Master Thesis

Streaming Web-Services for Calculating Live Hydrological Derivatives

By
Christian Autermann
autermann@uni-muenster.de
Matriculation Number: 347760

Submitted to the
Institute for Geoinformatics
University of Münster

In Partial Fulfillment of the Requirements for the Degree
Master of Science in Geoinformatics

May 4, 2014

Supervisors:

Prof. Dr. Edzer Pebesma
edzer.pebesma@uni-muenster.de
Institute for Geoinformatics
University of Münster

Jordan S. Read, PhD
jread@usgs.gov
Center for Integrated Data Analysis
United States Geological Survey

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

About 4.6 million km^2 of earth's continental land surface is estimated to be covered by water of which 91% are constituted of over 300 million lakes (Downing et al., 2006). Previously considered "closed systems" that embody a largely independent ecosystem, recent research has revealed strong influences on global environmental processes like the carbon cycle (Cole et al., 2007). To investigate phenomenons at continental or global scales appropriate analysis software has to be developed that can aggregate, analyze, and ultimately interpret hydrological data at large temporal and spatial scales (Read et al., 2013), i.e. hundreds, thousands, or millions of lakes.

Analysis not only has to compromise lake specific measurements but also other environmental data that interact or interdepend with aquatic ecosystems like catchment properties, local climate, anthropomorphic stressors, local topography specifics or canopy heights (Read et al., 2013).

Given the wide range of input parameter sources, multi-system models are based on data brokers (such as the United States Geological Survey (USGS) Geo Data Portal[1] (GDP)) that are build upon Open Geospatial Consortium (OGC) standards such as Catalogue Service for the Web (CSW, Open Geospatial Consortium, 2007a), Web Processing Service (WPS, Open Geospatial Consortium, 2007d), Web Map Service (WMS, Open Geospatial Consortium, 2006), Web Feature Service (WFS, Open Geospatial Consortium, 2010) and Web Coverage Service (WCS, Open Geospatial Consortium, 2012b), but currently still rely on local algorithms that comprise functionality for statistical quality assurance and quality control as well as the calculation of various metrics related to the physical state of the lakes (often linked with ecosystem function or disturbance). Building standardized and flexible infrastructures for analyzing foundational data used by domain scientists is an important challenge given legacy and heterogeneous architectures.

1. http://cida.usgs.gov/gdp/

One approach for interoperable and scalable analysis is to encapsulate the model in an open and standardized web-based processing framework. This enables the usage of models in web-based model chains that can facilitate other models, translators and existing data brokers. This enables an easy composition of models and data sources from different domains and allows advanced, large-scale and cross-domain analysis. Considering the spatial and temporal extent of available data and possible future extensions to include data from additional domains, the web based processing should be conducted in a streaming manner, i.e. the processing should start before the last chunk of data comes in, and the output should also be available in parts before the processing has completely finished to reduce latency for domain users of the system and to not to require the complete data set to be present at once.

One key component in analysis and monitoring of lakes is the LakeAnalyzer (Read et al., 2011). It is a tool to compute key characteristics of lakes with regards to the lake's thermal stratification and its stability. It analyzes times series data of water temperature measurements obtained using instrumented lake buoys as well as wind speed observations using the lake's bathymetric areas with respect to depth and optional salinity measurements to improve water density calculations. It promotes comparative lake research, which is made possible due to the increased temporal resolution and spatial extent of lake measurements by offering a consistent methodology that can be applied to many types of lakes (ibid.).

LakeAnalyzer allows to predict biological phenomenon (e.g. the likelihood of nutrient upwelling that can cause algal blooms), explain phenological pattern in aquatic organisms, control and monitor the state of a lake as well as to improve the quality of models by comparing their outputs to the physical state of the lake that is computed by the LakeAnalyzer.

Besides a raw data output as Comma Seperated Values (CSV), the LakeAnalyzer features visualizations of the produced stratification and mixing indices, which can be seen in Figure 1.1: Lake Number (see Figures 1.1a and 1.1b, Imberger and Patterson, 1990), metalimnion extent (see Figures 1.1c to 1.1f), Brunt-Väisälä buoyancy frequency (see Figures 1.1g and 1.1h), mode-1 vertical seiche period (see Figures 1.1i and 1.1j, Monismith, 1986), Wedderburn Number (see Figures 1.1k and 1.1l, Thompson and Imperger, 1980), thermocline depth (see Figures 1.1m and 1.1n), u^* (wind stress introduced water friction velocity, see Figures 1.1o and 1.1p) and Schmidt Stability (see Figure 1.1q, Schmidt, 1928; Hutchinson, 1957; Idso, 1973). The LakeAnalyzer is able to error check and/or down sample input time

series and also outputs wind speed (Figure 1.1r) and water temperature (Figure 1.1s).

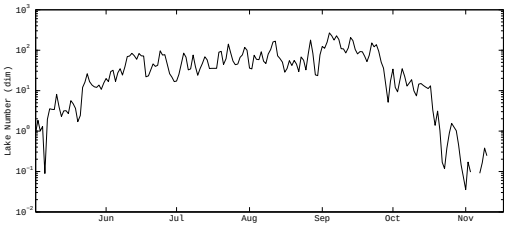
LakeAnalyzer is written as an open source MATLAB application and thus is cross-platform, but requires a MATLAB license and thus is not easily portable. To overcome this issue, a web portal[2] was created that allows the remote LakeAnalyzer execution without a local MATLAB installation.

This thesis work comprises the evaluation, design and prototypical implementation of a web-based lake analysis chain for large data sets and live sensor data. Therefore it will evaluate how an analysis language commonly used by domain experts (in this case MATLAB) can easily be deployed in a web based processing chain, how large scale hydrological data can be processed in a service-based processing chain and whether available web-processing interface definitions support a streaming scenario, or, if not, what is missing to enable streaming processing of geospatial data. Furthermore, this thesis will evaluate how spatial dependencies between streamed features can be modeled and how continuous statistical quality assurance and quality control in the application area of lake ecology can be modeled in a web service chain.

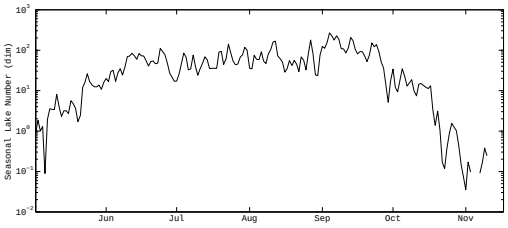
To accomplish this, Chapter 2 will feature an detailed introduction of the established standard for web-based processing of spatiotemporal data, the OGC Web Processing Service. Chapter 3 will conceptualize a WPS implementation that allows the deployment of generic MATLAB based software as WPS processes and thus will create a component to expose the LakeAnalyzer using a standardized web processing interface. To enable large-scale processing of geospatial data, a Streaming WPS will be developed in Chapter 4, which is not only able to conduct analysis of live hydrological sensor data, but can also be applied to various other use cases. Chapter 5 will finally summarize the accomplished results and will give an outlook about possible future developments and conceivable research topics.

2. http://lakeanalyzer.gleon.org/ (Last retrieved May 4, 2014)

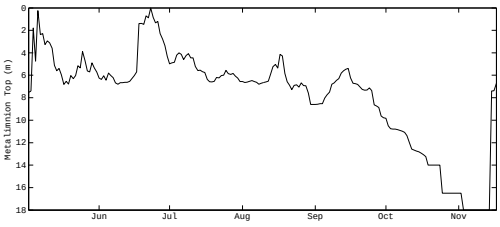
(a) Lake Number



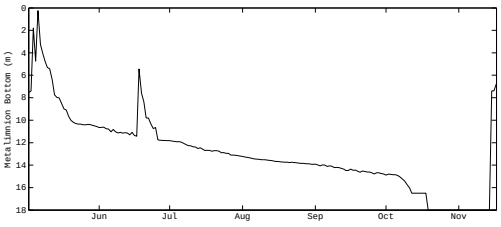
(b) Seasonal Lake Number



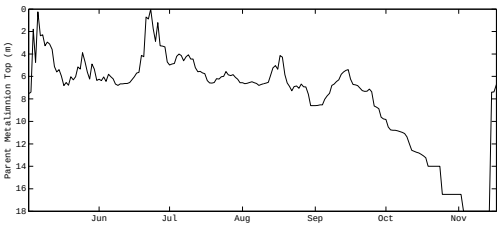
(c) Metalimnion Top



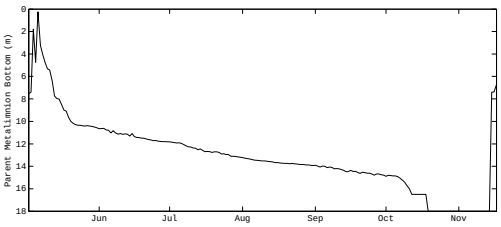
(d) Metalimnion Bottom



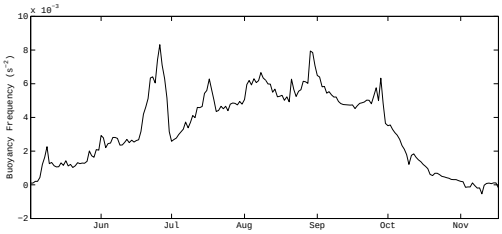
(e) Seasonal Metalimnion Top



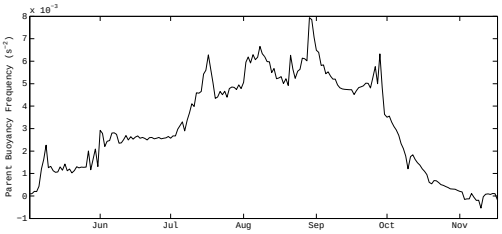
(f) Seasonal Metalimnion Bottom



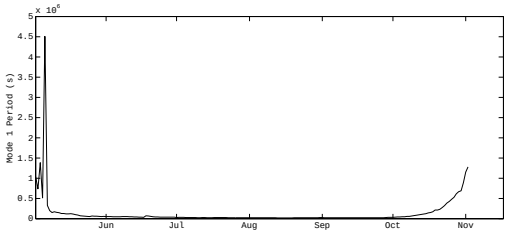
(g) Buoyancy Frequency



(h) Seasonal Buoyancy Frequency



(i) Mode-1 Vertical Seiche Period



(j) Seasonal Mode-1 Vertical Seiche Period

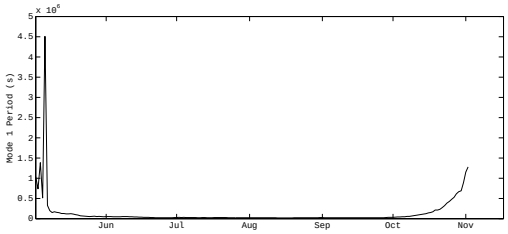
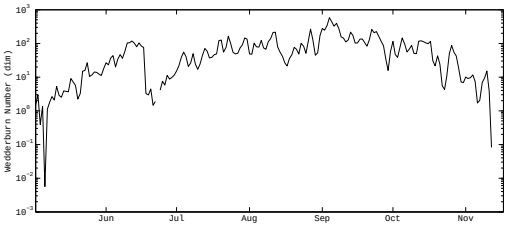
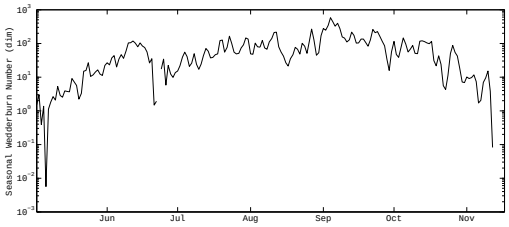


Figure 1.1: Visualization of outputs created by the LakeAnalyzer based on an example data set of the Sparkling Lake, WI, USA *(continued on Page 5).*

