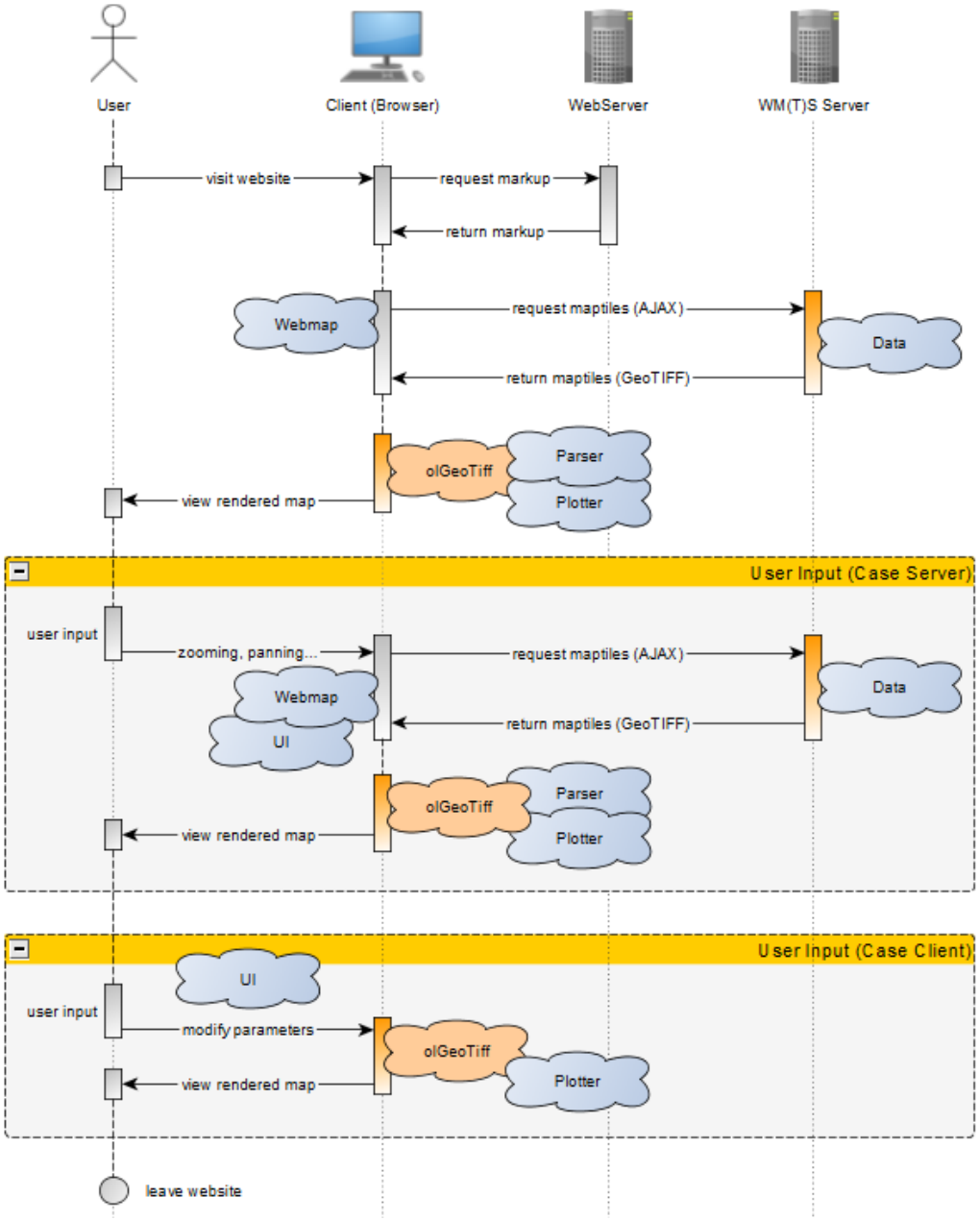

Figure 19: Sequence diagram including the prototype components

It starts on the client side when the browser initialises the OpenLayers **web-map**. It then requests the map tiles (**data**) via several AJAX requests from the WM(T)S Server. In the

prototype setup this is actually not a WMTS server but a regular Apache webserver using the WMTS simple profile (see chapter 1.2.2.1) to return the pre-processed tiles.

The returned GeoTIFF file then has to be **parsed** by geotiff.js and rendered by **plotty**. The newly developed class **olGeoTiff** handles this process. This sequence is the same for all user inputs where tiles are requested from the server. When the user only changes some parameters, the data does not need to be requested from the server. This case is shown in the figure as group "**User Input (Case Client)**". The difference to the upper group is that the user does not interact with the web-map, it only interacts with the parameter settings of the plot (e.g. colour palette, domain limit values, …), illustrated as **UI** in the diagram. There is absolutely no interaction with the server (everything happening in the browser) and therefore the data does not have to be parsed any more. Only the rendering has to be done via plotty.

3.3.2 Prototype Code

All code examples of this thesis can be downloaded from GitLab [22]. The basic HTML is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>olGeoTiff Multiband Example</title>

    <!-- scripts & styles -->
  </head>
  <body>
    <div class="mapcontainer">
      <!-- map -->
      <!-- ui -->
    </div>

    <script>
    // set variables
    // create matrix
    // define the wmts layer
    // eox baselayer
    // define the map
    // setup datafunctions
    // olGeoTiff setup
    // handle user input
    </script>
  </body>
</html>
```

Figure 20: Basic HTML structure

First, all dependencies and stylings are added to the head, then the map and UI markup is added to the body and finally the map is initialised via some JavaScript that also handles the user input.

```
3   <head>
4     <meta charset="utf-8"/>
5     <title>olGeoTiff Multiband Example</title>
6
7     <!-- scripts & styles -->
8
9     <script src="dist/plotty.min.js"></script>
10    <script src="dist/nouislider.js"></script>
11    <script src="dist/geotiffjs/geotiff.browserify.js"></script>
12    <script src="dist/ol-debug.js"></script>
13    <script src="olGeoTiff.js"></script>
14    <script src="dist/jquery.min.js"></script>
15
16    <link rel="stylesheet" href="dist/nouislider.css" type="text/css">
17    <link rel="stylesheet" href="dist/ol-debug.css" type="text/css">
18    <style>
19      .slider { width: 100%; margin-top: 50px; }
20      .map { width: 100%; height: 80%; margin-top: 20px; }
21      select { margin-top: 15px; }
22      .mapcontainer { padding: 0; text-align: center; }
23      .slidercontainer { padding: 0 20px; }
24    </style>
25  </head>
```

Figure 21: HEAD of the prototype

As shown in Figure 16 plotty is included for rendering the tiles (line 9), nouislider for handling and styling the sliders for user input (lines 10 and 16), geotiff.js for parsing the GeoTIFFs (line 11), OpenLayers for the web-map (lines 12 and 17), jQuery for easier prototyping (line 14), some simple style instructions (lines 19 to 23) and finally olGeoTiff.js (line 13) that makes everything play together.

The HTML for the user inputs is kept very simple:

```
28  <body>
29    <div class="mapcontainer">
30      <!-- map -->
31      <div id="s2map" class="map"></div>
32
33      <!-- ui -->
34      <div class="slidercontainer">
35      <div class="slider domainslider"></div>
36      <div class="slider opacityslider"></div>
37      </div>
38      <select class="palette">…
65      </select>
66      <select class="datafunction">…
69      </select>
70    </div>
```

Figure 22: Map and user input

The map is a single div element and the UI elements are two DIVs that are rendered as sliders via the nouislider library and two SELECT fields with several options for colour palette and the dataFunction that defines what value is calculated from the multiple bands in the GeoTIFFs.

The plugin code itself is discussed in the next section. The setup of the web-map is almost a basic OpenLayers setup and there are only a few things to mention:

```
86    // define the wmts layer
87    var s2layer = new ol.layer.Tile({
88      source: new ol.source.WMTS({
89        url: '/wmts_simple/s2/{TileMatrix}/{TileRow}/{TileCol}.tif',
90        projection,
91        tileGrid: new ol.tilegrid.WMTS({
92          origin: ol.extent.getTopLeft(projectionExtent),
93          resolutions,
94          matrixIds,
95        }),
96        requestEncoding: 'REST',
97        transition: 0,
98      })
99    });
```

Figure 23: s2layer – WMTS source for the OpenLayers map.

s2layer is a regular OpenLayers WMTS source [23], but in this case (line 89) a GeoTIFF is requested instead of a PNG or JPG. Also, a relative URL is used, pointing to a local folder "wmts_simple". This means that the "WM(T)S Server" in Figure 19 is actually not a WMTS Server but a regular webserver returning prerendered GeoTiff tiles. The "WMTS Server" and the "WebServer" in that figure are actually the same machine. This makes it easier to develop and test all the client related work, but it is something that needs to be taken account for in the benchmark chapter, where the impact of network speed and transfer sizes is analysed.

The baselayer seen in the figure is a regular WMTS layer hosted online at s2maps-tiles.eu and removed for the benchmarks not to sophisticate any results.

3.3.3 The olGeoTiff Plugin

For combining all the existing tools and components a JavaScript class was developed. The initialisation of the plugin is simple. After setting up the OpenLayers map and its layer holding the GeoTIFF data, the plugin needs to be initialised like this:

```
// olGeoTiff setup
  var olgt = new olGeoTiff(s2layer);
  olgt.plotOptions.domain = [-0.5, 0.2];
  olgt.plotOptions.noDataValue = 10;
  olgt.plotOptions.palette = 'blackbody';
  olgt.plotOptions.dataFunction = datafunctions['NDVI'];
```

Figure 24: olGeoTiff initialisation

The code is quite self-explaining: First, the class is initialised on the s2layer that was defined some lines before. Next, the domain limits are set: In this case a lower limit of minus 0,5 and an upper limit of 0,2. The noDataValue is defined as 10 (resulting in transparent areas if the value at this pixel is 10) and the colour palette is set to “blackbody”. Finally, the NDVI function that was also defined some lines before is assigned to the plugin as its “dataFunction”.

```
var datafunctions = {};
datafunctions['NDVI'] = function(b) {
  if(b[0]+b[1]+b[2]+b[3]==0) return 10; // return 10 as nodata value

  var green = b[1];
  var blue = b[2];
  var nir = b[3];
  return ( nir - (green + blue ) ) / ( nir + ( green + blue ) ); // otherwise return NDVI
};
datafunctions['GLI'] = function(b) {
  if(b[0]+b[1]+b[2]+b[3]==0) return 10; // return 10 as nodata value

  var red = b[0];
  var green = b[1];
  var blue = b[2];
  return ( 2*green - red - blue ) / ( 2*green + red + blue ); // otherwise return GLI
};
```

Figure 25: Calculation of different indices using “dataFunctions”

The datafunction is responsible for the client-side rendering process. Here, an object containing two datafunctions is created. First, a Green-Red NDVI [24] and second, an Green Leaf Index (GLI) [25]. In both cases 10 is returned if there is no data available. This value matches the “noDataValue” setting shown in the previous figure so that the resulting plot will be transparent at those places.

Another thing to mention is that the number of bands does NOT correlate with the original Sentinel band number (Figure 18), because the sample data contains only a subset of the bands available and those bands are referenced by an incremental array index (in this case 0 to 3):

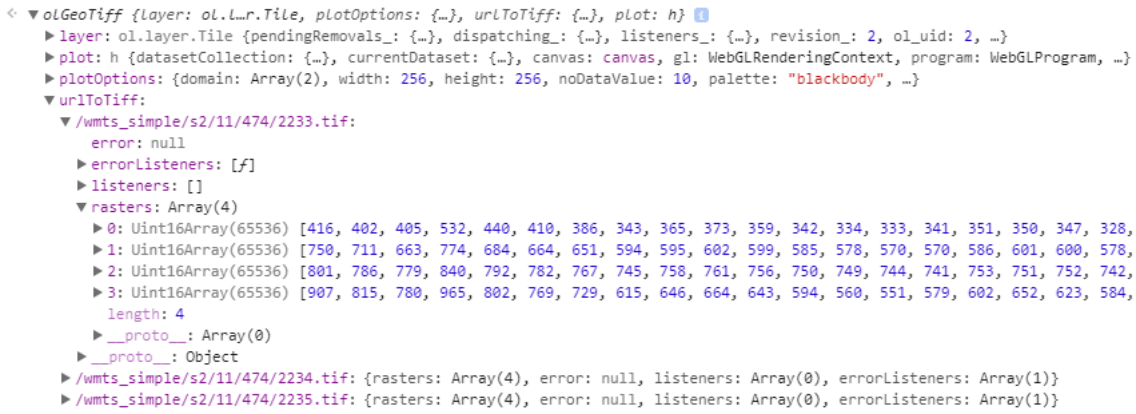

Figure 26: Inspecting olGeoTiff using Chrome dev-tools

olGeoTiff.urlToTiff holds all the data that was already downloaded from the server and parsed via geotiff.js. Each tile is referenced by its URL and the different bands are stored in the “rasters” array.

The next figure shows the code of the Plugin itself:

```
1   // olGeoTiff class
2
3   /**…
7   function olGeoTiff(layer) {…
63  }
64
65  /**…
71  olGeoTiff.prototype.fetchTiff = function(url, listener, errorListener) {…
123 }
124
125 /**…
130 olGeoTiff.prototype.tileLoadFunction = function(imageTile, src) {…
182 };
183
184 /**…
187 olGeoTiff.prototype.redraw = function() {…
189 }
```

Figure 27: Basic structure of the olGeoTiff plugin

On initialisation of the class (line 7) some default options are set. The most important part is the method “tileLoadFunction” (line 130): This function is set at the very end of class initialisation as a replacement of the default tileLoadFunction that ships with OpenLayers:

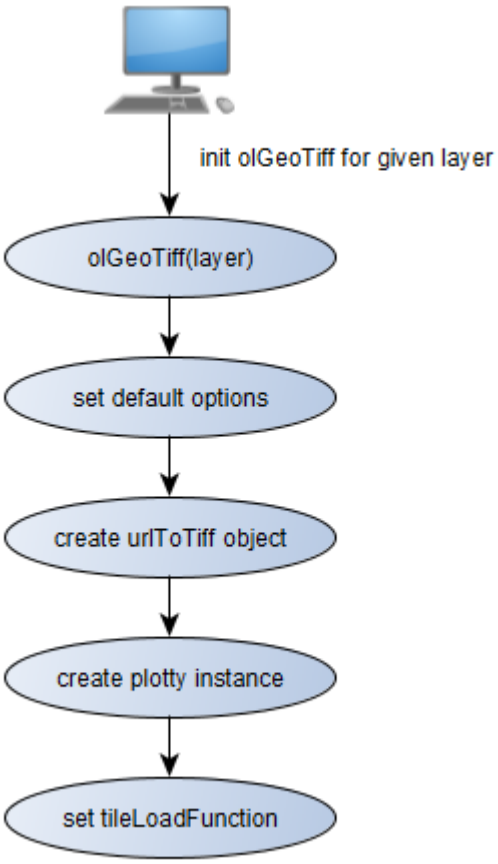

Figure 28: Initialisation of olGeoTiff

The default tileLoadFunction is very simple because the WMTS usually requests regular PNG/JPG images and just sets the SRC attribute of the image tile [26]:

```
368  /**
369   * @param {ol.ImageTile} imageTile Image tile.
370   * @param {string} src Source.
371   */
372  ol.source.TileImage.defaultTileLoadFunction = function(imageTile, src) {
373    imageTile.getImage().src = src;
374  };
```

Figure 29: Default OpenLayers tileLoadFunction

The new concept requests GeoTIFF tiles and therefore needs to take some extra steps shown in the following code example:

```
olGeoTiff.prototype.tileLoadFunction = function(imageTile, src) {
  // replace the imageTile with a canvas

  // fetch data of this tile
  this.fetchTiff(
    src, // url of tile

    // callback function that executes when the tiff is parsed and ready
    function(urlToTiff) {
      // get plotty instance
      // set plotty settings
      // render plot and trigger load event
    }.bind(this),

    // callback function in case of AJAX error
    function(error) {
      imageCanvas.dispatchEvent(new Event('error')); // trigger error event
    }
  );
};
```

Figure 30: olGeoTiff tileLoadFunction code

This tileLoadFunction is executed on every tile load and creates a canvas element and calls the fetchTiff method. This method takes care of drawing the TIFF to the web-map and works as shown in this diagram:

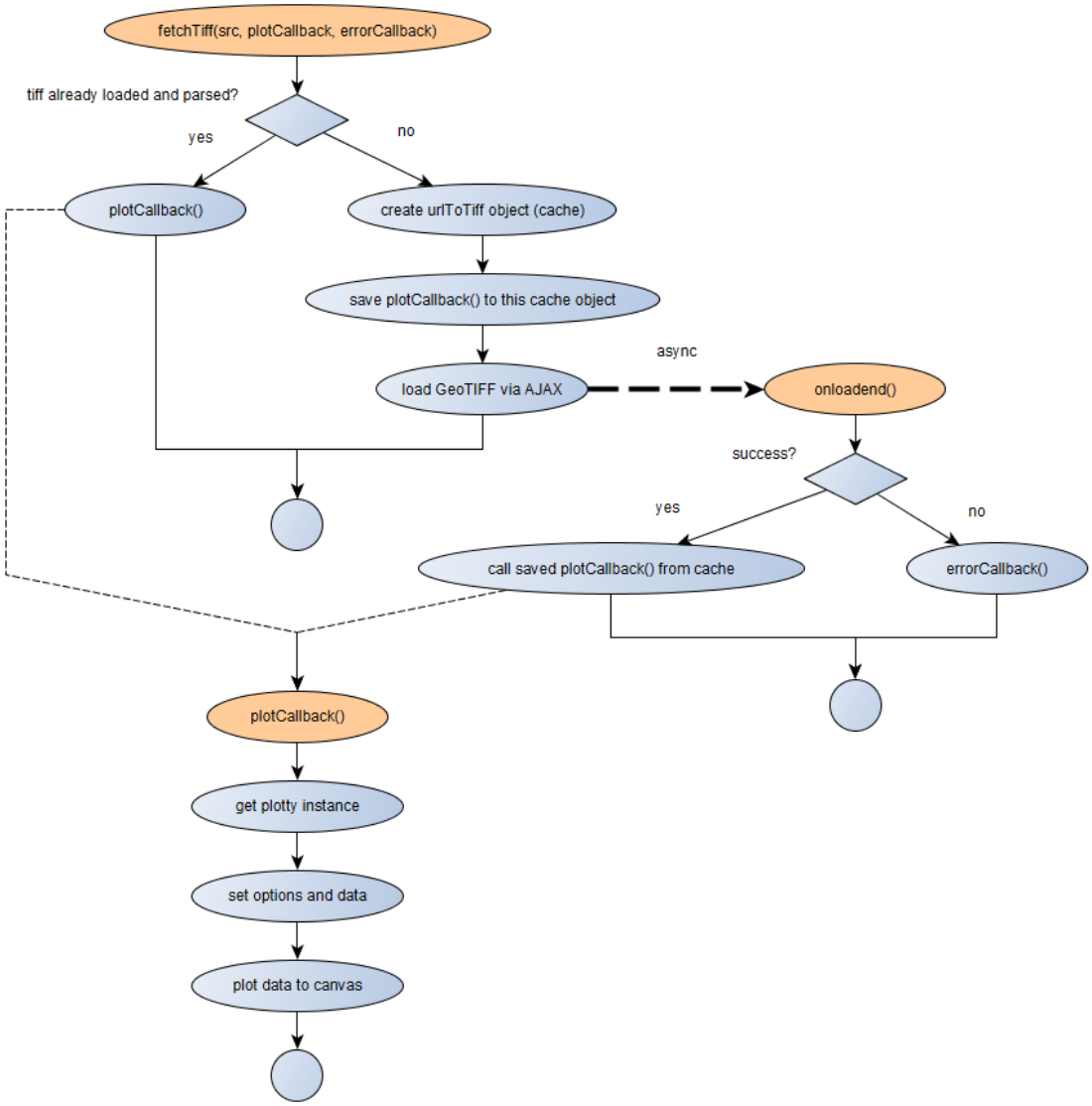

Figure 31: Illustration of the olGeoTiff.fetchTiff() method

Rendering of the GeoTIFFs has to be done in a callback function (called “plotCallback()” in the illustration above). This is necessary because the tileLoadFunction is executed immediately when the tile is requested by the client. On a regular OpenLayers WMTS map this is not an issue because the SRC attribute of the image tile is set to the tile’s URL and the browser takes care of downloading and placing the image.

In case of client-side processing the data to draw is not yet available when the tileload-function executes, thus the callback function needs to be stored as a callback to the cached tile (urlToTiff[url]) and executes right when the AJAX request was successful.

4 Results

The KPIs that were defined in chapter 2 are benchmarked against traditional methods (WMS, WMTS) using a custom-built benchmark tool. All tests are done on a desktop computer using the chrome browser with WebGL support (stating that mobile usage is not a necessity for analysing and visualising complex scientific data). The plain JavaScript fallback for geotiff.js is not benchmarked.

4.1 Tool Explanation

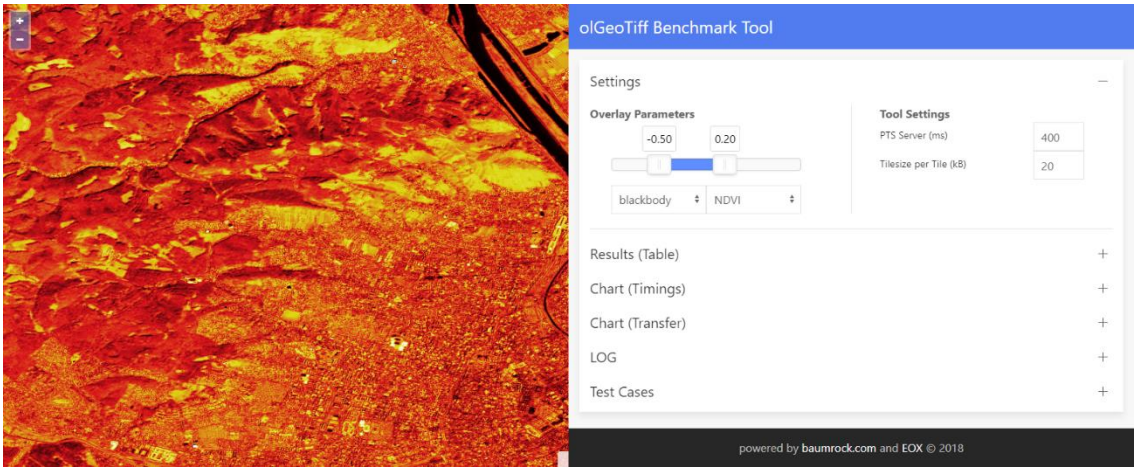

Figure 32: olGeoTiff benchmark tool

The tool was developed to compare server-side processing to client-side processing in an interactive way. The assumptions in the method chapter where that the results depend a lot on the user interactions, so we needed a way to benchmark the two situations while the user is interacting with the map.

Like shown in the figure above the tool consists of two main areas: The web-map on the left and the settings & results on the right. Some of the KPIs can be monitored through the browsers dev-tools, but not all of them. That's why we needed to modify the olGeoTiff plugin so that we can measure all our defined KPIs.

All the following screenshots will be taken from the same test-case:

```
-- 20 Interactions on Client --
1: Initial Map load
2: Change lower domain
3: Change lower domain
```

```
4: Change lower domain
5: Change lower domain
6: Change lower domain
7: Change lower domain
8: Change lower domain
9: Change lower domain
10: Change lower domain
11: Change lower domain
12: Change lower domain
13: Change lower domain
14: Change lower domain
15: Change lower domain
16: Change lower domain
17: Change lower domain
18: Change lower domain
19: Change lower domain
20: Change lower domain
```

This test-case makes no sense practically but is easy and makes the understanding of the tool a lot simpler. The tool settings are 400ms for PTS and 20kB tile-size if not declared differently.

4.1.1 Settings

In the settings section of the tool the user can define the parameters for the map rendering. This is the same user interface as shown in the prototype of chapter 3.1 only without the opacity setting that does not influence any of the variables since this setting is done by the browsers CSS engine in both server- and client-side processing scenarios.

On the right, there are two input fields for two variables: The time needed for processing on the server (PTS) in milliseconds and the tilesize per tile in kilobytes. These two variables are defined by the user. This is because the tool does only measure the KPIs and variables for the CLIENT side. The values “measured” for the server side are actually estimates based on the user input (like zooming, panning, parameter changes) and then calculated to estimates as realistically as possible.

This is a crucial point to know when discussing the results and there are several reasons for choosing this route. First, setting up a benchmark scenario that measures both the client-side and the server-side would be a whole more complex and would take a lot more time. This would go beyond the scope of this thesis.

Second, the effort taken for this addition would not gather a lot more benefit since the values measured always depend a lot on the used hardware. For example, you could have a slow server that takes a lot more time processing the WMTS requests than another. This would lead to totally different results and the desired findings of the tool – comparing client-side to server-side rendering – would take a wrong direction and develop more towards comparing different hardware setups.

Finally, having those variables definable through simple input fields makes it possible to change values with a single user input, making it possible to compare different (virtual) hardware setups and therefore gather more results and new findings with a lot less effort.

4.1.1.1 PTS Server (ms)

This setting shows the number of milliseconds that is needed for processing the request on the server. Using server-side processing the server gets a WMTS request and returns a rendered PNG or JPG tile to the client. This process takes some time and is for sure slower than just returning a regular, pre-rendered tile as it is the case at client-side rendering.

Caching is not possible in this case, because the client can request a virtually endless set of variation of those tiles. Every change in the user settings (domain, colour palette, …) leads to a different result, thus the server would have to return a different tile to the client.

To find a realistic setting for this value a WMS request was made to the sentinel-hub website that returns a 256x256 JPG tile: [1]

[1] https://services.sentinel-hub.com/ogc/wms/f3ab47b0-ac4c-4a67-a5db-24e2122a80d9?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&FORMAT=image%2Fjpg&TRANSPARENT=true&LAYERS=TRUE_COLOR&STYLES=&time=2018-01-20T00%3A00%3A00%2F2018-01-21T00%3A00%3A00&WIDTH=256&HEIGHT=256&SRS=EPSG%3A4326&BBOX=10.546875%2C55.546875%2C11.25%2C56.25

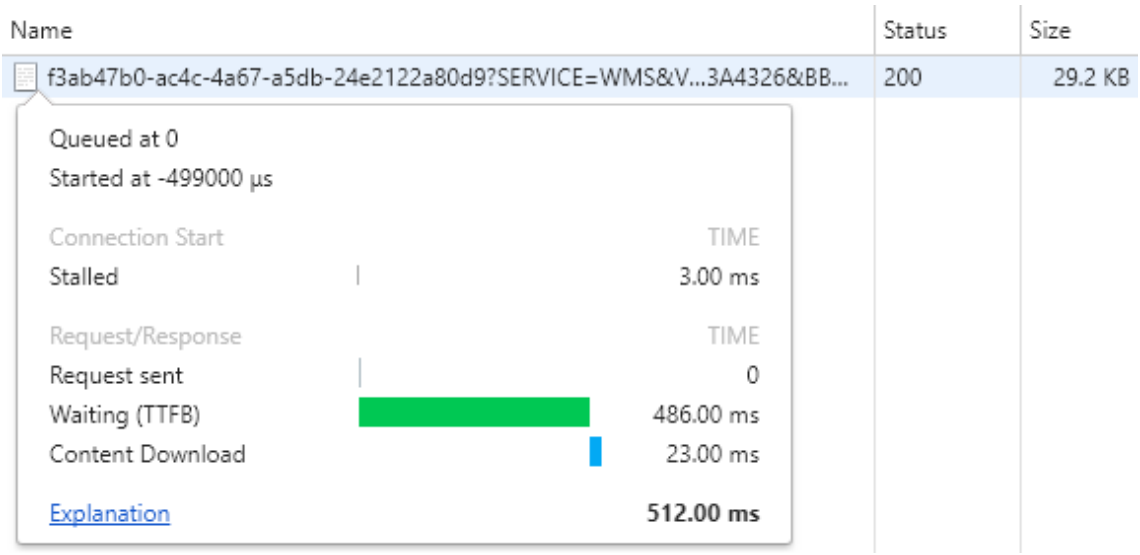

Figure 33: TTFB for a WMS request

The TTFB (Time To First Byte) describes the time from sending the request to returning the first byte. [27] In the figure above it shows that the request took 486ms. Another 29 requests were made to build average values and lower and upper limits to work with. From those 30 requests the lowest 5 and the highest 5 values were removed (one runaway took 2,6 seconds!) which resulted in a minimum time of 327ms, a maximum of 491ms and an average of 402ms. For the tests later on 300, 400 and 500ms will be used as PTS settings.

4.1.1.2 Tilesize per tile (kB)

This variable is a lot easier to estimate. Analysing other WMTS requests it can be seen that one 256x256 pixel tile has around 20kB as a JPG and around 100kB as PNG. 20kB is used as the default value since JPG should be fine for most cases and the value is a lot lower, meaning the KPIs would get even better using PNGs.

4.1.2 Results (Table)

Results (Table)

CLIENT-SIDE PROCESSING		
KPI/VARIABLE	LAST INTERACTION	TOTAL
NOI	1	20
NOT	0	16
RT	12.99ms 0ms/12.99ms	12.85s 11.56s/1.29s
DT	0kB	5.57MB
PTC	12.99ms 0ms/12.99ms	1.29s 1.06s/230.03ms
PTS		

SERVER-SIDE PROCESSING		
KPI/VARIABLE	LAST INTERACTION	TOTAL
NOI	1	20
NOT	16	320
RT	7.12s	142.29s
DT	320.00kB	6.25MB
PTC		
PTS	400.00ms	8.00s

Estimated Network Speed: 444.24kB/s

Figure 34: Results table

The results table shows all the KPIs for the last interaction and a total for all interactions both for client-side processing and server-side processing. It also shows an estimated network speed that is calculated by dividing the total amount of data transferred and the total amount of time consumed for that transactions.