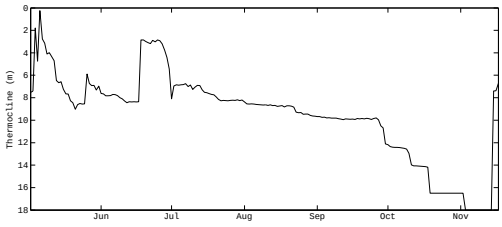
(k) Wedderburn Number



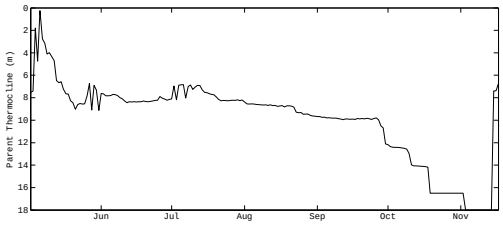
(l) Seasonal Wedderburn Number



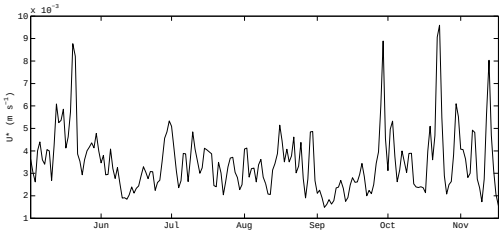
(m) Thermocline Depth



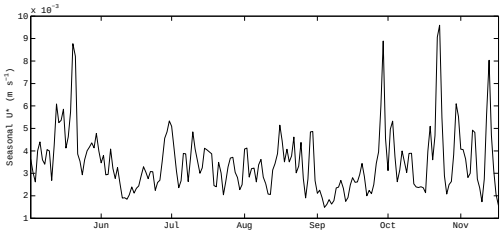
(n) Seasonal Thermocline Depth



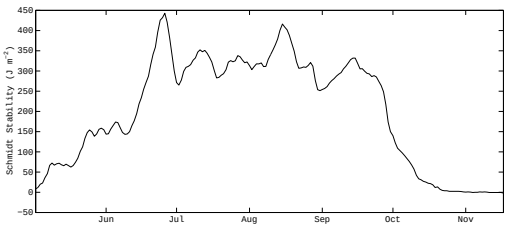
(o) u^*



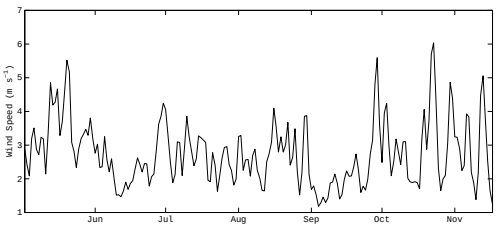
(p) Seasonal u^*



(q) Schmidt Stability



(r) Wind Speed



(s) Water Temperature

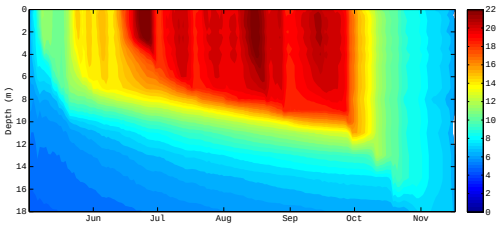


Figure 1.1 (*Cont.*): Visualization of outputs created by the LakeAnalyzer based on an example data set of the Sparkling Lake, WI, USA.

2 Web Processing Service

The Web Processing Service (WPS, Open Geospatial Consortium, 2007d) is the quasi standard for web based processing of spatiotemporal data (Foerster et al., 2012). It is an open service standard specified by the OGC and is embedded in the OGC Web Services Common (OWS) (Open Geospatial Consortium, 2007c) environment. Even though the WPS is mostly used in the geospatial domain, it's interface is not restricted to spatiotemporal data and also can be deployed in other professional contexts. Within the WPS, it is possible to publish and execute models, algorithms or generic calculations and computations in a standardized web service interface, so called processes. The WPS describes a generic interface, that imposes no restrictions on the type of process, their inputs and outputs and so it can encapsulate any kind of algorithm or model. By this, an interoperability is offered, which leads to a number of significant advantages. It adds a layer that hides complexity and permits – by it's consistency across implementations – a high level of reusability, flexibility and scalability. Server and client software implementations become reusable and generic client implementations are possible. Scalable and complex computations, like grid (e.g. Baranski, 2008; Di et al., 2003; Lanig et al., 2008) or cloud computing (e.g. Baranski et al., 2011), as well as super computer processing are hidden behind a simple to use service interface and become accessible.

The WPS specifies mechanisms to discover algorithms and models by offering generic encoding formats for process descriptions and a uniform interface to explore and retrieve these. Besides that, it defines a universal process execution model, that includes request and response encodings, synchronous and asynchronous process executions, long running processes as well as a data encoding for input and output parameters. The interface offers the possibility to retrieve a process output either in a raw format, embedded in a response, or stored in the WPS for later retrieval. This facilitates process chaining and enables the subsequent retrieval of process results. The specification describes three different bindings to access a WPS using the Hypertext Transfer Protocol (HTTP, Fielding et al., 1999). It may

be addressed using key value pair (KVP) encoding with HTTP GET, XML encoding with HTTP POST, or clients may use SOAP (Lafon et al., 2007) to access the web service.

Functionalities are exposed by means of three distinct methods. As every OGC web service the WPS has a *GetCapabilities* method, that can be used to request a detailed description of the service and its capabilities. It offers a service identification structure which contain information about the organization operating the WPS. Also present is a service provider section that contains informational meta data about the service instance which can be used for service discovery. Besides that, detailed information about supported operations, bindings, languages as well as a list of available processes are incorporated.

The detailed description of a single process may be requested by using the *DescribeProcess* operation. Its response contains informational meta data (like textual descriptions) and the process capabilities in regards to asynchronous execution and response/output storage. Comprehensive information about required and supported inputs, their cardinalities, supported formats and restrictions, and available outputs as well as their supported formats are also included.

Processes are executed using the *Execute* operation. Besides the necessary input parameters and information about their encoding, the request describes selected outputs that should be generated by the process. Furthermore, it informs the WPS whether the process should be executed synchronously or asynchronously and how the results of the process should be encoded.

Typical interaction patterns of the Web Processing Service are depicted in Figure 2.1. During process discovery, *GetCapabilities* and *DescribeProcess* are used to request a list of available processes and their descriptions. Process execution takes place either synchronously or asynchronously by issuing an *Execute* request to a specific process. In the case of asynchronously process executions, the WPS returns an URL to an *ExecuteResponse* which is continuously updated and which the client can request periodically to get the current process status.

The WPS describes three basic types of input and output parameters: *literal*, *complex* and *bounding box* parameters. Complex data parameters are data structures that can be described by a mime type, an encoding and a schema. They can represent raster data, XML structures such as Geography Markup Language (GML) (Open Geospatial Consortium,

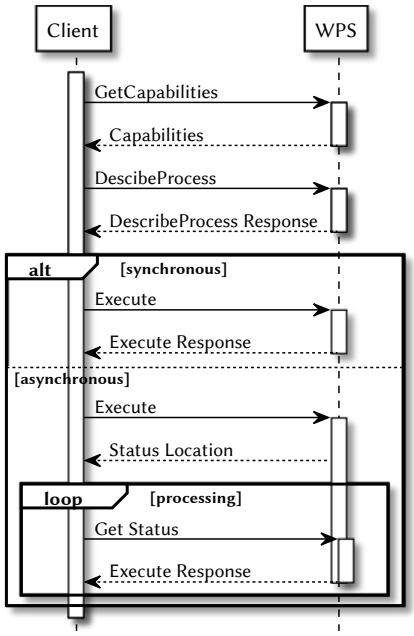


Figure 2.1: Typical interaction patterns of the Web Processing Service: process discovery using *GetCapabilities* and *DescribeProcess* and synchronous as well as asynchronous process execution using *Execute*.

2007b) feature collections, CSV or any other type of data. This data can be supplied embedded in XML or as a reference to an external HTTP resource. Referenced complex data structures may be requested by using HTTP GET or POST and can transport HTTP headers and any body payload (or reference to one). By this, chaining of WPS processes can easily be implemented, either by referencing a previous generated output or even by encoding another *Execute* call into the reference. Literal data can be represented by a single string value. The value is described by a data type and can be accompanied by a unit of measurement. Typical data types include single strings, URIs, boolean values, dates and integral or decimal numbers. Bounding box data represents a rectangular region of arbitrary dimension which is described by a coordinate reference system (CRS).

As shown in this chapter, there are several benefits that can be expected by using the WPS. Especially in the context of domain specific models, the interoperability and reusability can be increased significantly. Until now, the LakeAnalyzer can not be deployed in a web based processing chain and has to be executed in a manual procedure. Web based execution is currently realized by using a web form that allows remote execution of the LakeAnalyzer. This way of proceeding presupposes the recourse to specialized software or scripts for

automation and can be characterized as very disadvantageous, e.g. for the reuse of the developed model in other projects. The following section focuses on a WPS that allows the deployment of models developed in MATLAB – like the LakeAnalyzer– as WPS processes with the purpose of profiting from the positive aspects of standardized web based processing solutions.

3 MATLAB WPS

MATLAB[1] is a closed source, commercial software by The MathWorks, Inc. for numerical computation, visualization and programming. It features a high-level programming language as well as an cross-platform (Windows, Linux and Mac OS X) interactive desktop environment. Initially developed for matrix computations (hence *MAT*rix *LAB*oratory)[2], today MATLAB is widespread across different domains in academics, engineering and industry. The base program is extensible by using so called *toolboxes*, that add functionalities for various domains, like statistics, curve fitting, neural networks, image processing, economics, bioinformatics or signal processing. Besides that, functions, algorithms, files or toolboxes can be installed through *MATLAB Central*, a repository of user contributions. These are mostly licensed under the two-clause BSD license[3][4].

Creating a specific WPS process implementation for the LakeAnalyzer would be possible. Considering the wide spread usage of MATLAB based scripts and applications, a generic solution, that enables the easy deployment of MATLAB based functionalities as WPS processes would have a huge benefit for the geospatial community as well as for the the acceptance of the WPS across disciplines. A generic *MATLAB WPS* would not only open the LakeAnalyzer for an interoperable usage in existing web processing chains, but would also make existing models and algorithms implemented in MATLAB instantly available to a larger audience and can increase reusability of software components and exchange between different areas of research, development and business. Considering the diversified fields MATLAB is used in, a software component such as a MATLAB WPS can not assume an extensive programming experience beyond MATLAB. Domain experts developing models or algorithms in MATLAB should be able to offer a MATLAB script or function as a WPS process using a simple and straightforward procedure, without any knowledge

1. http://www.mathworks.com/products/matlab/
2. http://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html
3. http://opensource.org/licenses/bsd-license.php
4. http://www.mathworks.com/matlabcentral/FX_transition_faq.html

of other programming languages or a comprehensive expertise in web services or their development. To accomplish this, switch from MATLAB to other languages should not be required, and rather complex and verbose process descriptions should not be manually be written, but automatically generated. A key goal of the MATLAB WPS is to expose existing models and algorithms as WPS processes. Therefore, the procedure to convert a MATLAB script or function should not require intrusive changes to be compatible with the MATLAB WPS.

Approaches to offer data analysis and modeling languages like MATLAB as WPS processes do already exist. Specially emphasized should be the *WPS4R* (Hinz et al., 2013) project that creates WPS processes from scripts written for the statistical analysis environment *R* (R Core Team, 2014). Written as a module for the 52°North WPS implementation, it shares many requirements and challenges with a MATLAB WPS. R is also an environment used mostly by domain experts and features a massive amount of existing models and algorithm implementations. These are worth to be opened to the web processing environment and to be made available to a broader user base using interoperable standards like the OGC Web Processing Service.

Listing 3.1: Example for a comment containing annotations used by WPS4R (Hinz et al., 2013).

```
# wps.des: id = process, title = "my script",
#          abstract = "analyze 42 things"

# wps.in: id = myFactor, type = integer, title = "numerical factor",
#         abstract = "the number to be used for factorization",
#         value = 1, minOccurs = 0, maxOccurs = 1;

# wps.out: id = myResult, type = string,
#          title = "factorized output",
#          abstract = "output number as text in scientific notation (a x
    10^b)";
```

WPS4R takes an R script and executes it on a remote or local R instance using *Rserve* (Urbanek, 2003). In contrast to the WPS interface which explicitly states types of input and output parameters to allow service discovery and the usage of generic clients, R is a weakly and dynamically typed language. By this, the WPS is not able to parse the script and determine appropriate input and output parameter types, as these are only available at runtime. To bind static types to input and output parameters, an annotation mechanism was developed which is also capable to detail input/output and process meta data.

In contrast to other programming languages, like Java (JSR-175 Experts Group, 2004) or C# (European Computer Machinery Association, 2006), R does not feature a native annotation mechanism. Because of this, the annotations are encoded as comments featuring special keywords (*wps.in*, *wps.out* and *wps.des*), followed by a key value list representing the necessary information to generate a process description (see Listing 3.1). During process execution, WPS4R will populate the described input parameter variables using WPS inputs, execute the script, read the specified output variables from the R session and transform them to WPS outputs. The usage of annotations embedded in comments support the deployment of R functionalities as WPS processes by providing a single script file that the WPS4R can parse.

Literal input parameters are translated into native R types, whereas complex inputs are transferred as files to a temporary working directory. Complex input and output parameters have to be described by a single keyword, denoting the mime type of the parameter, that has to be registered to WPS4R using a configuration file. Describing complex inputs and outputs using *schema* or *encoding*, or using about any mime type without changing the WPS4R configuration, is not possible. This may be caused by the reduced expressiveness through the usage of a structureless description format (e.g. denoting multiple supported complex input formats, would be hard to specify). Scripts are run on globally configured Rserve connections. Different remotes for different processes or a load balancing between multiple remote nodes running R are not possible. Furthermore, the easy deployment of scripts consisting of multiple files is currently not possible.

The comment based approach taken by WPS4R has several advantages like having WPS configuration and actual code side by side (which results in less maintenance effort), but also introduces considerable drawbacks, especially if the annotation mechanism should be applied to MATLAB. Conveying important information in comments can be problematic. Even though there are many examples where comments are used (e.g. to generate documentation as seen on the example of Javadoc (Oracle Corporation, 2013)) these are often standardized at language level or include a large user base and a wide support in editors and development environments. The syntax of a custom comment based annotation mechanism as used in this approach, can not be verified in editors or interpreters. By this and the unstructured notation of comments, the approach becomes heavily prone to user error, that can not be detected before the deployment to a WPS instance. Additionally, annotations are not actually bound to any language construct, but just happen to be in the same file.

Typical MATLAB programs would not benefit from combining annotations and scripts in a single file, as it is common practice – or even a requirement to access a function from outside – to place a function in its own file. By this, MATLAB programs tend to consist of multiple files, and can not easily deployed as single script files.

...

In contrast to R, MATLAB offers multiple return values of functions as a native language feature (see Listing 3.2). Through this, MATLAB functions are able to directly represent a WPS process, and the MATLAB WPS should use MATLAB functions instead of scripts to offer functionalities as WPS processes. As stated before, MATLAB is a weakly and dynamically typed language, and the parsing of a function signature can not create a statically typed binding as the WPS standard requires. For this, an additional description mechanism has to be developed that allows the semi-automatically generation of process descriptions. This should be done without extensive knowledge of web service development or programming languages. Also the deployment of existing MATLAB functions should be a straightforward non-intrusive process. Similar to R, MATLAB instances are single threaded, and so can only process one WPS process execution a time. Moreover, and contrary to R, opening the MATLAB workbench even in a headless mode (i.e. without any user interface) can take considerable time. This requires an efficient usage of MATLAB instances, especially the reuse of already started MATLAB instances to reduce latency of process executions. Complex inputs should be usable inside of MATLAB without restrictions to any format, and without the need to change any configuration files.

Listing 3.2: MATLAB example function that calculates statistical characteristics (mean and standard deviation) of an input vector.

```
function [ave, sd] = stat(x)
    n   = length(x);
    ave = sum(x)/n;
    sd  = sqrt(sum((x-ave).^2/n));
end
```

This chapter will outline the conceptualization and implementation of a MATLAB WPS by describing its architecture and configuration mechanism. Furthermore, details of the conversion between MATLAB and WPS types will be discussed and legal implications of offering commercial software as web services will be shortly examined. Finally, the generic

capabilities of the MATLAB WPS will be used in order to offer the LakeAnalyzer as a WPS process.

3.1 Architecture

The MATLAB WPS features a multi-tier architecture to offer MATLAB functions as Web Processing Service processes. A detailed sequence diagram depicting a MATLAB WPS process execution can be seen in Figure 3.1. An incoming WPS *Execute* request is accepted by the MATLAB WPS (step 1). The *Execute* request is verified (e.g. no missing inputs, inputs within the range described by the process description, etc.) and then translated into a MATLAB request (step 2). This request is send via a WebSocket connection to a configured MATLAB server (step 3). The MATLAB server maintains a pool of MATLAB instances and will dispatch the request to one of these as soon as one becomes available (step 4). The instance transforms the MATLAB request into MATLAB syntax (step 5) and evaluates the MATLAB command in an associated MATLAB session. After this (step 6), the return values are read from the session (step 7) and encoded as a MATLAB response (step 8). It is then passed through the MATLAB server (step 9) to the MATLAB WPS (step 10). The MATLAB WPS process translates the MATLAB response to a WPS *Execute* response (step 11) and returns it to the client (step 12).

Besides an option to run the MATLAB server locally, all communication between the MATLAB WPS and MATLAB server is done over WebSockets (Fette and Melnikov, 2011). WebSockets are defining a TCP-based protocol that creates a bidirectional communication channel between client and server. A primary goal of WebSockets is to bring the benefits of efficient full-duplex communication to the web browser environment. This is accomplished by an HTTP compatible socket initiation mechanism (see Listing 3.3). A client opens a new WebSocket by issuing an HTTP request to the server, in which it requires an upgrade to the WebSocket protocol. Afterwards, the connection is kept open and both client and server can send messages to the opposing party. These messages are transported using one or more text or binary frames and allow an efficient bidirectional information exchange. By using HTTP for the initial handshake, WebSockets can be used in most proxy setups and despite the presence of firewalls that filter non HTTP traffic and can facilitate HTTP's access control mechanisms and client side security measures like Cross-Origin Resource Sharing (CORS, van Kesteren et al., 2014).

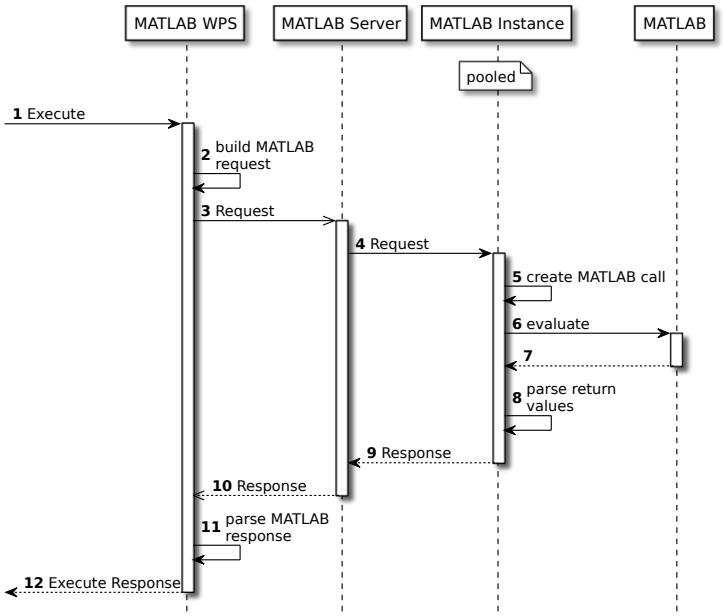


Figure 3.1: Sequence diagram of a MATLAB WPS process execution.

Listing 3.3: WebSocket opening handshake using a HTTP upgrade request (Fette and Melnikov, 2011).

```
> GET /matlab HTTP/1.1
> Host: example.com
> Upgrade: websocket
> Connection: Upgrade
> Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
> Sec-WebSocket-Version: 13
> Origin: http://example.com

< HTTP/1.1 101 Switching Protocols
< Upgrade: websocket
< Connection: Upgrade
< Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Even though the opening handshake is using HTTP, WebSockets do not conform to the HTTP protocol. To ensure that a web server can handle WebSocket connections, the client sends the header *Sec-WebSocket-Key* in the opening request, containing 16 bytes of random data in base 64 encoding (Josefsson, 2006). The server has to append the Globally Unique Identifier (GUID, Leach et al., 2005) *258EAFA5-E914-47DA-95CA-C5AB0DC85B11* to the header value and return the base 64 encoded SHA-1 (National Institute of Standards and Technology, 2008) hash sum using the *Sec-WebSocket-Accept* header field. Because of the

incompatibility with the HTTP protocol, WebSockets define two separate URL schemes: *ws* for normal WebSocket connections and *wss* for secure WebSocket connection, which resemble the HTTP and HTTPS protocol and share their default ports 80 and 443.

The WebSocket protocol is accompanied with an HTML5 JavaScript Application Programming Interface (API) (Hickson, 2012) that is implemented in all recent versions of major desktop browsers[5] (Deveria, 2014). Besides that, WebSocket client and server implementations for nearly all programming languages (e.g. R[6], C[7], C#[8], Java[9]) exist.

As previously noted, function calls are used as the central element of MATLAB WPS processes. Using the native language feature of multiple return values, WPS processes can be represented as MATLAB functions one to one. The MATLAB WPS is not designed to easily interface MATLAB with the WPS implementation to allow process development from within the WPS, but to allow the deployment of any MATLAB model using a WPS. Because of this, the MATLAB WPS only offers functionalities to evaluate a single function call and is not required to evaluate scripts, parse MATLAB code or maintain variable references. By this, a very thin implementation is possible and the configuration and maintenance efforts are reduced to a minimum.

The components *MATLAB server* and *MATLAB instance* as shown in Figure 3.1 are developed separately from the MATLAB WPS and can be easily used in other contexts. This *matlab-connector*[10] consists of a small Java CLI application (the server) and an associated Java client library used in the MATLAB WPS, that offers a simple API to build MATLAB requests. The server component is started on the machine on which MATLAB is installed, and then offers a configurable amount of headless MATLAB instances using a small WebSocket server. The MATLAB instance communicates with a Java Virtual Machine (JVM) exposed by the MATLAB program using a Java Remote Method Invocation (RMI) wrapper called *matlabcontrol*[11]. As previously noted, MATLAB instances are, even in headless mode, heavy weight applications that require a considerable amount of resources and time to start. MATLAB instances are created at server startup and then are used to process re-

5. And with the exception of Opera Mini also mobile browsers.
6. https://github.com/rstudio/R-Websockets
7. http://libwebsockets.org/trac/libwebsockets
8. http://msdn.microsoft.com/library/system.net.websockets.websocket.aspx
9. https://jcp.org/en/jsr/detail?id=356
10. The matlab-connector was initially developed for the UncertWeb project (http://www.uncertweb.org/), but was heavily extended for this thesis.
11. https://code.google.com/p/matlabcontrol/

quests. By reusing and preallocating a fixed amount of instances, the pooling of MATLAB instances reduces latency for WPS processes and saves resources on the server machine.

3.2 Configuration

Because of the aforementioned problems regarding comment annotations, the MATLAB WPS features another configuration mechanism. Process configurations are conveyed using YAML (Ben-Kiki et al., 2009) which facilitate a particular human-readable syntax. It allows easy structuring of data without delimiters like quotation marks or braces, but allows these e.g. to enable a more compact syntax. The structure of YAML has close resemblance with JSON (which is actually a valid subset of YAML since version 1.2) and features the same basic types of scalars, sequences and associative arrays (maps), but has additional features that make it more expressive. This includes comments, multi-line strings, references, multi-document files, sets, complex key types for maps, ordered/unordered maps and maps that allow duplicate keys.

Listing 3.4: MATLAB process configuration describing the function in Listing 3.2.