```
574     // things to do when interaction was done
575     $(document).on('interactiondone', function() {
576       console.log('interactiondone');
577       olgt.tilesStarted = 0;
578       olgt.tilesDone = 0;
579
580       // update stats table
581       for(where in {server:null,client:null}) {…
616       }
617
618       // update charts
619       var lastclient = olgt.benchy.lastInteraction('client');
620       var lastserver = olgt.benchy.lastInteraction('server');
621       addChartData(timeChart, [olgt.benchy.total('client', 'rt'),olgt.benchy.total('server', 'rt'),lastclient.rt(),lastserver.rt()]);
622       addChartData(dataChart, [olgt.benchy.total('client', 'dt'),olgt.benchy.total('server', 'dt'),lastclient.dt,lastserver.dt]);
623
624       // calc estimate network speed
625       $('#ns').text(fileSize(getNS())+'/s');
626     });
```

Figure 35: Callback “interactiondone”

The table is updated on each interaction (lines 581ff) and also both charts get updated with the new data (lines 618 – 622). At the end the network speed is calculated, formatted and written to the DOM element with the id “ns”.

4.1.2.1 NOI

NOI stands for “Number Of Interactions”. Every user input is counted as one interaction (like panning, zooming, changing parameters).

4.1.2.2 NOT

NOT stands for “Number Of Tiles requested”. Figure 34 shows an example of a client-side interaction: 16 tiles where requested from the server on the right table while 0 tiles where requested on the left.

4.1.2.3 RT

RT stands for “Request Time”. For client-side processing the PTS (TTFB) is not measured, because it was stated that the tiles are already pre-rendered and cached on the server, therefore the server just has to return the requested files without any further processing. Also, the RT measures the overall time needed for sending the ajax request until finishing the download of the time, which already includes any processing on the server.

The RT consists of the time needed for downloading the data from the server to the client and – in case of client-side processing – the time needed for parsing this data and rendering it to the map.

4.1.2.4 DT

DT stands for “Data Transferred”. This is the overall data transferred for each interaction. For client-side processing this is a measured value, for server-side processing this is simply the tile-size setting multiplied by the number of tiles requested.

4.1.2.5 PTC

PTC stands for “Processing Time on Client”. This value is only available for client-side processing. It consists of the time needed for parsing the data (only when it was requested from the server and not requested from the cache) and rendering it to the map.

4.1.2.6 PTS

PTS stands for “Processing Time on Server”. This is only available for the server-side processing and as explained above, this is a static setting that can be input by the user. Details are discussed in chapter 4.1.1.1.

4.1.3 Chart (Timings)

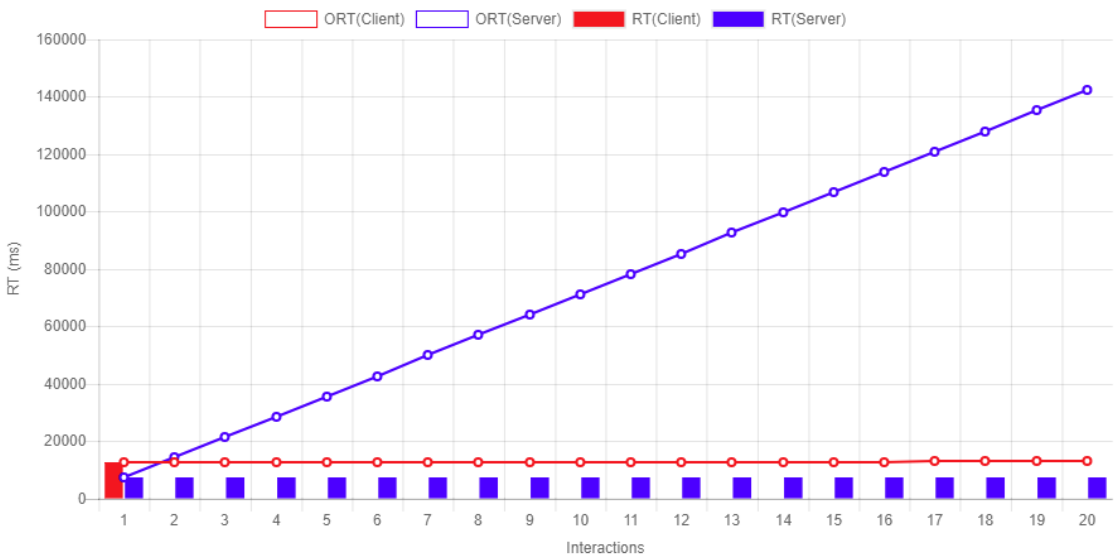

Figure 36: Timings chart

The visual representation gives an instant idea of what is going on regarding the RT for each interaction (red and blue bars) and the accumulated values (red and blue lines), showing the ORT. The x-axis shows the interactions that where triggered by user input, the y-axis shows the RT in milliseconds.

4.1.4 Chart (Transfer)

The transfer chart has the same concept but is showing a different KPI. The x-axis stays the same, the y-axis shows the DT in kilobytes. Both charts are interactive and instantly updated when the user does some input. This makes it very intuitive and creates an instant understanding of the differences between both techniques (server-side and client-side processing).

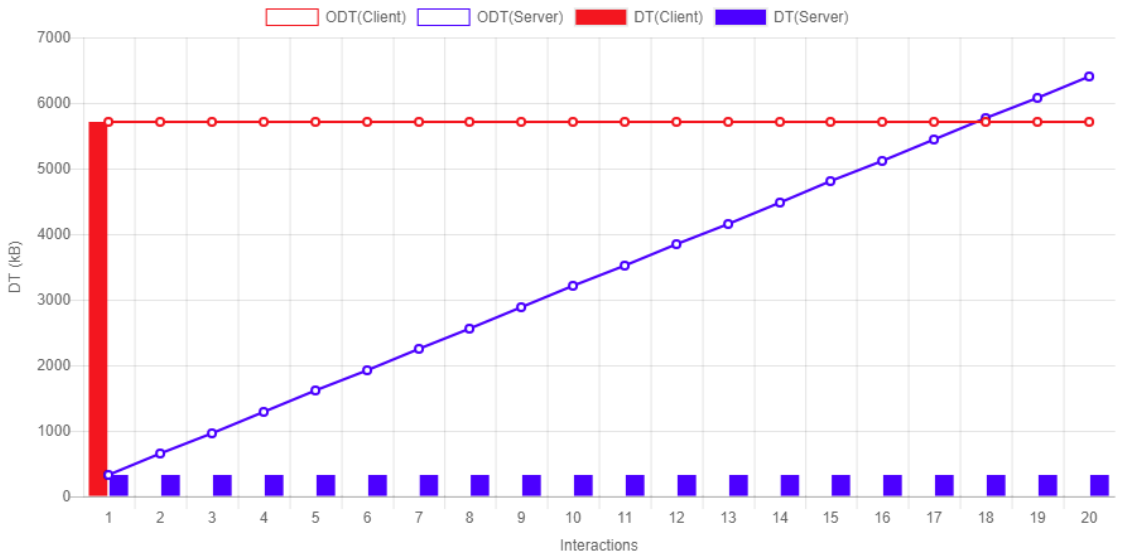

Figure 37: Transfer Chart

4.1.5 Log

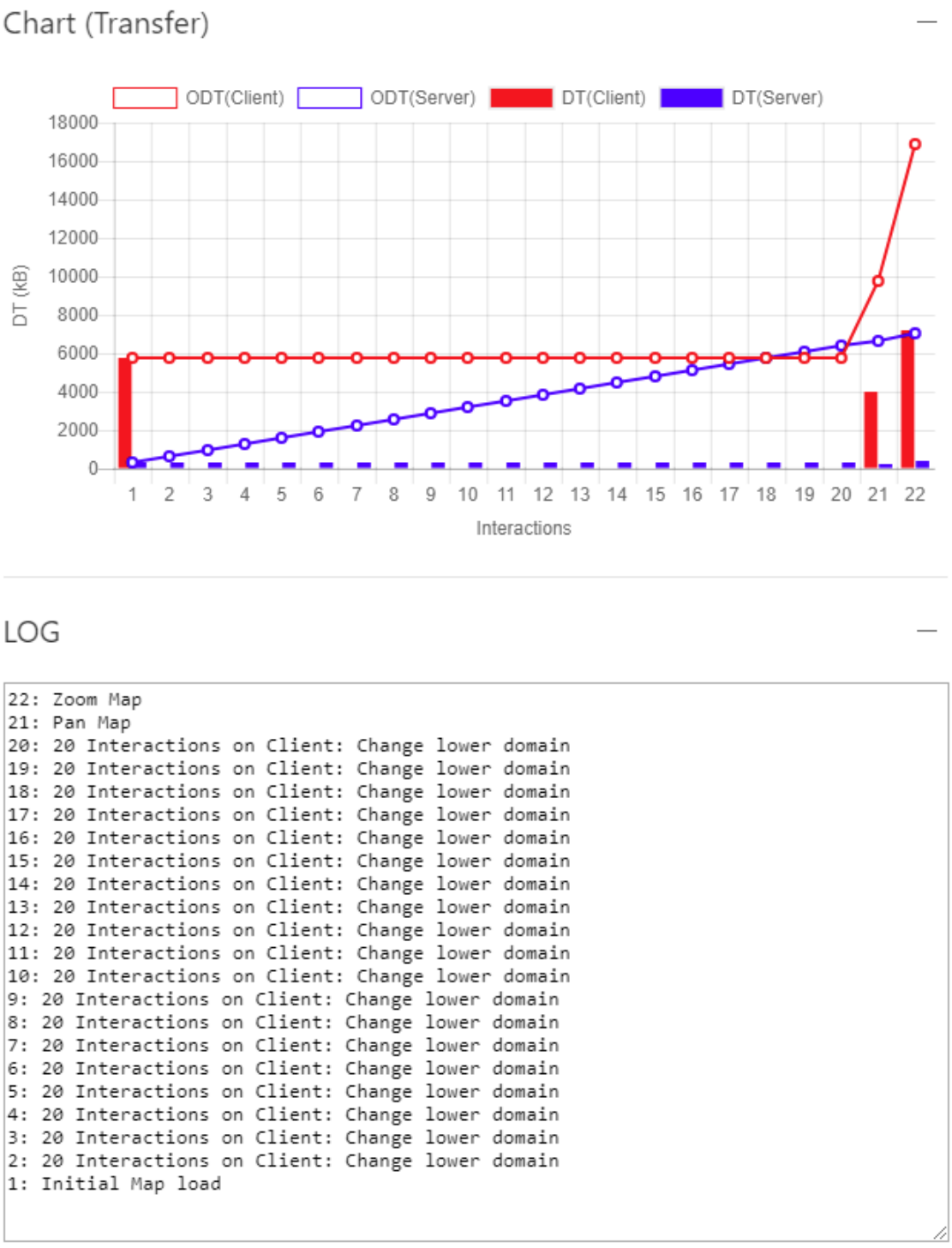

Figure 38: olGeoTiff benchmark log

The benchmark log helps to interpret the chart. It shows all interactions that have taken place, prefixed with the interaction number and in case of an automated test-case the name of the test-case (here “20 Interactions on Client” from log-item 2 to 20).

4.1.6 Test Cases

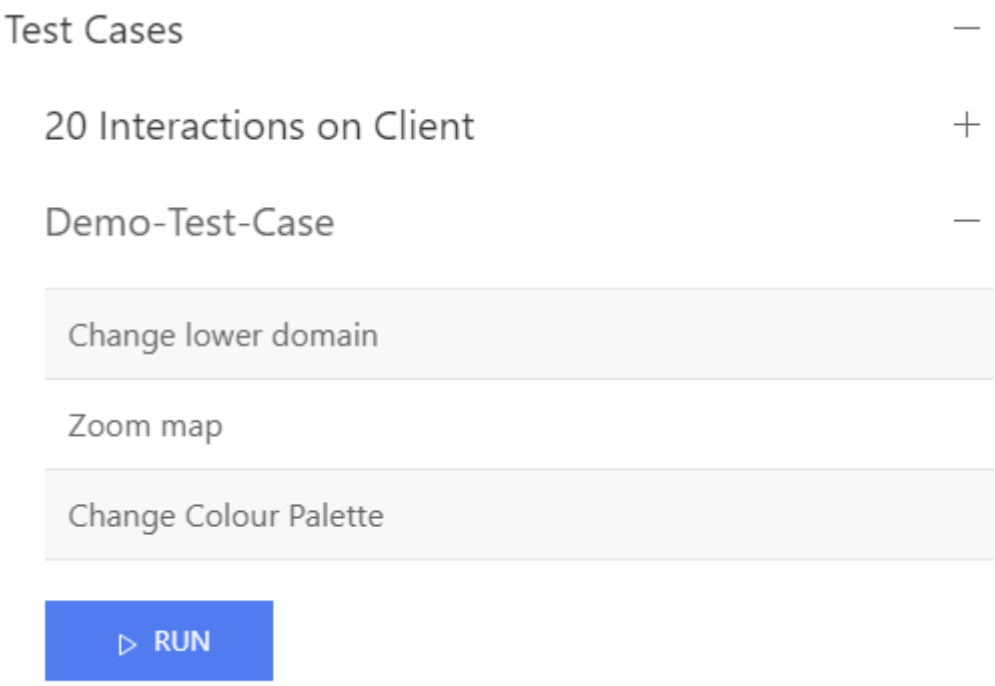

Figure 39: Test cases

Test cases can easily be defined in the code of the benchmark tool. At the user interface they get listed with all their actions followed by a “run” button that starts the execution of this case. Those test-cases make it possible to compare different standardised situations at different circumstances (like different network speeds for example).