```
---
function: stat
connection: local
identifier: matlab.stat
version: 1.0.0
title: Arithmetic Mean and Standard Deviation
abstract:>
  Calculates the arithmetic mean and
  standard deviation of a numerical vector.
inputs: # the input definitions
  - identifier: x
    type: double
    maxOccurs: unbounded
    title: input vector
    abstract: A numerical input vector.
outputs: # the output definitons
  - identifier: ave
    title: Arithmetic Mean
    abstract: The arithmetic mean of the input vector.
    type: double
  - identifier: sd
    type: double
    title: Standard Deviation
    abstract: The standard deviation of the input vector.
...
```

Configuration files for the MATLAB WPS can contain multiple process configurations expressed as an associative array. These are describing a MATLAB function, their input and outputs as well as where the function should be executed. It resembles the basic structure of a WPS process description while concealing the verbosity and complexity of XML. Listing 3.4 shows an example process configuration for the function displayed in Listing 3.2. The process description generated from the YAML configuration can be found in Listing 3.5.

Listing 3.5: Process description generated from the configuration in Listing 3.4 (see Appendix E for omitted XML namespaces).

```
<ProcessDescription wps:processVersion="1.0.0">
  <ows:Identifier>matlab.stat</ows:Identifier>
  <ows:Title>Arithmetic Mean and Standard Deviation</ows:Title>
  <ows:Abstract>Calculates the arithmetic mean and standard deviation of
      a numerical vector.</ows:Abstract>
  <DataInputs>
    <Input minOccurs="1" maxOccurs="2147483647">
      <ows:Identifier>x</ows:Identifier>
      <ows:Title>input vector</ows:Title>
      <ows:Abstract>A numerical input vector.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
  </DataInputs>
  <ProcessOutputs>
    <Output>
      <ows:Identifier>ave</ows:Identifier>
      <ows:Title>Arithmetic Mean</ows:Title>
      <ows:Abstract>The arithmetic mean of the input vector.</
          ows:Abstract>
      <LiteralOutput>
        <ows:DataType ows:reference="xs:double"/>
      </LiteralOutput>
    </Output>
    <Output>
      <ows:Identifier>sd</ows:Identifier>
      <ows:Title>Standard Deviation</ows:Title>
      <ows:Abstract>The standard deviation of the input vector.</
          ows:Abstract>
      <LiteralOutput>
        <ows:DataType ows:reference="xs:double"/>
      </LiteralOutput>
    </Output>
  </ProcessOutputs>
</ProcessDescription>
```

Top level attributes are describing the process itself, whereas *inputs* holds a sequence of input descriptions and *outputs* a sequence of output descriptions in the very same order

the function is defined. The function to describe is denoted by the keyword *function*. *identifier*, *title* and *description* are directly mapped to their equivalent in the OGC name space. The attribute *maxOccurs* holds either an integral number or the special value *unbounded* which will be translated to the platform specific maximum possible value (typically the greatest possible integer value). Data types, described under the keyword *type*, are translated to their respective XML data type. Complex data types can be described using a map containing a combination of *mimeType*, *schema* and *encoding*. Bounding box inputs are described using a map containing the keyword *crs*, which holds one or more supported CRS.

The attribute *connection* denotes how the function should be executed. The keyword *local* will cause the MATLAB WPS to start a pool of MATLAB instance in the current working directory. The function has to be either at this path or at any other path searched by MATLAB. Other possible values for *connection* are URIs in the *ws*, *wss* or *file* scheme. The latter will start a connection pool inside the specified directory, while a WebSocket URL will cause the MATLAB WPS to connect to the remote server and will run the function there. In both cases, the file containing the function has to be able to be found in the MATLAB search path.

Through the very clear and concise YAML notation, complex process description can be easily written in a human readable format, which is way easier to maintain than custom annotations in inline comments. It results in a less error prone procedure for unexperienced domain experts, whereas advanced users are able to benefit from advanced YAML features. Furthermore, future enhancements and additions can be easily implemented backwards compatible.

3.3 Type Mapping

MATLAB, like any other language, has a wide variety of data types. These include numeric types – floating point numbers in single (32 bit) and double (64 bit) precision and signed and unsigned integers in 8, 16, 32 and 64 bit size – logical, character/string types as well as structures, tables, cell arrays and function handles. Except for the latter, all of these types have the form of (possible multidimensional) arrays (The MathWorks, Inc., 2014).

As previously described, the Web Processing Service specification knows three different types of data: literal, complex and bounding box data. WPS Literal data is mostly converted

to their respective native MATLAB data type, but due to limitations in the MATLAB API, this is not always possible. The API exposed by MATLAB transfers every numerical type as floating point numbers of double precision. By this, an efficient handling of other basic data types like integral numbers or single precision floating point numbers is not possible. Within the WPS specification and implementation, these data types are each handled differently, but due to the limitations exposed by the MATLAB interface, MATLAB processes have to reduce precision on their own in order to reduce memory usage.

Single and multiple occurrences of input parameters can be handled in MATLAB in the very same way, because every basic data type consists not only of a single value, but an array of it's type. The sole exception are string based data types, which are represented as an array of characters. Placing several strings in an array results in an concatenated string and so a MATLAB *cell* is used for these data types. Boolean values are represented as *logical* 0 or 1 or a respective array and time stamp values are converted to their numerical representation[12].

Bounding box input data is mapped to a *struct* consisting of the fields *crs* and *bbox* holding the CRS identifier and a two-dimensional array with the upper and lower corner of the bounding box respectively. This format is also expected for bounding box outputs.

Complex data is neither parsed nor converted using the MATLAB WPS. It is transferred to a temporary file and passed to the MATLAB function as a file name. For complex outputs, the MATLAB functions saves them to a temporary file and returns the file name. The file is read by the MATLAB WPS and deleted when the process finishes. By delegating the parsing of complex data inputs to the MATLAB function, the WPS is independent from specific data formats – both in case of specific MATLAB classes and in case of different XML or binary encodings at the WPS end – and can easily be adopted to existing MATLAB models.

The usage of complex outputs is currently limited to a single format. Even though the WPS specification allows the request of different formats (e.g. a raster or image can be requested as PNG, JPEG or TIFF, or a feature collection may be requested in different XML schemata), the MATLAB WPS does not offer this feature to MATLAB processes. This is owed to the MATLAB based handling of complex inputs. To become independent of file formats and encodings, the MATLAB WPS can not be used to transform inputs or outputs between different formats. While the inputs and outputs of different format still could

12. A double value containing the fractional number of days since the January 0, 0000.

be created and consumed on the MATLAB process side, this possibility was neglected to ease MATLAB process development and to allow a more simple transformation of existing MATLAB models.

MATLAB lacks a value to represent the absence of a value (often denoted as *null*, *nil*, *none* or *nothing* in other programming languages). Even though MATLAB supports optional parameters in function calls, it does not support named function parameters, and a function can only interpret the amount of input parameters to determine if an optional parameter is present or not. As WPS processes can contain a multitude of optional input parameters, the value *NaN* (IEEE Task P754, 2008), which represents an undefined or unrepresentable numeric value and so comes close to a null value, is used to transport absent optional input parameters, regardless of their type.

The WPS specification offers the possibility to only request specific outputs of a process. This enables the process to only compute the outputs that are really needed and thus can reduce the time needed for process executions. The MATLAB WPS currently does not feature not mechanism and MATLAB functions are required to compute all outputs regardless which are requested by a client. To overcome this issue, the requested output identifiers could be saved in a globally accessible environment variable. In addition to this, other contextual information could be conveyed using this method, e.g. the WPS service URL, which was used to execute the process or other meta data that the function may use.

A list of literal (based on Biron and Malhotra, 2004), bounding box and complex data types and their mapping to MATLAB types can be seen in Table 3.1. Structured data like structs, multidimensional arrays, cells or other objects can not be used as process outputs or inputs, as the WPS specification lacks support for such types. A MATLAB process has to create a XML application schema or transform the structures to another file based data typed that can be transported as WPS complex outputs.

3.4 License Issues

MATLAB usage is, as any software, restricted by the software's license. MATLAB is a proprietary and commercial product and as such, the software and its usage is more restricted than e.g. an open source software such as the R Project. Relevant for the MATLAB WPS

Table 3.1: Mapping between WPS data types and MATLAB types. Absent optional parameters are denoted by *NaN* (1×1).

		MATLAB Type	
		Single	**Multiple**
Complex Data		char $(1 \times m)$	cell of chars $(1 \times n)$
Bounding Box Data		struct (1×1)	cell of structs $(1 \times n)$
Literal Data	xs:string xs:anyURI	char $(1 \times m)$	cell of chars $(1 \times n)$
	xs:byte xs:short xs:int xs:long xs:integer xs:double xs:float	double (1×1)	double $(1 \times n)$
	xs:boolean	logical (1×1)	logical $(1 \times n)$
	xs:dateTime	double/datenum (1×1)	double/datenum $(1 \times n)$

is section 4.8 of *The MathWorks, Inc. Software License Agreement* (The MathWorks, Inc., 2013):

> *"4. LICENSE RESTRICTIONS. The License is subject to the express restrictions set forth below. Licensee shall not, and shall not permit any Affiliate or any Third Party to: [...] 4.8. provide access (directly or indirectly) to the Programs via a web or network Application, except as permitted in Article 8 of the Deployment Addendum;"*

As the MATLAB WPS offers MATLAB functionalities through a web service interface, the usage is highly restricted, as the referenced *Deployment Addendum* (The MathWorks, Inc., 2013) states:

> *"8. WEB APPLICATIONS. Licensee may not provide access to an entire Program or a substantial portion of a Program by means of a web interface.*
>
> *For the Network Concurrent User Activation Type. Programs licensed under the Network Concurrent User Activation Type may be called via a web application, provided the web application does not provide access to the MATLAB command line, or any of the licensed Programs with code generation capabilities. In addition, Licensed Users may not provide*

access to an entire Program or a substantial portion of a Program. Such operation of an application via a web interface may be provided to an unlimited number of web browser clients, at no additional cost, for Licensee's own use for its Internal Operations, and for use by Third Parties.

For the Network Named User and Standalone Named User Activation Types. Programs licensed under the Network Named User and Standalone Named User Activation Types may be called via a web application, provided the web application does not provide access to the MATLAB command line, or any of the licensed Programs with code generation capabilities, and such application is only accessed by designated Network Named User or Standalone Named User licensees of such Programs.

Programs licensed under any other Activation Type may not be called via a web interface."

Only the *Network Concurrent User Activation Type* is allowed to offer MATLAB scripts and functions as long it does not offer access to the MATLAB command line interface. *Network and Standalone Named User* license types require an additional authentication mechanism in place in order to restrict access to the web application. As the MATLAB WPS does not offer the possibility to access the MATLAB command line interface or substantial portion of MATLAB, but restricts access to configured MATLAB function calls, customers owning a license of the first type are allowed to deploy a WPS offering MATLAB processes to an open network, whereas users of the second class of licenses are still allowed to deploy them with an additional authentication mechanism. On the other hand, using a pool of MATLAB instances on a remote server introduce additional problems in regard of the license. In theory, these MATLAB instances can be used to perform about any function call, and thus provide access to the MATLAB command line interface. Even though the access is restricted to simple function calls and does not allow variable declaration, nested function calls or function definitions, it may be considered a license violation to deploy this infrastructure in a public environment.

A conclusive analysis of the legal implications of the system is out of the scope of this thesis, but certainly should be done before a system facilitating the MATLAB WPS or any of its components is deployed in a public or productive environment.

3.5 LakeAnalyzer WPS

Using the generic capabilities of the MATLAB WPS, the LakeAnalyzer can easily be exposed as a WPS process. In its original form, the LakeAnalyzer takes a folder and a lake name as input parameters and will search for appropriate named files in that directory. Besides CSV input files, it also searches for two configuration files containing parameters for analysis and plotting of outputs. Output files are also created in that directory with appropriate names (Read and Muraoka, 2011).

As this approach conflicts with the allocation of complex input parameters in temporary files by the MATLAB WPS as well as with the concept of making configuration parameters separate WPS input parameters, the structure of the LakeAnalyzer has to be broken up. By separating configuration and analysis in two different functions, two wrapper functions can be created that allow the execution of the LakeAnalyzer either as a standalone program or as a WPS process. In the first case, the function simply encapsulates the traditional configuration behavior by reading parameters from configuration files and using the supplied folder for input and output files. For the second case, the configuration as well as the location of input files are passed as separate function arguments (see Appendix A) and output files are allocated in a separate folder. While the original LakeAnalyzer does not provide any function return parameters, the WPS wrapper function returns the file names of the output files and by this, handing control over these files to the MATLAB WPS. By encoding configuration files as distinct input parameters, generic WPS clients are able to present the configuration options to the user without knowledge of specific configuration file formats.

The wrapper function is described in a separate YAML configuration file (see Appendix B) containing the necessary meta data to publish the function as a WPS process. It assigns the function to the process identifier *org.gleon.LakeAnalyzer* and expresses process input and output definitions (taken from the LakeAnalyzer user manual, Read and Muraoka, 2011). As the WPS specification is not able to express dependencies between specific outputs and inputs, and due to the fact that the MATLAB WPS requires all outputs, all necessary input parameters are mandatory and only the globally optional water level and salinity files are optional for the WPS process.

After loading the configuration file into the MATLAB WPS, it will create a WPS process (see Appendix C) offered under the specified identifier and will direct all *Execute* requests

to the MATLAB server specified in the configuration file.

...

4 Streaming WPS

In contrast to conventional data processing, such as the method used in the WPS, streaming processing approaches show considerable benefits. Regarding to time efficiency and with reference to the already mentioned problems of processing substantial large data sets or live data, the development of a streaming enabled WPS seems to be of great value.

Data streams can be seen as an abstract concept that stands in contrast to conventional batch data. Data streams are (possibly infinite) sequences of data items (or chunks) that become available over time, whereas conventional batch data describes a pile of data that is either completely available or not. The abstract concept of streaming can be observed across different technologies and fields of application. Starting from the concept of pipes and filters on unix-like operating systems, over interprocess communication using sockets (either local or over a network, Buschmann et al., 1996), the ubiquitous usage in programming languages (as a concept of I/O or in functional programming languages in the form of inductive data type definitions), general-purpose computing on graphics processing units (GPGPU) to modern media streaming solutions like RTP, RTCP, RTSP (Schulzrinne et al., 2003, 1998) or SIP (Rosenberg et al., 2002). The best way to illustrate this concept is to look on its most popular usage form: media streaming. The conventional approach to view a video or play a sound file over a network is to download the file and to play it locally. Depending on the encoding and compression which has been applied to the media file, it is not possible to play the file until the download is finished. By sending smaller parts of the media file (e.g. one or more single frames) over the network, the time to start playing is reduced to a great degree. Suitable players are now able to play this stream of frames long before the whole file is transmitted. Besides the on-demand streaming of media (the streamed file is completely available on the remote side), the transmission of live audio or video becomes possible by transferring audio or video frames as soon as they are recorded.

The concept of streaming processing extends this simple pattern by not only accepting a

stream of input data, but also by generating a stream of output data. The processing takes place on small chunks of the input data instead of the complete data set. By sequentially processing the stream, software is able to process very large or infinite data sets because the complete data set neither needs to be kept in memory nor it is needed to be stored. This permits the analysis of live data, e.g. the evaluation of continuously collected sensor data. Also the initial response time (the time until the first outputs of a program are available) is equally reduced as in media streaming. Reducing the latency of initial data output has various advantages, e.g. earlier appearance of errors (and by this the possibility to stop processing to save computing resources and time and thus also reducing financial costs) or the ability to develop more responsive end user solutions, e.g. by gradually updating a data visualization instead of presenting the data after waiting for the complete result.

In the case of spatiotemporal data, streaming processing is especially useful and advisable, as data sets tend to become rather large and the analysis of real-time data can have great benefits. Especially as spatiotemporal data is often adequate for streaming: spatial data sets are often aggregates or collections that can be easily broken down into smaller parts (like single features, observations or tiles). On the other side, spatiotemporal data has the salient characteristic of showing strong dependencies to nearby data and thus can be difficult to analyze using non-random-access paradigms like streaming. The case of inter-feature dependencies needs to be considered when transferring the concept of streaming to spatiotemporal processing. Algorithms used in streaming are required to operate on smaller chunks of the complete data set. Computations that require global knowledge are not expected to have any advantage from streaming. For example, graph algorithms like Dijkstra's algorithm (Dijkstra, 1959) can not start the computation before the complete graph is available.

Streaming processing can be divided into three categories that differ from conventional processing (see Figure 4.1a). Characteristic for input streaming (Figure 4.1b) is the parallel occurrence of input and processing with a subsequent output after processing finished. On the other hand, output streaming processing describes the isolated input supply and parallel processing and output (Figure 4.1c). These two approaches are combined in the third category, full input and output streaming (Figure 4.1d), in which input, processing and output take place concurrently. Despite their respective level of concurrency, all three categories have the very same advantage. By parallelizing processing and input and/or output, the overall execution and initial response time is appreciably shorter. Full input and output streaming enabled processes have the additional advantage to be able to process

indefinite large data sets by processing each input data chunk separately and outputting an output data chunk for each of them. Through this, the analysis of live sensor data can be accomplished. Each of these categories of processing demand different requirements from the process or algorithm. To create a stream, the data set needs to be divided into smaller chunks; input streaming enabled algorithms need to be able to operate on each of these chunks separately and output streaming enabled processes need to be able to produce intermediate results. Input streaming would not result in any benefits for algorithms requiring global knowledge of the data set because they can not start processing until all data chunks have been arrived. Processes that result in a single output value, for which the processing has to be completed, offer no advantage when they are output streaming enabled.

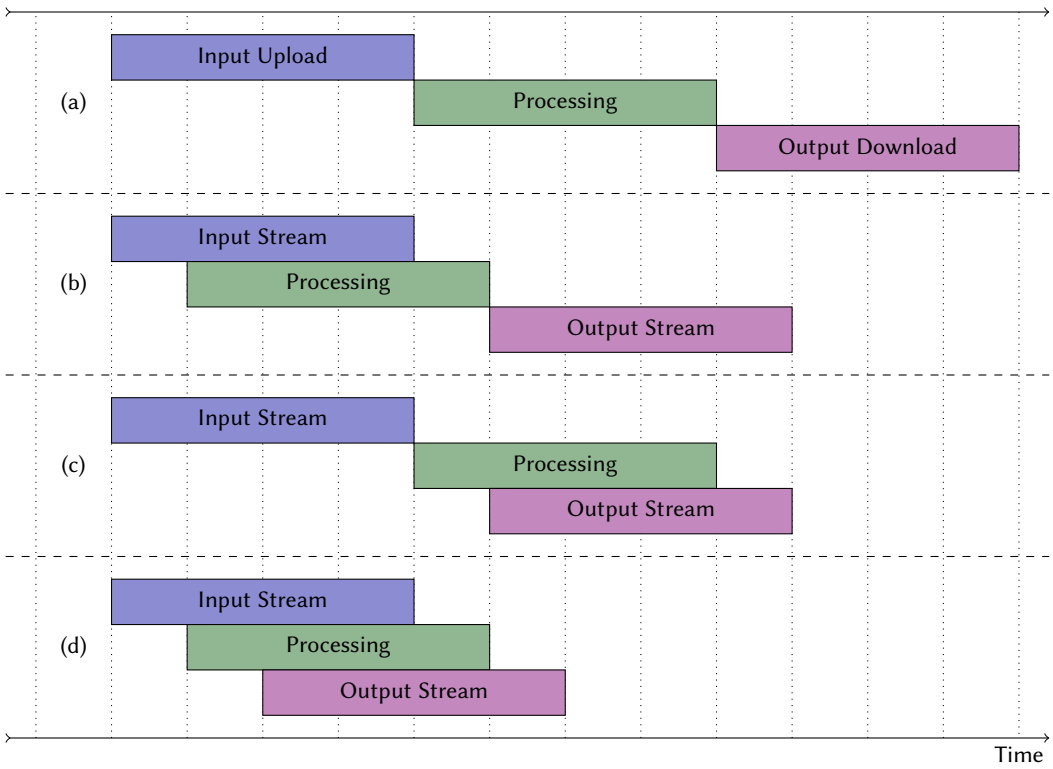


Figure 4.1: Four different types of processing data: (a) conventional processing, (b) streaming input data (c) streaming output data, (d) full input and output streaming (based on Foerster et al., 2012).

While there are efforts to utilize popular techniques like grid and cloud computing, there are few efforts in research and development to facilitate streaming processing (Foerster

et al., 2012). Previous approaches to combine the concept of streaming and web-based processing of spatiotemporal data using the WPS are drafted in strong correlation to media streaming (ibid.) by using playlist files (Pantos and May, 2013) as inputs and outputs of a WPS process. The process is executed asynchronously and the output playlist location is published using the `<wps:ProcessStarted>` element of the process status response (see Figure 4.2). As the WPS specification is not designed to be extensible, the element's content is restricted to a simple string and can not contain complex Extensible Markup Language (XML) structures. Furthermore, the element's definition states that it should be used to convey a human readable text that is presented to an user:

> *"A human-readable text string whose contents are left open to definition by each WPS server, but is expected to include any messages the server may wish to let the clients know. Such information could include how much longer the process may take to execute, or any warning conditions that may have been encountered to date. The client may display this text to a human user."*

Despite the goal of maintaining compatibility to WPS specification and existing software components, this represents a misappropriation of the element and will result in incompatibilities with existing WPS client solutions. Besides that, this solution is only able to transport a single playlist location to the client and thus, a WPS process may only have a single streaming output.

Input parameters may also be supplied using a playlist file. The coordination of several streaming inputs is either not possible or heavily depending on the streaming enabled process. A process accepting two ore more streamed data sets has to decide which data chunks it has to combine. Even the simplest case of combining chunks with the same index of both streams can have serious implications in the use case of live analysis. If a data chunk gets lost, either due to hardware or network failure, the process will combine chunks that are not related. In continuous processes this error can not be detected because two indefinite streams of data will always have matching indices. Use cases, in which the rate of incoming data differs between streams or in which data chunks depend on other chunks, are very hard to model and will result in highly specialized processes. These models depend not only on the structure and format of input data, but also on the data source, and thus the incoming rate of the data. By this, generic solutions, that convert existing WPS processes into streaming enabled processes, are hard to develop, and most streaming enabled processes may not be used in contexts apart from the one that it was

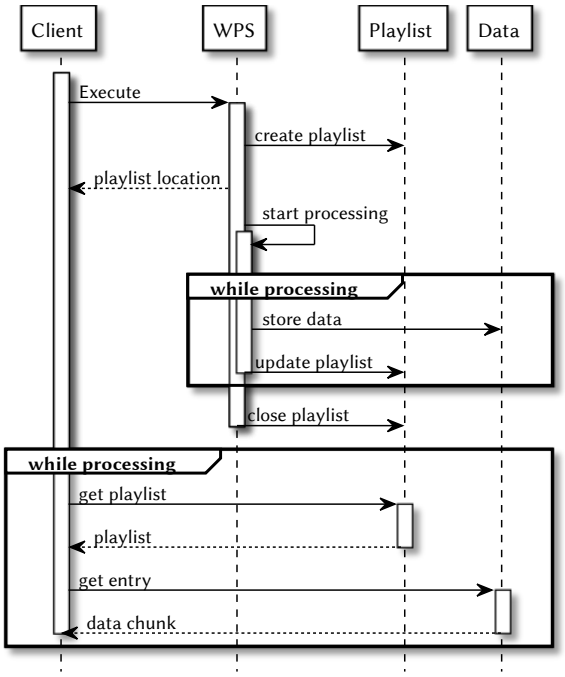


Figure 4.2: Sequence diagram of the playlist-based streaming enabled WPS (Foerster et al., 2012).

developed for.

Moreover, realizing streaming by continuous polling of playlists is highly inefficient. Neither can the client know the rate output data is produced nor can the WPS process know at which rate input data becomes available. By polling at a too slow rate the arrival of data chunks may be missed, which results in a slower process execution, and by polling at a too high rate, network and computation resources are wasted. Adaptive polling rates may be a solution for this problem, but are useless in cases, where the rate of incoming data changes across the process execution. In contrast to transporting data from the server to the client, for which the playlist concept was originally developed in the context of media streaming, the usage of playlists to transport data from the client to the server is additionally questionable. Clients need the capability to publish files as resources, which are accessible using an URL (e.g. on a FTP or HTTP server). In a web browser environment, a JavaScript client is only able to do this using an external service that stores the data and maintains the playlist. A pure JavaScript browser client is not able to use streaming inputs in this playlist-based streaming WPS approach. The implementation of this approach