```
// test cases
var domainslider = $(document).find('.domainslider')[0];
var tests = [
  {
    title: '20 Interactions on Client',
    actions: [···
    ]
  },{
    title: 'Demo-Test-Case',
    actions: [
      {
        title: 'Change lower domain',
        action: function() {
          domainslider.noUiSlider.set([-0.5,null]);
        }
      },{
        title: 'Zoom map',
        action: function() {
          map.getView().setZoom(12);
        }
      },{
        title: 'Change Colour Palette',
        action: function() {
          $('#palette').val('bone').change();
        }
      },
    ]
  },
];
```

Figure 40: Definition of a sample test-case

This simple test-case sets the lower domain to -0.5, sets the zoom to 12 and then changes the colour palette to “bone”.

4.2 Results

In this chapter the results of three test cases will be shown. It was already mentioned that the results heavily depend on the user input – especially the type of user input (namely user input that leads to communication with the server or user input that happens completely on the client). Test case 1 has interactions of both types, test case 2 has 20 interactions happening completely on the client and finally test case three shows an example of interactions that involve communication to the server.

4.2.1 Test Case 1: 7 Interactions (Client and Server)

This test case is a mix of interactions that happen only on the client (when data was already fetched from the server and only has to be processed by the client due to changes of rendering parameters, for example) and involve some data transfer from the server to the client (for example panning and zooming). The log of the test case is as follows (the benchmark log shows action in reversed order so that the last interaction is always on top):

```
7: 7 Interactions (Client and Server): Change lower domain
6: 7 Interactions (Client and Server): Change upper domain
5: 7 Interactions (Client and Server): Pan map
4: 7 Interactions (Client and Server): Change Colour Palette
3: 7 Interactions (Client and Server): Zoom map
2: 7 Interactions (Client and Server): Change lower domain
1: Initial Map load
```

This test case shows the two main KPIs: The request time for each interaction (RT) and overall (ORT) and the data transferred – also for each interaction (DT) and overall (ODT). All setups are tested with different settings for the server-side (PTS 300, 400 and 500ms) and at different network speeds:

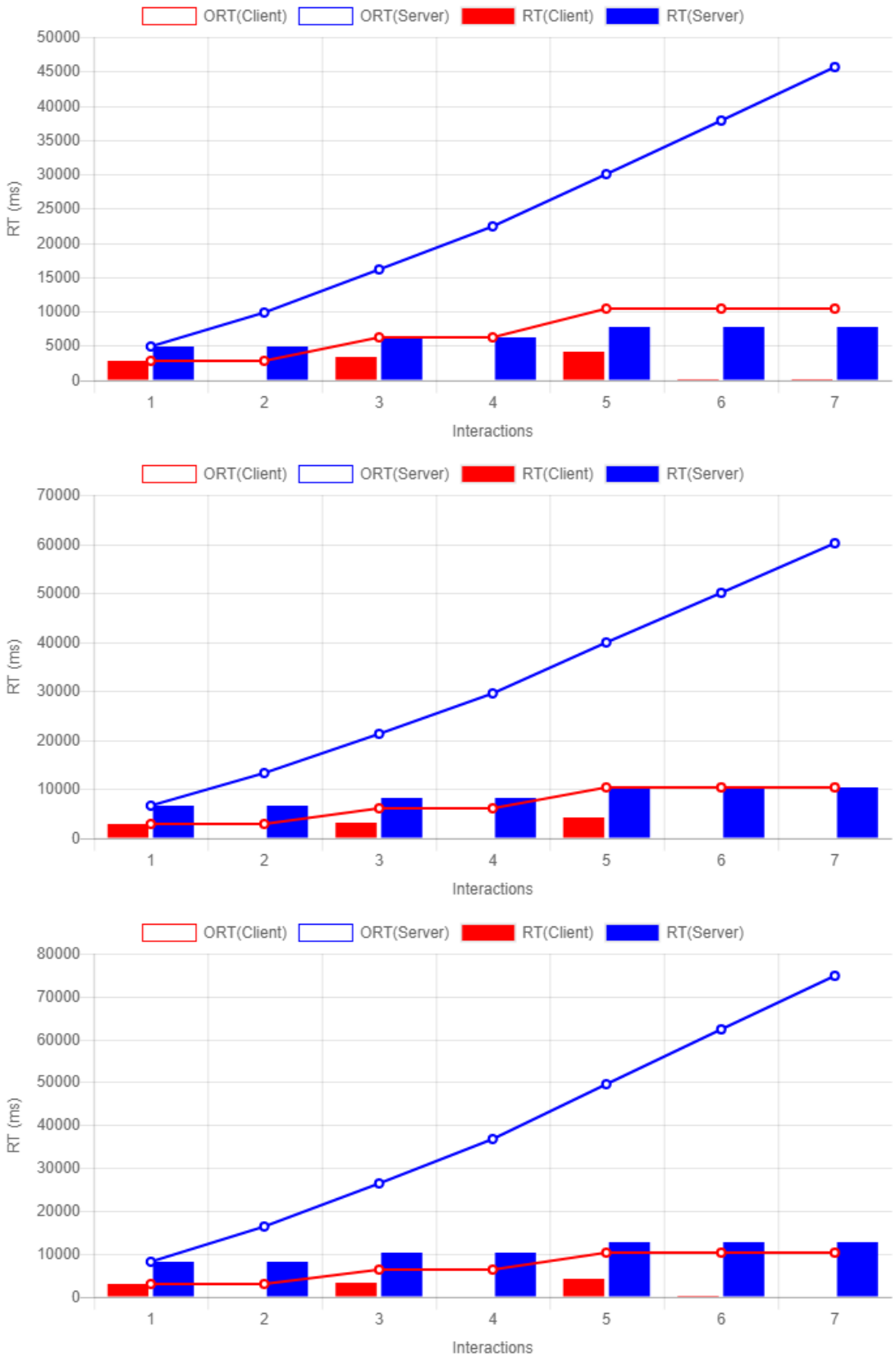

Figure 41: Results for network speed 2,00MB/s (PTS 300/400/500)

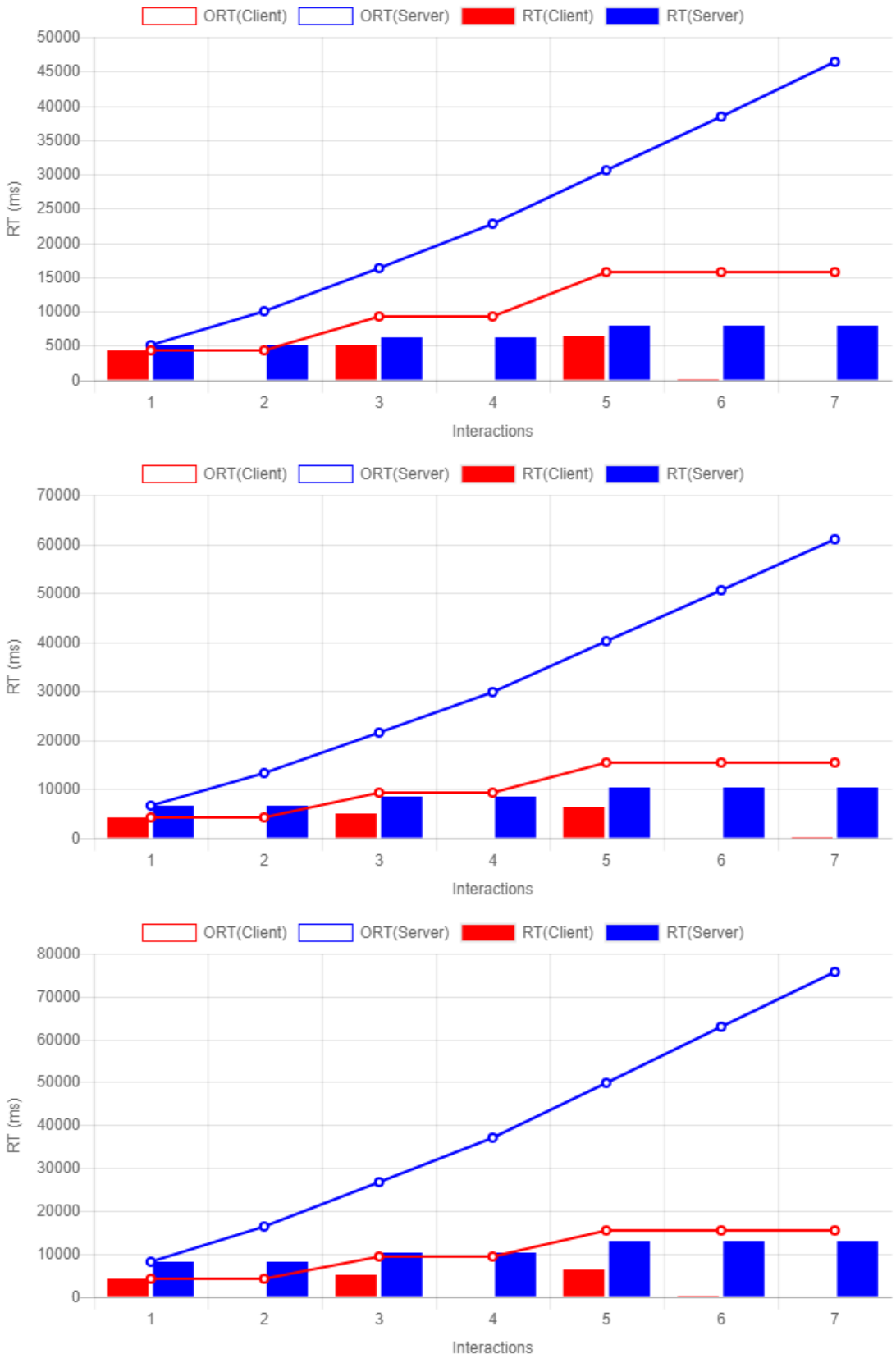

Figure 42: Results for network speed 1,31MB/s (PTS 300/400/500)

Figure 43: Results for network speed 0,5MB/s (PTS 300/400/500)

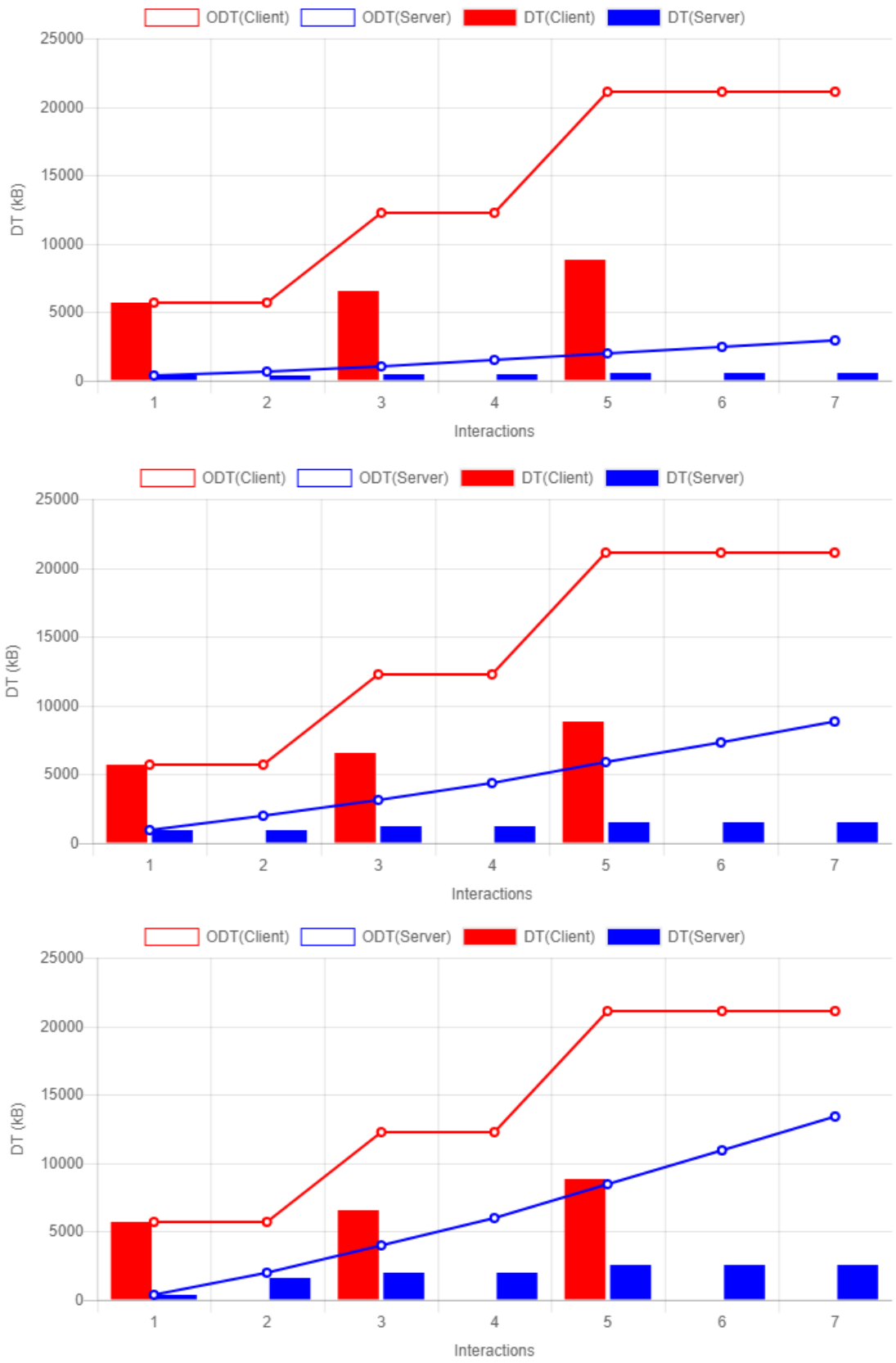

Figure 44: Results for ODT (Tilesize 20/60/100kB)

4.2.2 Test Case 2: 20 Interactions on Client

Testcase two is a very monotonous test. Again, the timings and the data transferred were tested. For all tests regarding the timings the tilesize was 20kB. The log of the test is as follows:

```
20: 20 Interactions on Client: Change lower domain
19: 20 Interactions on Client: Change lower domain
18: 20 Interactions on Client: Change lower domain
17: 20 Interactions on Client: Change lower domain
16: 20 Interactions on Client: Change lower domain
15: 20 Interactions on Client: Change lower domain
14: 20 Interactions on Client: Change lower domain
13: 20 Interactions on Client: Change lower domain
12: 20 Interactions on Client: Change lower domain
11: 20 Interactions on Client: Change lower domain
10: 20 Interactions on Client: Change lower domain
9: 20 Interactions on Client: Change lower domain
8: 20 Interactions on Client: Change lower domain
7: 20 Interactions on Client: Change lower domain
6: 20 Interactions on Client: Change lower domain
5: 20 Interactions on Client: Change lower domain
4: 20 Interactions on Client: Change lower domain
3: 20 Interactions on Client: Change lower domain
2: 20 Interactions on Client: Change lower domain
1: Initial Map load
```

The reason for this test is to show both extreme situations: One having no data transfer from the server to the client at all after the initial map load and one having transfer at each user interaction (see test number 3 in the next chapter).

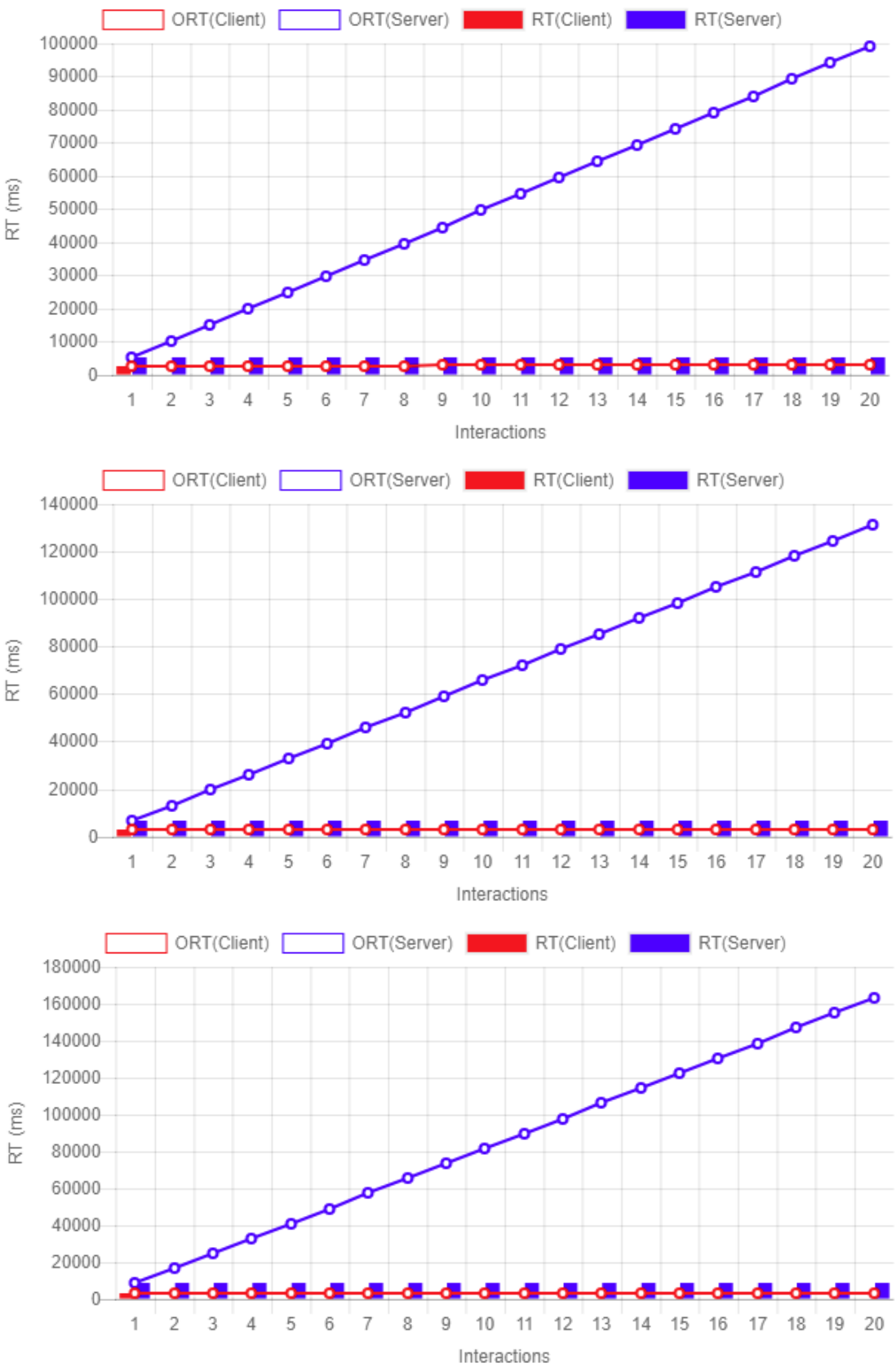

Figure 45: Results for network speed 2,6MB/s (PTS 300/400/500)

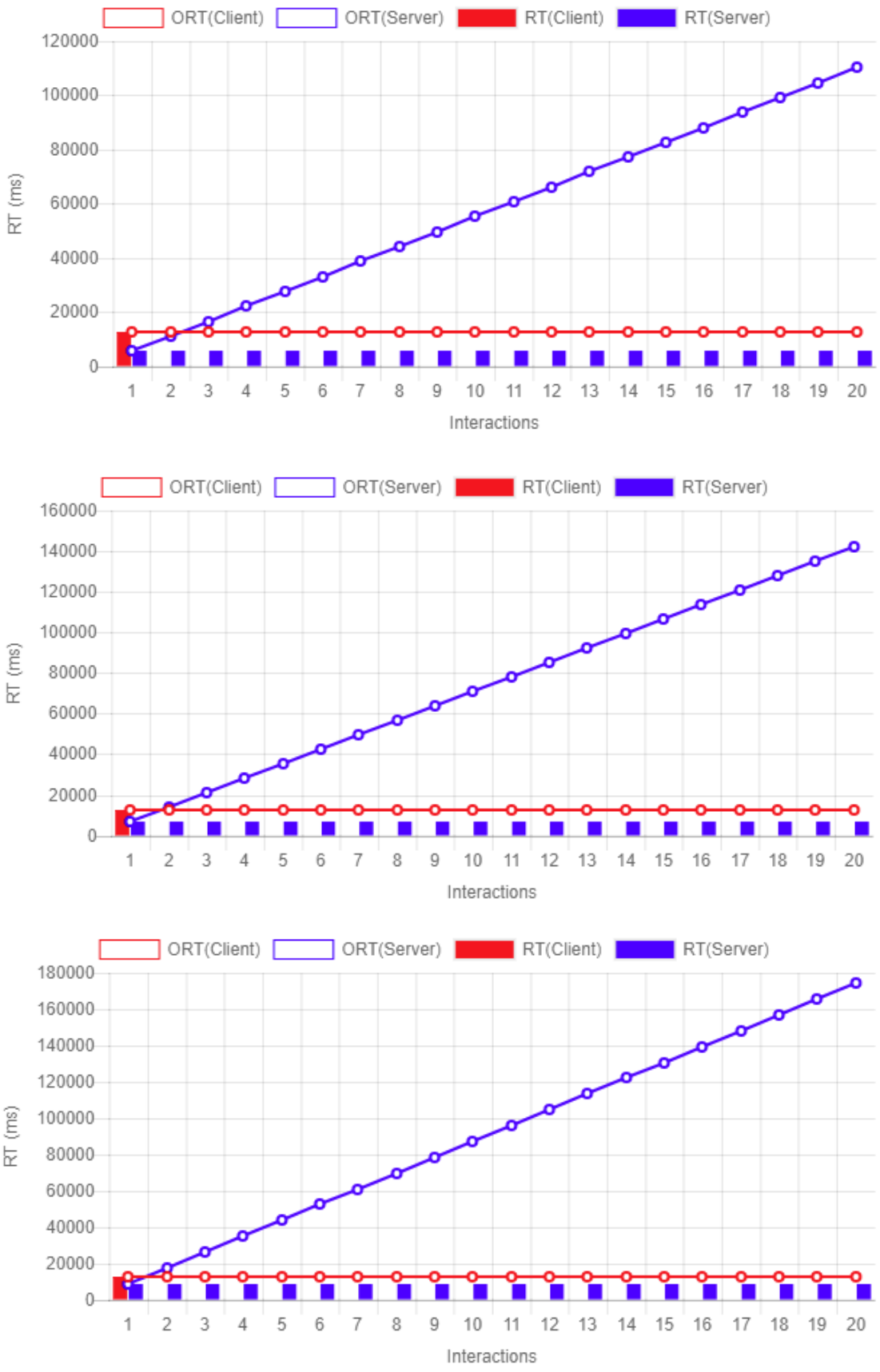

Figure 46: Results for network speed 0,5MB/s (PTS 300/400/500)

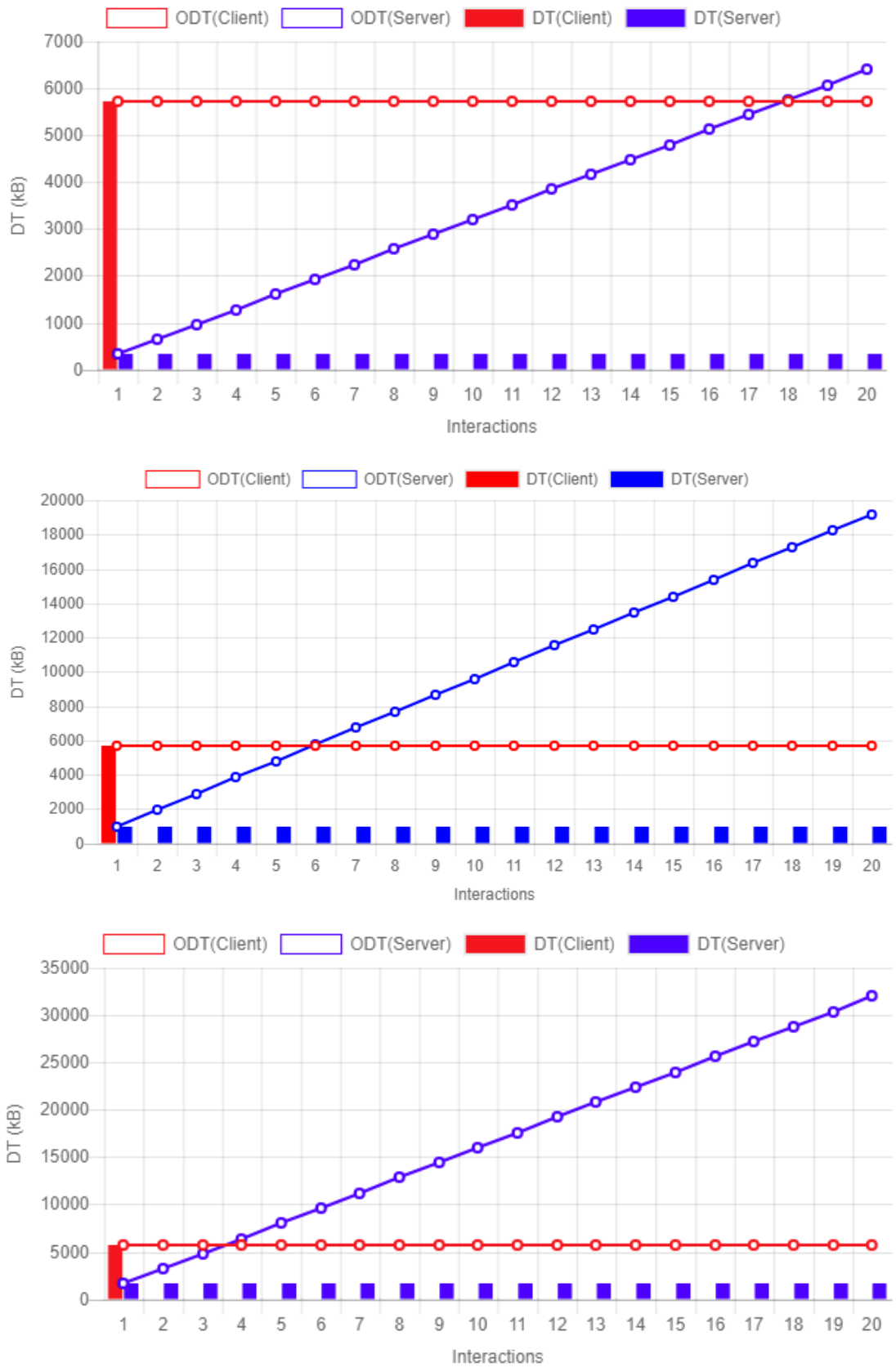

Figure 47: Results for different tile-sizes (20/60/100kB)

4.2.3 Test Case 3: 5 Interactions on Server

This test case shows the other extreme situation: User-input where every operation involves communication to the server both in client-side as well as server-side processing. Again, the tilesize for the timing-tests was always 20kB. The log is as follows:

```
5: 5 Interactions on Server: Pan map
4: 5 Interactions on Server: Pan map
3: 5 Interactions on Server: Pan map
2: 5 Interactions on Server: Zoom map
1: Initial Map load
```

Every interaction leads to a complete reload of the map, meaning all tiles are reloaded. This could be different when the map was panned only slightly to one direction and therefore would only reload a subset of all displayed tiles. The effects would still be the same, but the resulting lines would be less steep.

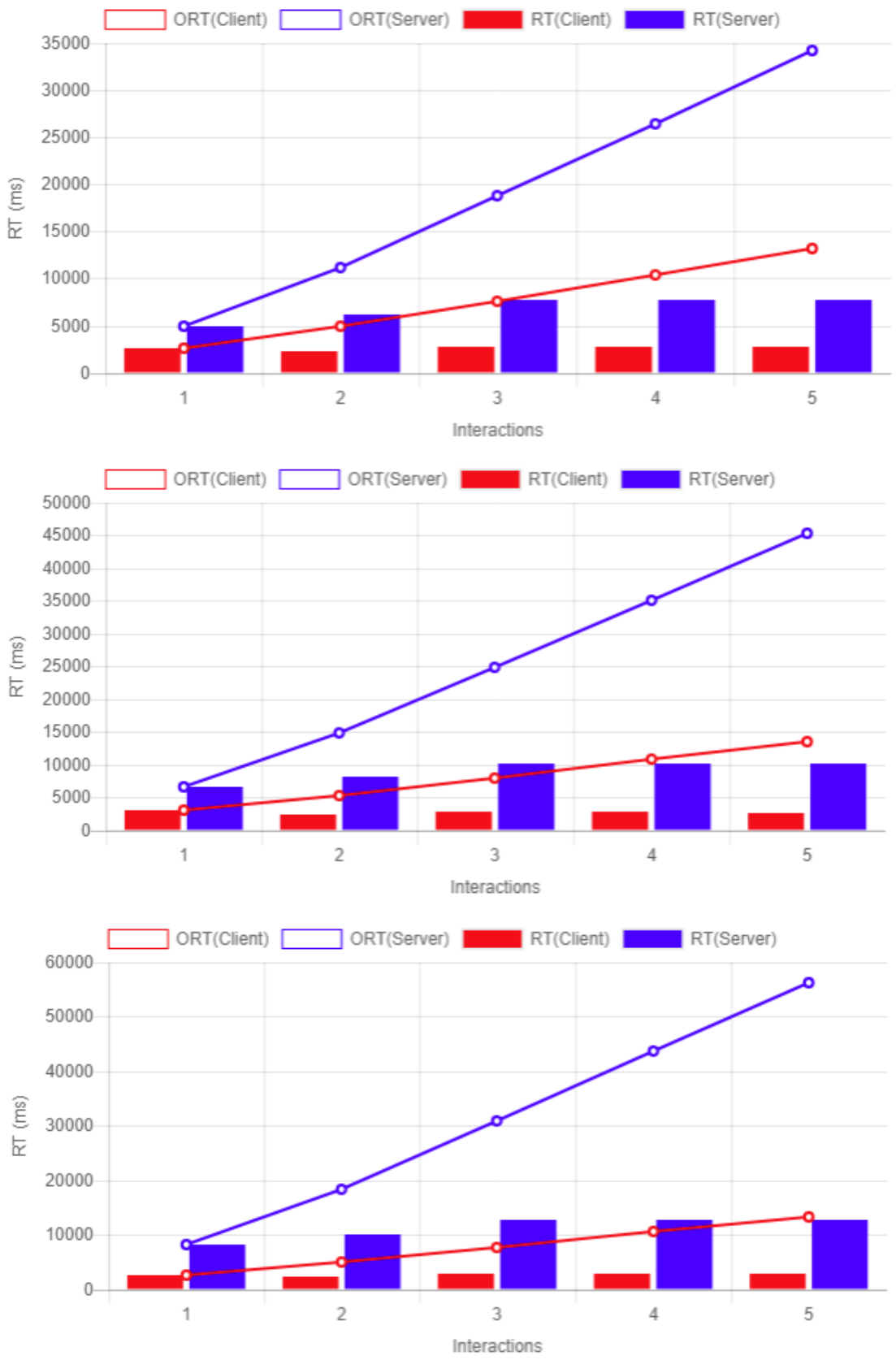

Figure 48: Results for network speed 2,7MB/s (PTS 300/400/500)

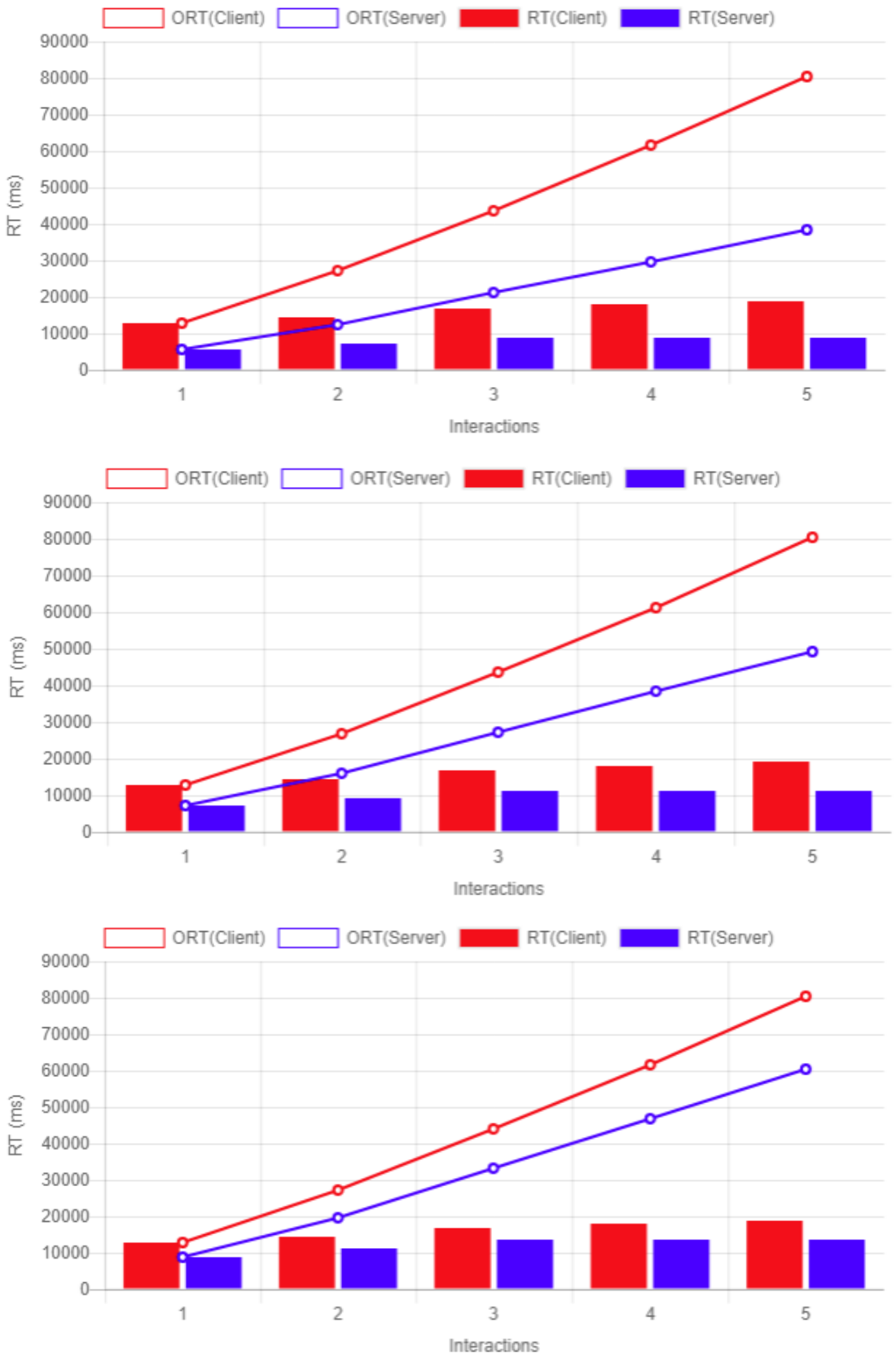

Figure 49: Results for network speed 0,45MB/s (PTS 300/400/500)

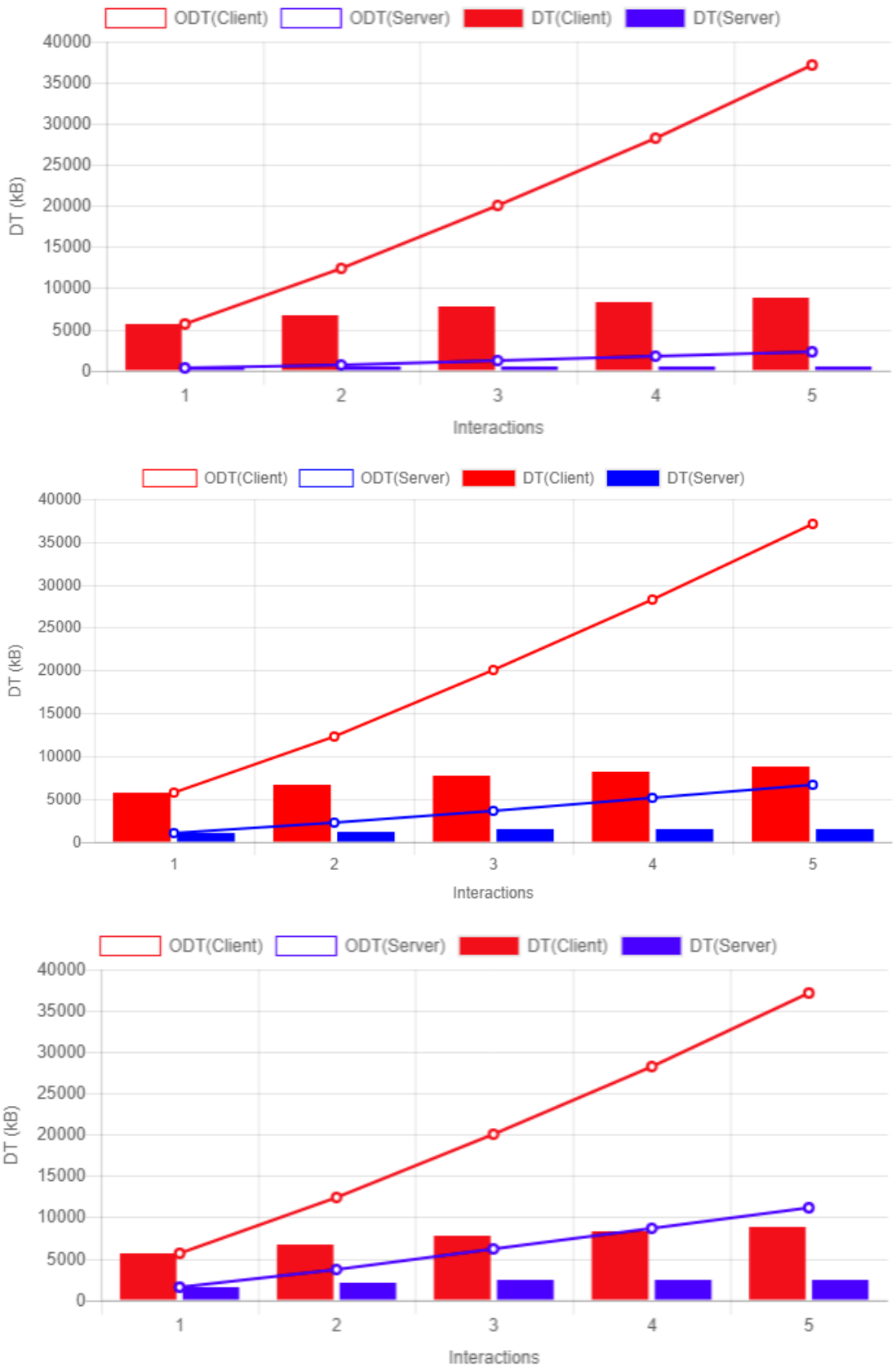

Figure 50: Results for different tile-sizes (20/60/100kB)

4.3 Discussion

4.3.1 Test Case 1: 7 Interactions (Client and Server)

The analysation of the results of test case 1 showed several interesting insights. First of all, the client-side request time is lower than the request time of server-side processing in all three cases (after the seventh interaction). We see, that the client-side timings for interactions 2, 4, 6 and 7 are very low compared to the both the other client-side interactions and all server-side interactions. Timings for those interactions are around 10ms, consisting of 0ms for parsing (because parsing is only done once when the GeoTIFF gets downloaded) and 10ms for rendering/plotting.

Obviously, the client timings at different PTS didn't change at all. Only the values for server-side processing changed. To see those changes better, a chart was created that shows the result of 3 individual tests in one chart. Again, client-side processing is in red colour, server-side processing is blue. PTS 300 has 80% opacity, PTS 400 has 40% and PTS 500 has 0% opacity. All three client lines have the same position whereas the server-side lines get steeper by increasing PTS.

This same principle was used for two more charts comparing the same relation but at a different network speed. The first chart shows the results for 2MB/s network speed, the second for 1,31MB/s and the last for 0,5MB/s.

Interestingly the blue lines do not change as much as the red lines change from one chart to another. To make this easier to see another chart was created, showing the same relation but this time it compares the results of different network speeds at the same PTS (Figure 52).

The last chart of the next page shows the most interesting case: Client- and server-side processing are almost heads-up. Every server-side interaction makes the client-side method slower, every interaction that is only rendered on the client makes the server-side method slower. This shows powerfully the impact of the user and the kind of interactions he makes. Though – in both other cases (faster network) this effect would be a lot less distinct or could even be no matter.

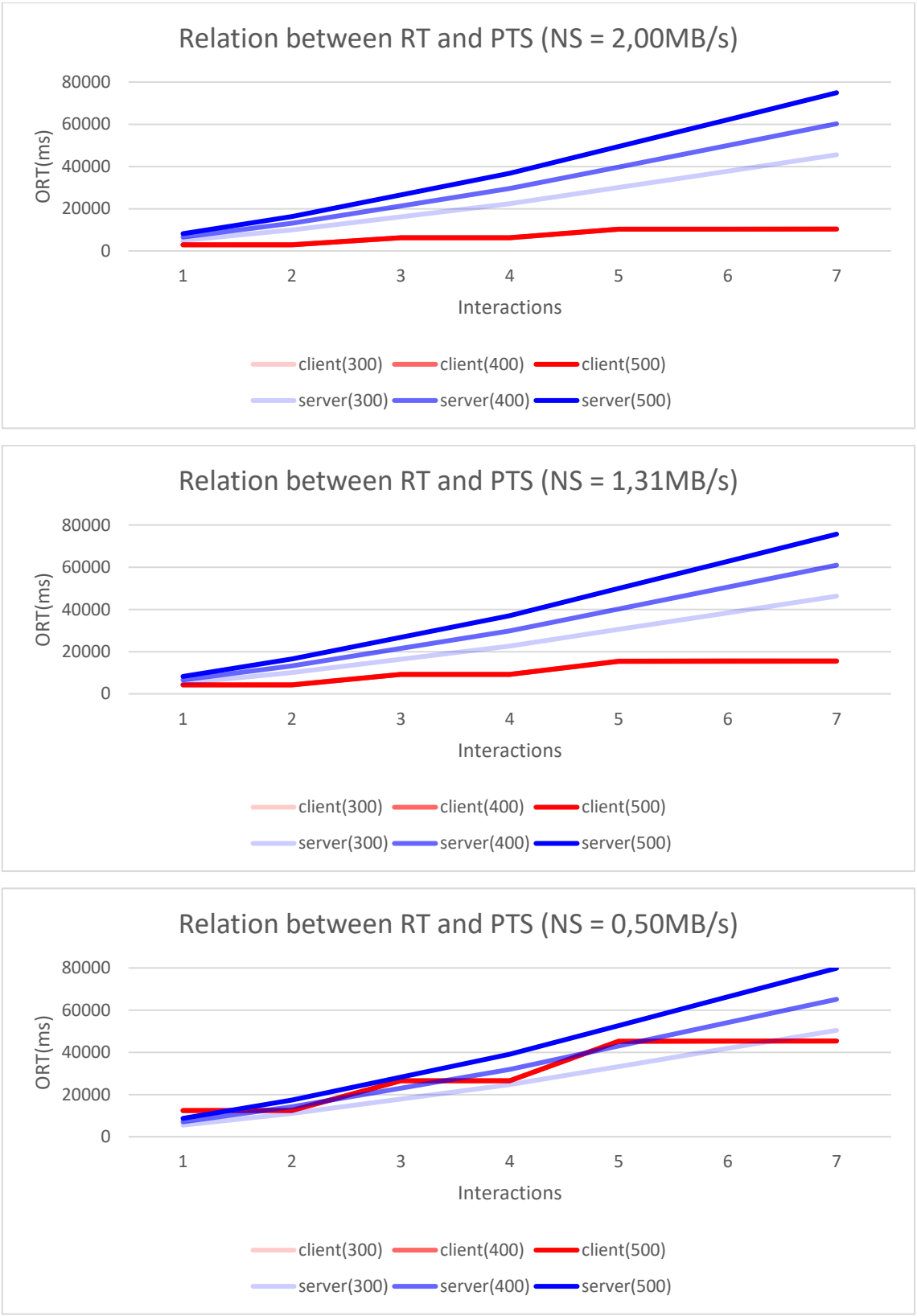

Figure 51: Comparing PTS at different network speeds

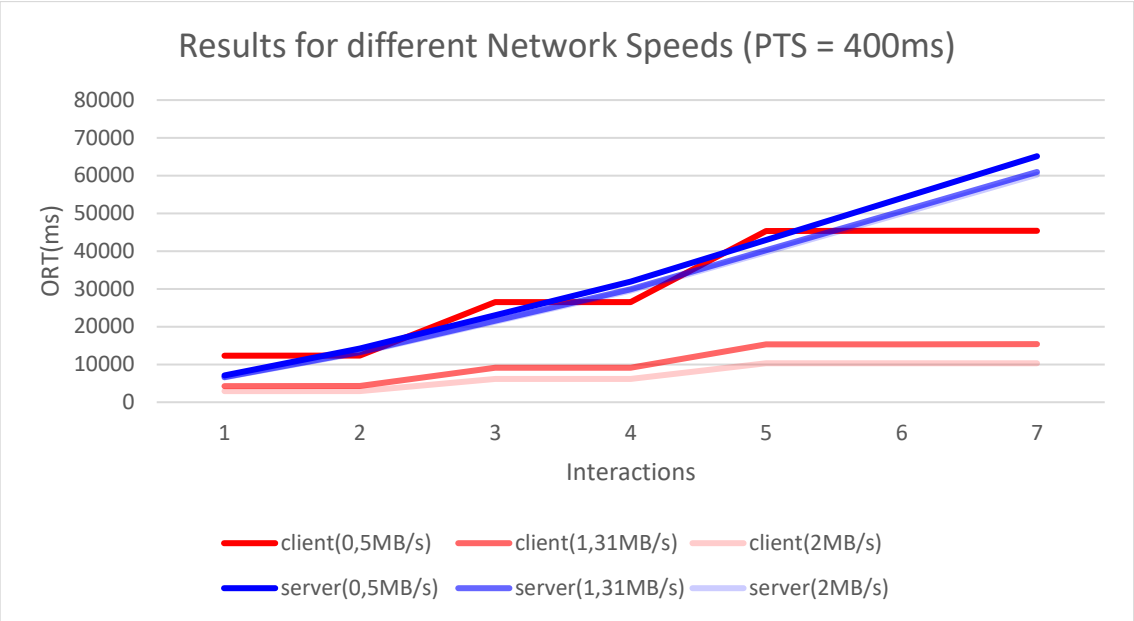

Figure 52: Different network speeds at PTS 400

The first insight is that the spread of the blue lines (server-side processing results) is a lot less than the spread of the red lines (client-side processing results). Both show an increase of ORT with lower network speeds, but the increase at client-side processing is a lot higher than at server-side processing. The network speed, on the other hand, changes almost linearly from 0,5 to 1,31 (+0,81) and from 1,31 to 2,0 (+0,69).

This is a very interesting behaviour, so another chart was created to show this effect:

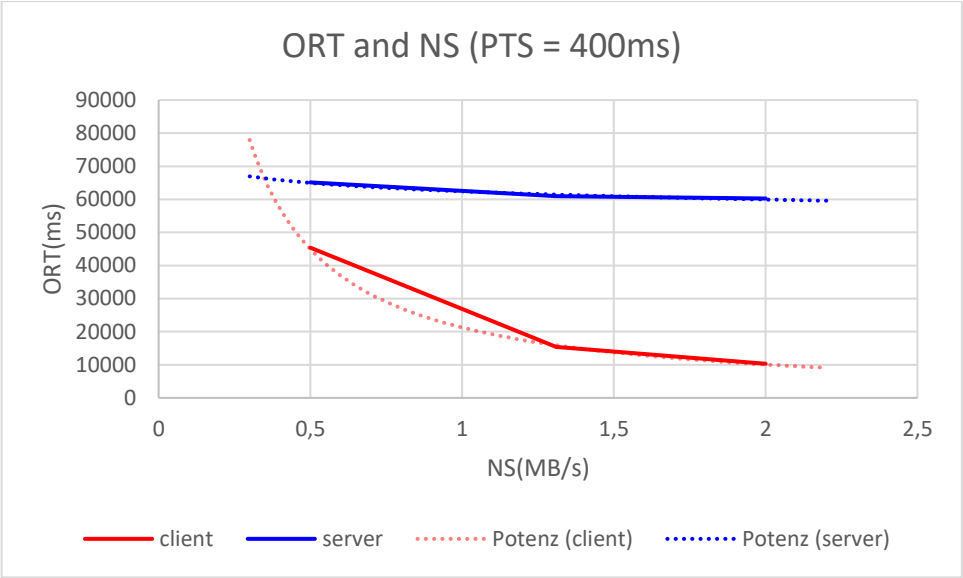

Figure 53: ORT and NS at PTS 400

This chart shows the ORT at different network speeds after the final interaction. The bespoken effect is clearly visible: The pitch of the red line clearly increases from 1,3 to 0,5. Trendlines show an estimate trend of both results. To check this estimation a fourth benchmark was taken:

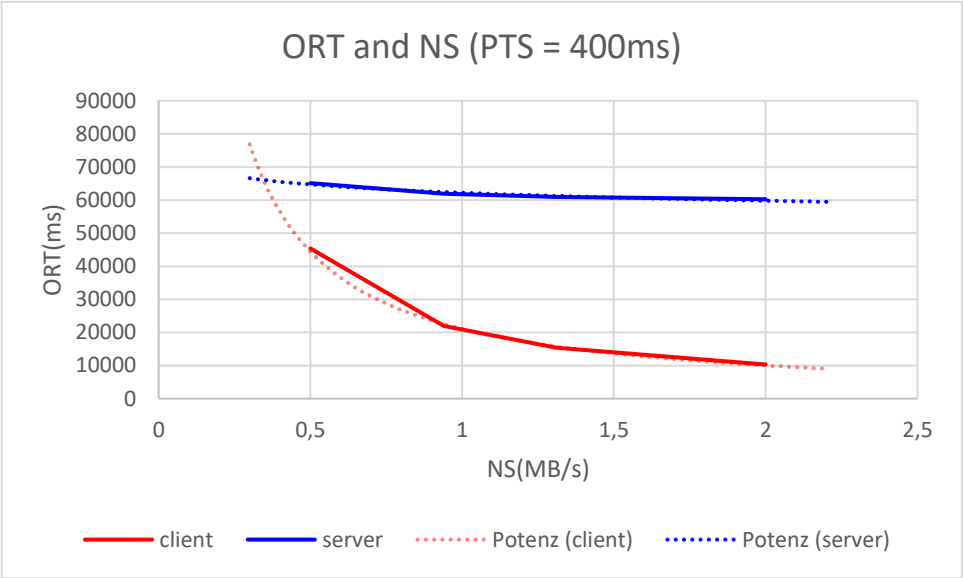

Figure 54: Estimate effect of network speed on ORT

The fourth value (taken at a NS of 0,94MB/s) shows a very good estimation, at least in the investigated range of network speeds. The reason for this effect is also discussed in the chapter to test-case 3.

The discussion of the results regarding the transferred date (ODT) is a lot easier. This KPI is constant across all tests on the client-side and it also does not vary at different network speeds. Therefore we need only one line for the client-side and three lines for the server-side: One at 20kB tilesize, one at 60kB and one at 100kB:

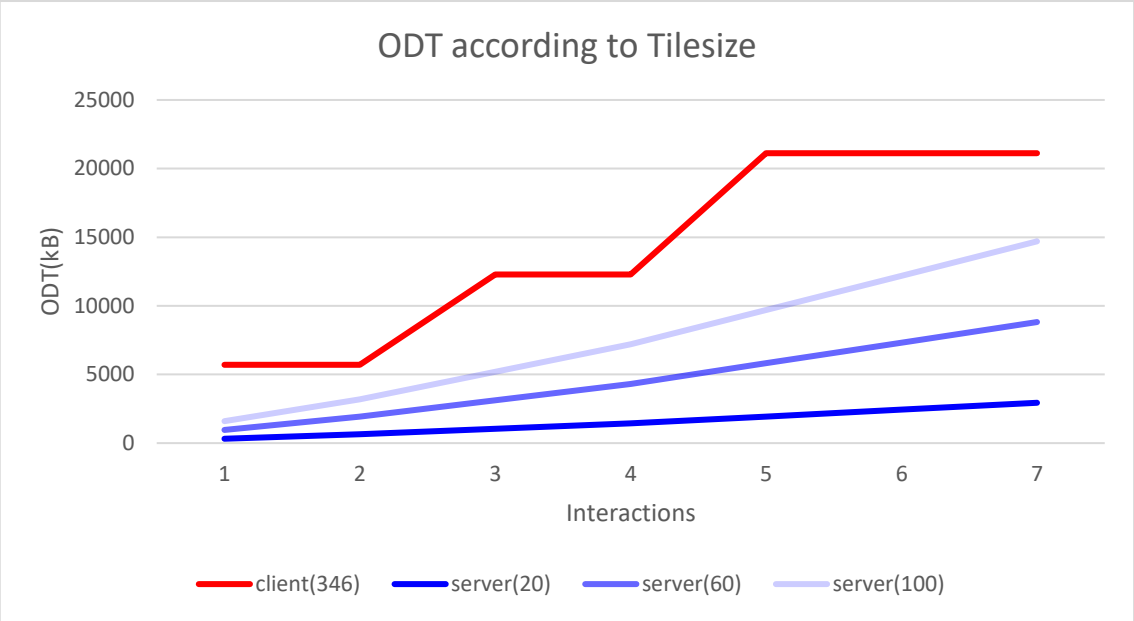

Figure 55: ODT according to tile-size

The average tile-size at client-side processing (loading GeoTIFF tiles instead of JPGs or PNGs) is 346kB per tile (total of 20,63MB over 61 tiles). Whereas the traffic increases only at interactions 1, 3 and 5 on the client-line, it increases linearly on the server-lines. This does not have to be like that, though – this is only the case when every interaction effects the same number of tiles (like zooming). Panning the map only slightly to one direction could lead to a lower number of tiles affected and therefore less traffic consumed (both on the client- and server-line).

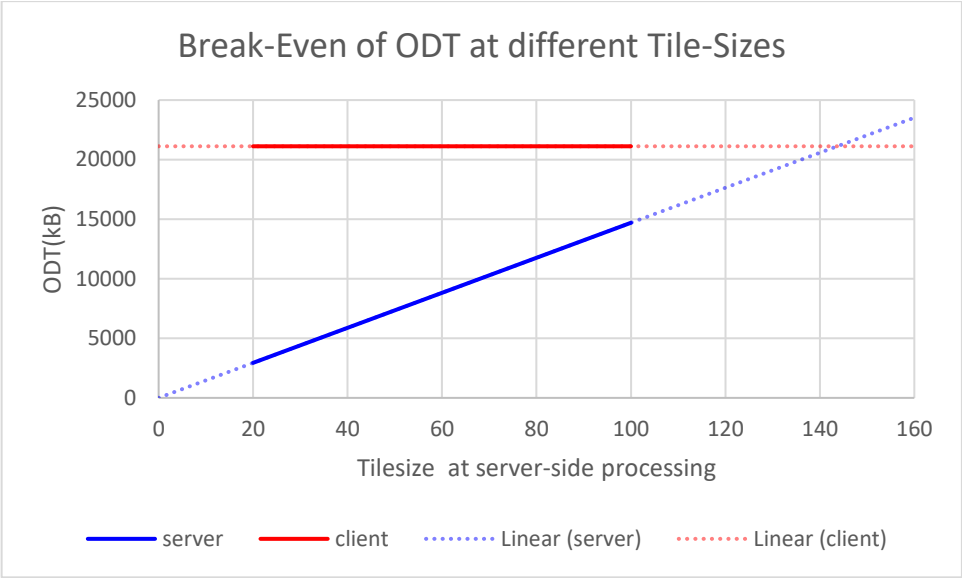

Figure 56: Break-even of ODT at different tile-sizes