is additionally limited. Input parameter data streams are not implemented and process implementations have to split inputs to create output streams (see Figure 4.1c). Splitting spatiotemporal data into smaller chunks is not as trivial as e.g. splitting an audio or video stream into single frames. By this, the process implementations become heavily format dependent and dependencies between data chunks can only be expressed as part of the data, and in a format, that the process is able to understand and to handle. Also this approach requires a reimplementation of already existing processes to achieve streaming outputs.

A streaming enabled WPS should extend the traditional processing paradigm (see Figure 4.1a) to enable input only streaming (Figure 4.1b), output only streaming (Figure 4.1c), and full input/output streaming (Figure 4.1d). For this, it should be possible to supply input parameters subsequently and to publish output data chunks as they become available. To accomplish this, a streaming enabled WPS should not rely on inefficient polling techniques, in which the server or client is requesting a resource continuously over time, but should rely on true streaming technologies that offer a full-duplex communication channel between client and server. Streaming enabled processes should be accessible from the same environments as conventional WPS processes. This especially includes web browser environments that are particularly restricted in their possibilities. A streaming enabled WPS process should rely on existing, widely known and standardized technologies, it should be especially as interoperable as possible to the WPS specification, but should not compromise streaming functionality by enforcing incompatible standards. As spatiotemporal data and its processing and analysis often can not be treated independently from surrounding data, dependencies between streamed data chunks have to be considered. This will require the streaming enabled process to be able not only to operate on sequential data but also be able to allow, to some degree, random access to the data. Despite handling of dependencies between spatiotemporal features should be considered, processes and algorithms that require global knowledge of the data set, may not profit from streaming and should not be considered relevant for a streaming enabled WPS. The system should be as generic as the existing WPS specification, so it should not rely on specific data formats and allow easy chaining of streaming processes. As possible use cases include not only live analysis of data, but also the processing of large data set, data chunks should be processed in parallel if possible. As this may result in an undefined order of outputted data chunks, clients need to be able to correlate output data chunks with the input parameter chunks. Existing WPS processes should be easily converted to streaming enabled processes, without the need to develop them from scratch.

The following sections should introduce a approach for a Streaming WPS, that will fulfill the above requirements. As seen in previous approaches, the constraints imposed by the WPS specification are too strict to implement a standard compatible streaming enabled WPS fulfilling the requirements. Previous solutions compromised functionality for the sake of (incomplete) compatibility with the inflexible standard. In order to enable true, browser compatible streaming, the approach presented in thesis will break out of the constraining WPS standard and develop a message based architecture using WebSockets to accomplish true full-duplex streaming of data while reusing terminology and technology specified by the WPS standard.

4.1 Protocol

As the WPS specification is not flexible enough to model a full streaming scenario, the WPS needs to be bypassed. In order to accomplish this, a more flexible interaction model was developed, which extends the conventional processing approach. This protocol is message based and enables full-duplex stream processing of spatiotemporal data. A *streaming enabled algorithm* is a WPS algorithm that supports the here defined protocol while a *streaming process* is the identifiable instance of an algorithm, created by executing the streaming enabled algorithm using the WPS *Execute* operation. The streaming process is the core of the Streaming WPS and receives subsequent inputs and will emit intermediate results. The execution of the streaming enabled algorithm is fully supported by the WPS specification, whereas all interaction with the streaming process is not part of the standard. To communicate with the streaming process, the client needs information on how to connect to the process. As the WPS specification does not allow subsequent outputs, the call of the *Execute* operation will return immediately to transport this information to the client, and can not persist over the lifetime of the streaming process.

To enable a full duplex communication with the streaming process, WebSockets will be used to transport messages. They are needed to *push* messages to clients instead of letting the clients constantly request updates.

The detailed interaction protocol is depicted in Figure 4.3. A client (*Sender*) issues an *Execute* to a streaming enabled WPS algorithm (step 1). The algorithm instantiates a delegate (step 2) that is responsible for processing data chunks, and a streaming process (step 3) that

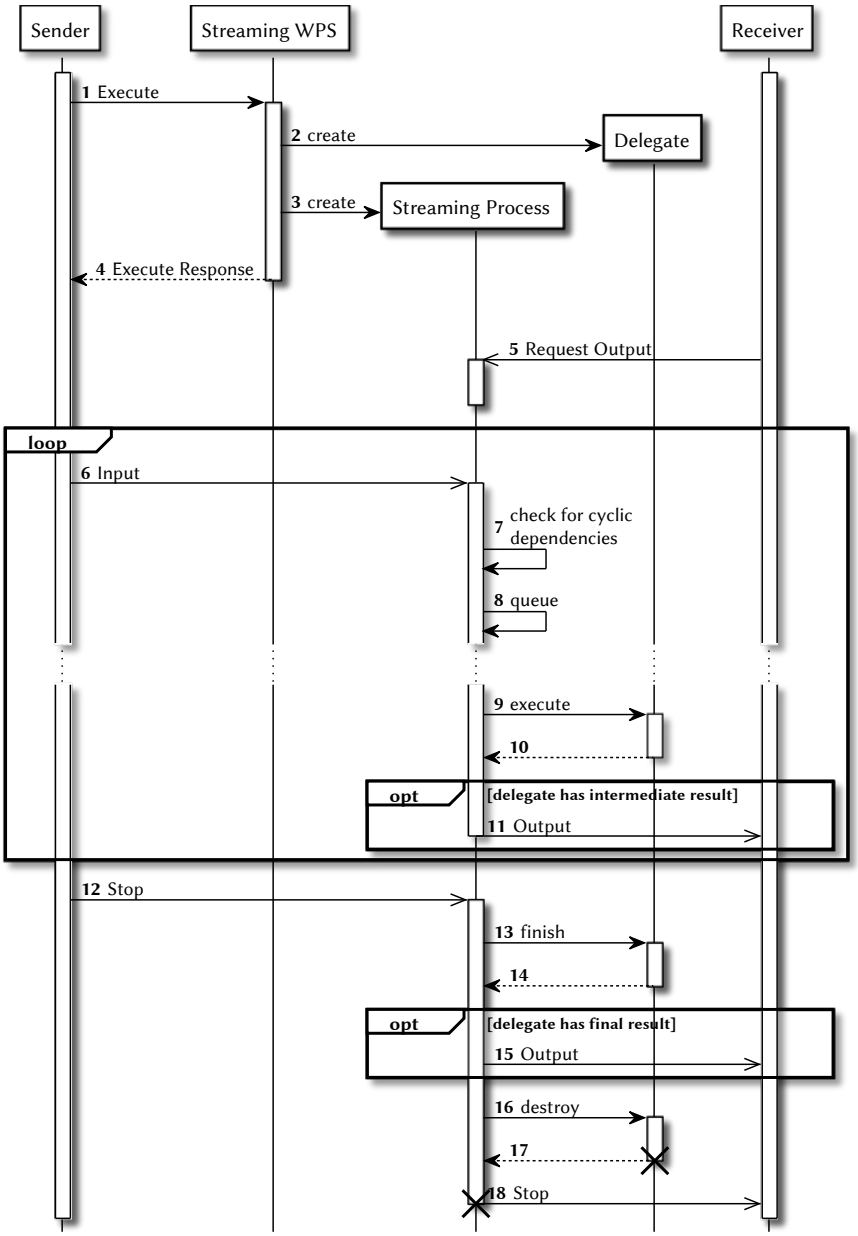


Figure 4.3: Sequence diagram of typical interaction pattern with a streaming enabled WPS algorithm using two distinct clients for sending and receiving data.

is responsible for client interactions and task scheduling. The Execute response will contain the necessary details to connect to the streaming processes, such as the the identifier of the streaming process and the WebSocket endpoint URL (step 4).

With these details a client can connect directly to the streaming process bypassing the WPS interface. In step 5 another client[1] (*Receiver*) connects to the streaming process and subscribes to the future outputs of the process. By this, the client does not need to constantly issue requests to the streaming process to check for new outputs, but receives outputs automatically as long as the receiving client stays connected using the WebSocket. After this, one or multiple clients start sending chunks of data as input parameters to the streaming process (step 6). The clients may open a new connection for every input or use the same connection over the lifetime of the streaming process. The streaming process checks the inputs for validity (step 7) and queues them for processing (step 8). Processing takes places asynchronously in parallel manner and there is no guarantee of order (besides restrictions imposed by dependencies, see Sections 4.3.3 and 4.4). When there are free capacities to process the data and all other requirements are met, the delegate is tasked to process the data (step 9). The delegate implementation can return an intermediate result in step 10, which is forwarded to all registered receivers in step 11. Steps 6 to 11 may be repeated indefinitely (e.g. live analysis of data) or until the sending client has no more inputs to feed. As the streaming process would wait in this case forever (or at least until some timeout interferes), the client has to stop the streaming process explicitly (step 12). This causes the streaming process to stop accepting inputs, to process all not yet processed inputs, and to request a last potential output from the delegate (steps 13 and 14), which is forwarded to all listening clients (step 15). After this, it destructs the delegate (steps 16 and 17) and notifies all registered listeners, that no further outputs will become available by forwarding the stop message (step 18) to the clients. The streaming process will destroy itself after this. A detailed description of the various messages of this protocol can be found in Section 4.2.

The protocol permits various streaming usage scenarios. A delegate that produces an output for every input message creates a full input/output streaming process (see Figure 4.1d). A delegate that produces only a final output results in an input only streaming process (see Figure 4.1b). By suppling a single input message and repeating step 11, a suitable delegate may create an output streaming process (see Figure 4.1c) and, although not reasonable, even the traditional processing approach depicted in Figure 4.1a can be simulated by passing all inputs in a single input message and producing a single output message.

Using message provoked streaming iterations (the combination of an input message, its processing and (optional) output message) allows the use of multiple streaming inputs

1. Even though sender and receiver are two different entities in this diagram, there are no restrictions imposed to the amount of clients, either senders or receivers, or their nature (senders may also be receivers).

and outputs. In contrast to previous approaches it is possible for the streaming process to relate these to a single processing iteration without any knowledge of their semantics because the client encapsulates them in a single message.

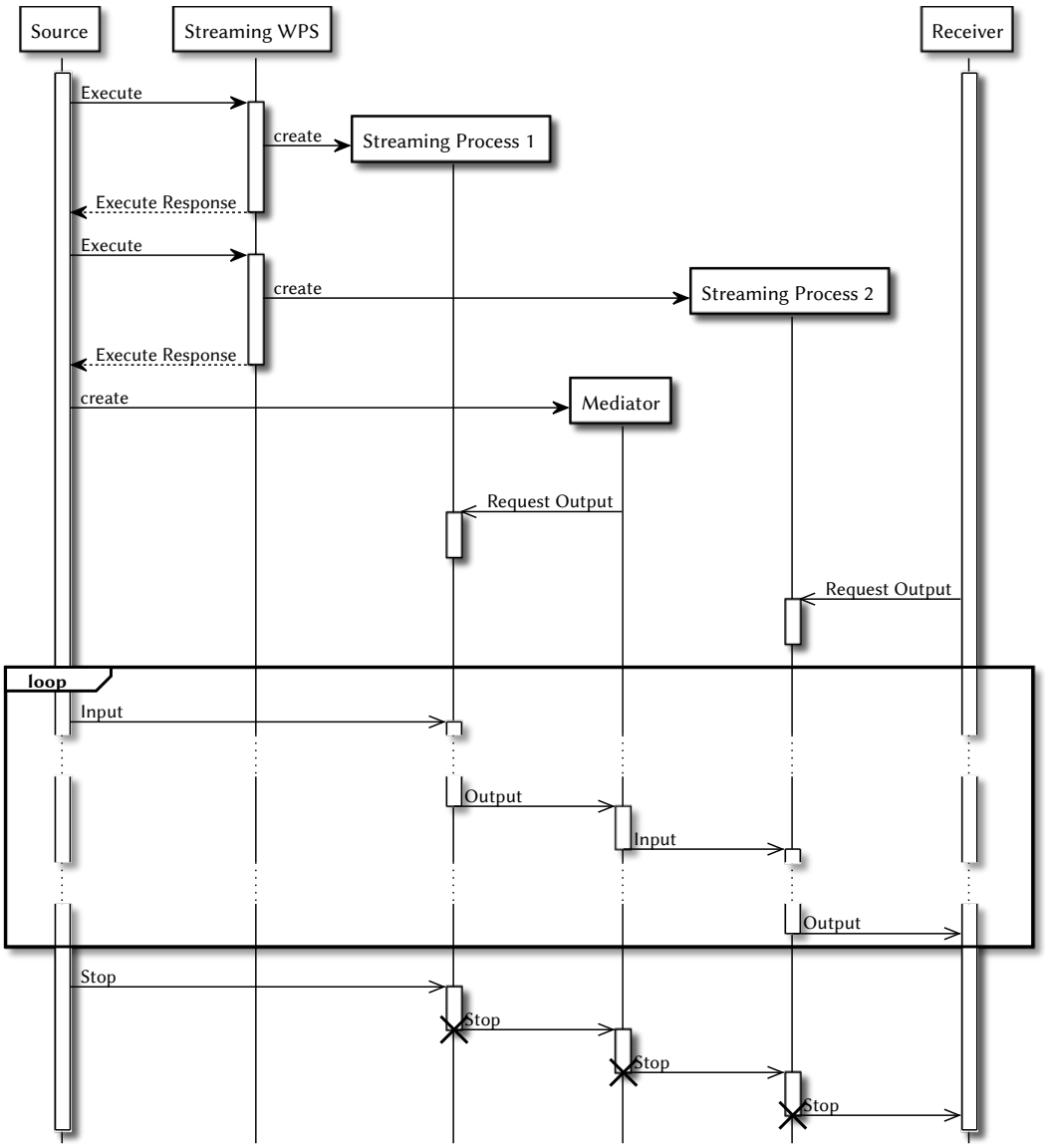


Figure 4.4: Sequence diagram of chaining two streaming processes using a generic mediator between the processes to translate output to input messages.

The protocol also enables the chaining of processing steps. This can be realized in two ways: on the one hand, a delegate itself may represent a WPS process chain and thus chain

every processing step, or, on the other hand, several streaming processes are chained. A simple mediator is translating input messages to output messages (see Figure 4.4). This mediator can be realized using a dedicated streaming enabled algorithm accepting an input/output mapping and the connection parameters of the streaming processes to connect. After requesting the outputs of the source streaming process, it can translate every output message to an input message and forward the stop message. A receiving client will connect to the second streaming process and receives the data process by the chain. By requesting the outputs of the first streaming process, even intermediate results of the chain are accessible.

4.2 Messages

To fulfill the above defined protocol, several messages have to be exchanged between sender, streaming process and receiver. In order to correlate input and outputs or to show the source of an error, the message format has to have a concept of message references. WebSockets do not have such a concept as it is only a thin layer on top of TCP, which introduces only a handshake and addressing mechanism to be compatible with HTTP and a minimal framing of messages. This framing is merely needed to establish a message-based instead of a stream-based protocol, as the latter would make it hard to differentiate between individual messages (Fette and Melnikov, 2011). To enable referencing of messages, and by this an asynchronous reply mechanism, another layer is needed. As the WPS is mostly based on XML, the message format should also be XML based. This enables the usage of large parts of the WPS schema and allows the reuse of many components written to interact with the WPS.

The widely known SOAP protocol (Lafon et al., 2007) – which may also be used as an optional binding of the WPS (Open Geospatial Consortium, 2007d) and thus can be easily adopted – is an ideal candidate for this. In combination with Web Services Addressing (WSA) (Rogers et al., 2006) it creates an XML based message framework, which allows asynchronous requests and responses over an arbitrary protocol. Besides introducing a concept of addressing and routing of messages (that will not be used in the Streaming WPS), one can assign a globally unique identifier to any message using WSA, which can be referenced with arbitrary semantics (e.g. reply).

The Streaming WPS defines seven SOAP messages. These are outlined in the following sections.

4.2.1 Input Message

Input messages are used by clients to supply subsequent inputs to a streaming iteration of a streaming process. They loosely resemble a WPS Execute request by consisting of any number of inputs and an identifier, that references the streaming process to which the inputs should be supplied. An example can be seen in Listing 4.1; possible inputs are discussed in depth in Section 4.3.

Listing 4.1: Example for a Streaming WPS input message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:RelatesTo RelationshipType="https://github.com/autermann/
        streaming-wps/needs">uuid:f31da315-bce3-4e26-8112-3ccf0ecf1ab5</
        wsa:RelatesTo>
    <wsa:MessageID>uuid:6a0e50c7-85c4-448c-962d-894c41c441bf</
        wsa:MessageID>
    <wsa:Action>https://github.com/autermann/streaming-wps/input</
        wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:InputMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
          stream:ProcessID>
      <stream:Inputs>
        [...]
      </stream:Inputs>
    </stream:InputMessage>
  </soap:Body>
</soap:Envelope>
```

4.2.2 Output Messages

Output messages are used by the streaming process to transport intermediate results at the end of a streaming iteration or a final result at the end of the streaming process to listening clients. They loosely resemble a WPS Execute response by containing an arbitrary number of outputs and the identifier of the process that produced the outputs. Output messages

containing intermediate results are replies to their corresponding input message and reference them using WSA. If the processing used the output of any other streaming iteration (see Sections 4.3.3 and 4.4), the corresponding output messages are also referenced. An example can be seen in Listing 4.2.

Listing 4.2: Example for a Streaming WPS output message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageID>uuid:ef9676f0-13b1-473b-a783-8fed8cbd6513</
        wsa:MessageID>
    <wsa:RelatesTo>uuid:6a0e50c7-85c4-448c-962d-894c41c441bf</
        wsa:RelatesTo>
    <wsa:RelatesTo RelationshipType="https://github.com/autermann/
        streaming-wps/used">uuid:cf19d698-f288-477b-a4ff-39611b46920e</
        wsa:RelatesTo>
    <wsa:Action>https://github.com/autermann/streaming-wps/output</
        wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:OutputMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
          stream:ProcessID>
      <stream:Outputs>
        <stream:Output>
          <ows:Identifier>output1</ows:Identifier>
          <wps:Data>
            <wps:LiteralData dataType="xs:string">input1</wps:LiteralData
                >
          </wps:Data>
        </stream:Output>
        <stream:Output>
          <ows:Identifier>output2</ows:Identifier>
          <wps:Data>
            <wps:ComplexData mimeType="application/xml" encoding="UTF-8">
              <hello>world</hello>
            </wps:ComplexData>
          </wps:Data>
        </stream:Output>
        <stream:Output>
          <ows:Identifier>output3</ows:Identifier>
          <wps:Data>
            <wps:BoundingBoxData crs="EPSG:4326" dimensions="2">
              <ows:LowerCorner>52.2 7.0</ows:LowerCorner>
              <ows:UpperCorner>55.2 15.0</ows:UpperCorner>
            </wps:BoundingBoxData>
          </wps:Data>
        </stream:Output>
      </stream:Outputs>
    </stream:OutputMessage>
  </soap:Body>
</soap:Envelope>
```

4.2.3 Output Request Message

An output request message is used by a client to subscribe to the outputs generated by a streaming process. There is no direct counter part in the WPS specification but the concept is similar to the continuous request of the WPS response during an asynchronous process execution. As WebSockets offer a full-duplex messaging channel, a continuous polling of outputs is not needed, but the streaming process can push outputs directly to listening clients. To initialize this listening, the client registers to one or more streaming processes using their corresponding identifiers. An example can be seen in Listing 4.3.

Listing 4.3: Example for a Streaming WPS output request message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageID>uuid:950a3380-1de4-4634-ba2d-ffdf324157d7</
        wsa:MessageID>
    <wsa:Action>https://github.com/autermann/streaming-wps/request-output
        </wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:OutputRequestMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
          stream:ProcessID>
    </stream:OutputRequestMessage>
  </soap:Body>
</soap:Envelope>
```

4.2.4 Stop Message

As streaming processes can run indefinitely long, input supplying clients need to be able to inform the streaming process that there will be no further inputs that become available. To achieve this, a stop message (see Listing 4.4) is sent to the streaming process. The process will propagate the stop message to all listening clients to notify them that there will be no further outputs. Before the stop message is propagated, all streaming iterations that are not yet processed will be finished, but the process will not accept any further inputs. If there are still unresolved dependencies (see Sections 4.3.3 and 4.4), the streaming process will fail with an error message.