Figure 56 shows that both the ODT for client-side processing and the ODT for server-side processing are linear lines. Whereas the ODT for the server-line is directly dependent from the tile-size the ODT of the client-line stays constant for all tile-sizes at the server-side. The break-even where the overall data transferred is the same for client- and server-side processing is (for this special test-case) at around 140kB, but again this is heavily dependent on the kind of interactions the user takes.

4.3.2 Test Case 2: 20 Interactions on Client

Looking at the timings this test-case does not show any new findings other than those shown in test-case 1. Client-side processing is – as expected – a lot faster than server-side processing, because the server is only involved in the initial map load. At fast network speeds it is even faster from the beginning! At slower network speeds it will be slightly slower for the initial load and rapidly get faster with each interaction (more on that in the details for test case 3).

Combining the charts with the data transferred it can be shown that the break-even for this test-case is at different positions depending on the tilesize of the tiles that are transferred from the server to the client. The tilesize for client-side processing is always the same, because each tile already includes all the necessary data and is only transferred on the initial map load.

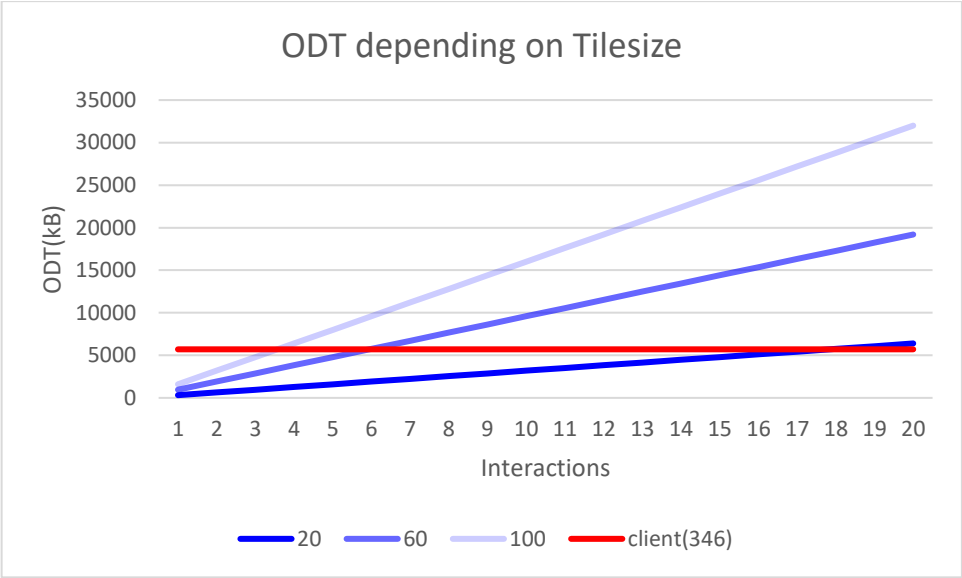

Figure 57: ODT depending on tile-size

The higher the tile-size, the earlier the break-even occurs. In case of 100kB per tile the break-even is reached at the fourth interaction, in case of 60kB it is reached at the sixth and in case of 20kB per tile it is reached at the eighteenth interaction.

4.3.3 Test Case 3: 5 Interactions on Server

Figure 48 shows that client-side processing is faster than server-side processing in all three tested scenarios (different PTS). Figure 49, on the other hand, shows the opposite: Here the server-side processing is faster – both over all interaction and also on each interaction separately. The difference is only the network speed and its effect was already discussed in the related chapter.

In this test-case, though, it gets obvious that every single client-interaction is slower or faster than the server-one depending on the network speed, whereas in test-case 1 only the final timings were analysed.

Figure 58: Time consuming components of one interaction

Figure 58 shows the details of one single interaction and analyzes which component of the interaction consumes which amount of time. Every interaction can consist of parsing and rendering the data (which only happens at client-side processing), processing the request (PTS – only at server-side processing) and the time needed for the download. The diagram shows powerfully the effect of a fast network compared to a slow one. For server-side processing the PTS is constant, because it only depends on the server setup. The lower bars (showing client-side processing) on the other hand show that parsing and rendering (so low that it is not visible in the chart) are almost constant and a lot lower than PTS on the server. So the component that has the greatest impact on the overall time is "download" and this part is still (and will always be) higher at client-side processing but since parsing and rendering are so fast on the client overall the client-side gets faster.

Looking at the data transferred (ODT) the server-side processing is more efficient. In this test-case the server-side processing is compared to client-side processing with GeoTIFF tiles at an average of 346kB each. The more bands are stored in those files the bigger the filesize would get and the bigger the difference to server-side processing would get.

5 Conclusion

By building a prototype (chapter 3) it was shown that it is possible to combine the principles of WCS (providing subsets of the original data with original semantics) and WMTS (providing the data as tile pyramids for fast and efficient transmission to the client) using a handful of open source tools.

Benchmarking this working prototype (chapter 4) showed that the network speed is the factor with the biggest impact on the overall request timings. This can mean that a client requesting a web-map and doing some interactions (like zooming, panning, changing rendering parameters) can get faster results using client-side processing than using server-side processing and waiting for the server to process all his requests. The kind of interactions that the user takes (an interaction can involve the server for downloading new data or can happen exclusively on the client when the data was already requested earlier) can have an impact on the result but it does not have to have an impact. If the network is fast enough and the server is slow, the client-side processing method can even be faster on each single request, even if the request involves downloading data from the server in both the server- and the client-side case (chapter 4.3.3).

Where the user really benefits from this new approach is when he needs to do lots of interactions with the same data (like modifying the colour palette or upper and lower domain limits for the rendering). In those cases, client-side processing can be a lot faster and return instant results whereas sending requests to the server and waiting for a response always implies a delay.

Unfortunately, the way the developed OpenLayers plugin works right now, it is not possible to update the map seamlessly with every user input. There is a short delay that is needed for resetting the map's internal cache to instruct it to redraw all the tiles. This leads to a short flicker on each redraw making it impossible to redraw the map while the user is moving around a slider, for example. This aspect could be topic of further work and would open up a lot of possibilities that would not be possible using server-side processing. An example could be showing animation-like time series of data directly reacting to the users input (sliders, settings, colour palettes, ...).

The concept could be taken even further and be used for offline applications running solely in the browser without the need of any complex additional software. Data could be downloaded once, stored locally on the filesystem and the analyses could be done later wherever and as often as needed.

All tests were done on state of the art desktop computers. Browser support and performance on mobile devices was not tested and could also be topic for further research.

Another topic that could be interesting for further research is how this method impacts the server compared to traditional WMTS/WCS requests. At the shown method of client-side processing it is possible to build cached tile pyramids of pre-processed GeoTIFF tiles which would not be possible at regular WMTS requests with an almost unlimited number of user-input-variations.

6 Bibliography

[1] M. Haklay, A. Singleton, and C. Parker, ‘Web Mapping 2.0: The Neogeography of the GeoWeb’, *Geogr. Compass*, vol. 2, no. 6, pp. 2011–2039, Nov. 2008.

[2] ‘Web mapping’, *Wikipedia*. 03-Feb-2018.

[3] J. Gaffuri, ‘Toward Web Mapping with Vector Data’, in *Geographic Information Science*, 2012, pp. 87–101.

[4] ‘Vector Tiles’, *Mapbox*. [Online]. Available: https://www.mapbox.com/vector-tiles/. [Accessed: 15-Feb-2018].

[5] ‘Welcome to the OGC | OGC’. [Online]. Available: http://www.opengeospatial.org/. [Accessed: 29-Jan-2018].

[6] ‘Web Map Service | OGC’. [Online]. Available: http://www.opengeospatial.org/standards/wms. [Accessed: 30-Jan-2018].

[7] ‘Introduction to WMS | OGC’. [Online]. Available: http://www.opengeospatial.org/standards/wms/introduction. [Accessed: 30-Jan-2018].

[8] ‘OpenGIS® Web Map Tile Service Implementation Standard’, 06-Apr-2010. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=35326. [Accessed: 04-Dec-2017].

[9] J. Masó, ‘OGC® Web Map Tile Service (WMTS) Simple Profile’, *WMS SWG*, 19-Jan-2016. [Online]. Available: http://docs.opengeospatial.org/is/13-082r2/13-082r2.html. [Accessed: 30-Jan-2018].

[10] ‘OGC® WCS 2.0 Interface Standard’, 2012.

[11] ‘OpenLayers - Welcome’. [Online]. Available: https://openlayers.org/. [Accessed: 02-Feb-2018].

[12] ‘Leaflet — an open-source JavaScript library for interactive maps’. [Online]. Available: http://leafletjs.com/. [Accessed: 02-Feb-2018].

[13] ‘Leaflet: :leaves: JavaScript library for mobile-friendly interactive maps’, 02-Feb-2018. [Online]. Available: https://github.com/Leaflet/Leaflet. [Accessed: 02-Feb-2018].

[14] ‘OpenLayers - Basic Concepts’. [Online]. Available: http://openlayers.org/en/latest/doc/tutorials/concepts.html. [Accessed: 02-Feb-2018].

[15] F. Schindler, ‘geotiff.js: Read raw data from GeoTIFF files’, 01-Feb-2018. [Online]. Available: https://github.com/constantinius/geotiff.js. [Accessed: 02-Feb-2018].

[16] D. Santillan, ‘plotty: Plotting library experiments using WebGL and Canvas2D to apply color scale to a bufferarray object’, 24-Jan-2018. [Online]. Available: https://github.com/santilland/plotty. [Accessed: 02-Feb-2018].

[17] ‘Sentinel-2 - Missions - Sentinel Online’. [Online]. Available: https://sentinel.esa.int/web/sentinel/missions/sentinel-2. [Accessed: 04-Feb-2018].

[18] ‘Sentinel-2 cloudless map of the world by EOX’. [Online]. Available: https://s2maps.eu/. [Accessed: 04-Feb-2018].

[19] W. J. Frampton, J. Dash, G. Watmough, and E. J. Milton, ‘Evaluating the capabilities of Sentinel-2 for quantitative estimation of biophysical variables in vegetation’, *ISPRS J. Photogramm. Remote Sens.*, vol. 82, pp. 83–92, Aug. 2013.

[20] ‘Spatial - Resolutions - Sentinel-2 MSI - User Guides - Sentinel Online’. [Online]. Available: https://earth.esa.int/web/sentinel/user-guides/sentinel-2-msi/resolutions/spatial. [Accessed: 04-Feb-2018].

[21] ‘Mapchete: geospatial processing — Mapchete 0.19 documentation’. [Online]. Available: http://mapchete.readthedocs.io/en/latest/index.html. [Accessed: 15-Feb-2018].

[22] ‘Bernhard Baumrock / thesis’, *GitLab*. [Online]. Available: https://gitlab.com/baumrock/thesis. [Accessed: 04-Feb-2018].

[23] ‘OpenLayers v4.5.0 API - Class: WMTS’. [Online]. Available: http://openlayers.org/en/v4.5.0/apidoc/ol.source.WMTS.html. [Accessed: 04-Feb-2018].

[24] R. Main, M. A. Cho, R. Mathieu, M. M. O’Kennedy, A. Ramoelo, and S. Koch, ‘An investigation into robust spectral indices for leaf chlorophyll estimation’, *ISPRS J. Photogramm. Remote Sens.*, vol. 66, no. 6, pp. 751–761, Nov. 2011.

[25] E. R. Hunt, C. S. T. Daughtry, J. U. H. Eitel, and D. S. Long, ‘Remote Sensing Leaf Chlorophyll Content Using a Visible Band Index’, *Agron. J.*, vol. 103, no. 4, pp. 1090–1099, Jul. 2011.

[26] ‘OpenLayers GitHub (openlayers/src/ol/source/tileimage.js)’, 04-Feb-2018. [Online]. Available: https://github.com/openlayers/openlayers. [Accessed: 05-Feb-2018].

[27] ‘Understanding Resource Timing | Tools for Web Developers’, *Google Developers*. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/network-performance/understanding-resource-timing. [Accessed: 09-Feb-2018].