Listing 4.4: Example for a Streaming WPS stop message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageID>uuid:01ea8dab-5da9-46eb-81b4-06dcea32ca01</
        wsa:MessageID>
    <wsa:Action>https://github.com/autermann/streaming-wps/stop</
        wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:StopMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
          stream:ProcessID>
    </stream:StopMessage>
  </soap:Body>
</soap:Envelope>
```

4.2.5 Error Message

Errors are transported, as in the WPS specification, using OWS exception reports (Open Geospatial Consortium, 2007c,d). If the delegate of a process fails or a supplied input message can not be processed due to whatever conditions, the error is propagated to listening clients. The error is always sent to the client that sent the message causing the error (if the client is still connected), and in case the error is caused during the execution of a streaming iteration, also to all listening clients that registered through an output request message. In contrast to failures during input validation, e.g. due to constraints imposed by dependencies (see Sections 4.3.3 and 4.4), errors raised during the execution of a streaming iteration can not be compensated, but will stop the streaming process. The causing message of a failure may be obtained from the reply relation encoded using WSA. An example of an error message can be found in Listing 4.5.

Listing 4.5: Example for a Streaming WPS error message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
    <soap:Header>
        <wsa:RelatesTo>uuid:6a0e50c7-85c4-448c-962d-894c41c441bf</
            wsa:RelatesTo>
        <wsa:MessageID>uuid:dc640a0a-d505-4591-baea-2a556412237e</
            wsa:MessageID>
        <wsa:Action>https://github.com/autermann/streaming-wps/error</
            wsa:Action>
```

```
    </soap:Header>
    <soap:Body>
        <stream:ErrorMessage>
            <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
                stream:ProcessID>
            <ows:Exception exceptionCode="RemoteComputationError">
                <ows:ExceptionText>Remote computation failed</
                    ows:ExceptionText>
            </ows:Exception>
        </stream:ErrorMessage>
    </soap:Body>
</soap:Envelope>
```

4.2.6 Describe & Description Message

Describe messages are directly adopted from the WPS *DescribeProcess* operation. Due to conditions described in Section 4.5, a client needs to be able to retrieve a description from a running streaming process. The message simply contains the identifier of the process the client wants to have the description from. An example for this process can be seen in Listing 4.6.

Listing 4.6: Example for a Streaming WPS describe message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageID>uuid:9ca0ed4a-0e24-4843-bb81-da2af3e23d8c</
        wsa:MessageID>
    <wsa:Action>https://github.com/autermann/streaming-wps/describe</
        wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:DescribeMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
          stream:ProcessID>
    </stream:DescribeMessage>
  </soap:Body>
</soap:Envelope>
```

The reply resembles a *DescribeProcess* response and is encoded in a description message referencing the describe message and containing the streaming process description (see Listing 4.7).

Listing 4.7: Example for a Streaming WPS description message (see Appendix E for omitted XML namespaces).

```
<soap:Envelope>
  <soap:Header>
    <wsa:RelatesTo>uuid:9ca0ed4a-0e24-4843-bb81-da2af3e23d8c</
        wsa:RelatesTo>
    <wsa:MessageID>uuid:5ba3d87b-85d0-47eb-9dac-57cf193abd06</
        wsa:MessageID>
    <wsa:Action>https://github.com/autermann/streaming-wps/description</
        wsa:Action>
  </soap:Header>
  <soap:Body>
    <stream:DescriptionMessage>
      <stream:ProcessID>uuid:f7683417-ab11-4317-a833-d73aa443443d</
          stream:ProcessID>
      <stream:StreamingProcessDescription wps:processVersion="1.0.0"
          finalResult="false" intermediateResults="false" storeSupported=
          "true">
        <ows:Identifier>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</
            ows:Identifier>
        <ows:Title>matlab.add</ows:Title>
        <DataInputs>
          <Input maxOccurs="1" minOccurs="1">
            <ows:Identifier>a</ows:Identifier>
            <ows:Title>a</ows:Title>
            <LiteralData>
              <ows:DataType ows:reference="xs:long"/>
              <ows:AnyValue/>
            </LiteralData>
          </Input>
          <Input maxOccurs="1" minOccurs="1">
            <ows:Identifier>b</ows:Identifier>
            <ows:Title>b</ows:Title>
            <LiteralData>
              <ows:DataType ows:reference="xs:long"/>
              <ows:AnyValue/>
            </LiteralData>
          </Input>
        </DataInputs>
        <ProcessOutputs>
          <Output>
            <ows:Identifier>result</ows:Identifier>
            <ows:Title>result</ows:Title>
            <LiteralOutput>
              <ows:DataType ows:reference="xs:long"/>
            </LiteralOutput>
          </Output>
        </ProcessOutputs>
      </stream:StreamingProcessDescription>
    </stream:DescriptionMessage>
  </soap:Body>
</soap:Envelope>
```

4.3 Input Types

The aforementioned requirements imply three different types of input for a Streaming Process. They differ in the aspect of time (*When are they supplied?*) and scope (*Where are they used?*). Besides that, all of them are based on the very same input types the WPS standard defines (see Chapter 2).

4.3.1 Streaming Inputs

The first and most obvious type of input are streaming inputs. They are provided for a single streaming iteration and will only be used in that iteration representing the core of streaming enabled processing (see Listing 4.8).

Listing 4.8: Example for a Streaming WPS streaming inputs (see Appendix E for omitted XML namespaces).

```
<stream:Inputs>
  <stream:StreamingInput>
    <ows:Identifier>intput1</ows:Identifier>
    <wps:Data>
      <wps:LiteralData dataType="xs:string">input1</wps:LiteralData>
    </wps:Data>
  </stream:StreamingInput>
  <stream:StreamingInput>
    <ows:Identifier>intput2</ows:Identifier>
    <wps:Data>
      <wps:ComplexData mimeType="application/xml" encoding="UTF-8">
        <hello>world</hello>
      </wps:ComplexData>
    </wps:Data>
  </stream:StreamingInput>
  <stream:StreamingInput>
    <ows:Identifier>intput3</ows:Identifier>
    <wps:Data>
      <wps:BoundingBoxData>
        <wps:BoundingBoxData crs="EPSG:4326" dimensions="2">
          <ows:LowerCorner>52.2 7.0</ows:LowerCorner>
          <ows:UpperCorner>55.2 15.0</ows:UpperCorner>
        </wps:BoundingBoxData>
      </wps:BoundingBoxData>
    </wps:Data>
  </stream:StreamingInput>
  <stream:StreamingInput>
    <ows:Identifier>input4</ows:Identifier>
    <wps:Reference mimeType="application/xml" encoding="UTF-8" schema="
        http://schemas.opengis.net/gml/3.1.1/base/gml.xsd" xlink:href="
        http://geoprocessing.demo.52north.org:8080/geoserver/wfs?service=
        WFS&amp;version=1.0.0&amp;request=GetFeature&amp;typeName=
        topp:tasmania_roads&amp;srs=EPSG:4326&amp;outputFormat=GML3"/>
```

```
  </stream:StreamingInput>
</stream:Inputs>
```

A conventional algorithm to compute the histogram of a raster (e.g. a satellite image) needs the complete raster as a single complex input for processing. A streaming enabled variant would split the raster in several smaller tiles and supply each of them in a single input message to the streaming process. The algorithm can process each tile on its own and update the global histogram. Besides that the process does not have to store the complete raster, it is also able to output intermediate histograms to the client.

4.3.2 Static Inputs

Algorithms operating on a streaming input often need inputs that are common to every iteration. It would be redundant and inefficient to transfer inputs like configuration parameters in every input message for every streaming iteration. For this, the concept of static inputs needs to be introduced. Static inputs are parameters that are supplied when a streaming process is created and apply to every streaming iteration (see Listing 4.9). While the streaming process handles a streaming iteration, the static inputs are merged with the inputs of the causing input message and transparently supplied to the process's delegate. This way, a conventional process can be easily converted into a streaming enabled process. Additionally, static inputs can be handled separately to configure a streaming process's delegate when it is created.

Listing 4.9: Example for a Streaming WPS static inputs (see Appendix E for omitted XML namespaces).

```
<stream:StaticInputs>
  <wps:Input>
    <ows:Identifier>input1</ows:Identifier>
    <wps:Data>
      <wps:LiteralData dataType="xs:string">input1</wps:LiteralData>
    </wps:Data>
  </wps:Input>
  <wps:Input>
    <ows:Identifier>input2</ows:Identifier>
    <wps:Data>
      <wps:ComplexData mimeType="application/xml" encoding="UTF-8">
        <hello>world</hello>
      </wps:ComplexData>
    </wps:Data>
  </wps:Input>
  <wps:Input>
```

```
    <ows:Identifier>input3</ows:Identifier>
    <wps:Data>
      <wps:BoundingBoxData>
        <wps:BoundingBoxData crs="EPSG:4326" dimensions="2">
          <ows:LowerCorner>52.2 7.0</ows:LowerCorner>
          <ows:UpperCorner>55.2 15.0</ows:UpperCorner>
        </wps:BoundingBoxData>
      </wps:BoundingBoxData>
    </wps:Data>
  </wps:Input>
  <wps:Input>
    <ows:Identifier>input4</ows:Identifier>
    <wps:Reference mimeType="application/xml" encoding="UTF-8" schema="
        http://schemas.opengis.net/gml/3.1.1/base/gml.xsd" xlink:href="
        http://geoprocessing.demo.52north.org:8080/geoserver/wfs?service=
        WFS&amp;version=1.0.0&amp;request=GetFeature&amp;typeName=
        topp:tasmania_roads&amp;srs=EPSG:4326&amp;outputFormat=GML3"/>
  </wps:Input>
</stream:StaticInputs>
```

For example, a traditional process implementation of the Douglas-Peucker algorithm (Douglas and Peucker, 1973) would require a feature collection and a ϵ value as inputs. In a streaming environment, one would model the ϵ input as a static input supplied at process creation and stream the feature collection as single features in streaming inputs. Other examples are a coordinate transformation process that accepts a feature collection and a target CRS, or a buffer algorithm that accepts a feature collection and a buffer size. Buffer size and CRS would be supplied as static inputs, whereas the feature collection would be split into several streaming inputs and would be supplied in independent streaming iterations.

4.3.3 Reference Inputs

While streaming offers no real benefit to algorithms that require global knowledge of the data set, there are often cases where algorithms only require knowledge about few other chunks of the data set or even only about the result of their processing. To model these dependencies between streaming iterations, reference inputs can be used (see Listing 4.10). These reference the output of another – previous or upcoming – iteration as an input parameter. Reference inputs break out of the conventional non-random access paradigm of streaming and allow a semi-random access processing of a data set. Inputs are described by referencing the corresponding output identifier and the input message that has or will produce the output data. The order of incoming input messages is irrelevant to the use of

reference inputs, as input messages referencing not yet available outputs will be delayed until they can be processed (see Section 4.4).

Listing 4.10: Example for a Streaming WPS reference input (see Appendix E for omitted XML namespaces).

```
<stream:Inputs>
  <stream:ReferenceInput>
    <ows:Identifier>input3</ows:Identifier>
    <stream:Reference>
      <wsa:MessageID>uuid:f31da315-bce3-4e26-8112-3ccf0ecf1ab5</
          wsa:MessageID>
      <stream:Output>output1</stream:Output>
    </stream:Reference>
  </stream:ReferenceInput>
</stream:Inputs>
```

A conventional algorithm to analyze a river system, in which each processing of a river depends on the processing results of the rivers flowing into it, the complete river system data set would be supplied as a single input parameter. In a streaming enabled process, each river would be supplied as a streaming input. The output of the rivers that a river depends on would be supplied as additional reference inputs.

4.3.4 Polling inputs

The last category of possible input types for a streaming WPS are polling inputs. These inputs are continuously polled from an external resource and a new streaming iteration would be started, when new inputs become available. Polling inputs would be supplied at process creation time and would contain a reference to an external resource, which is requested continuously. In order to not miss inputs when they become available, a playlist file as described in previous approaches (Foerster et al., 2012) would be needed. The implementation of polling inputs as part of this streaming WPS specification would present the very same issues that were criticized in previous approaches:

- How can polling frequencies used to retrieve the playlist be defined?
- How can multiple polling inputs be declared?
- How would they be combined by the streaming WPS?

For this reason, the Streaming WPS will not implement polling inputs. These input types are by far better handled on client side, as the client typically knows the rate data becomes

available. By this the client can choose an appropriate polling frequency and also is able to coordinate multiple polling inputs by having a deeper understanding of their affiliation. Polling inputs could be implemented as shown in Figure 4.5: the client polls a data provider (e.g. a Sensor Observation Service (SOS, Open Geospatial Consortium, 2012a)) to check if new data is available and convert this data into a streaming input for the Streaming WPS.

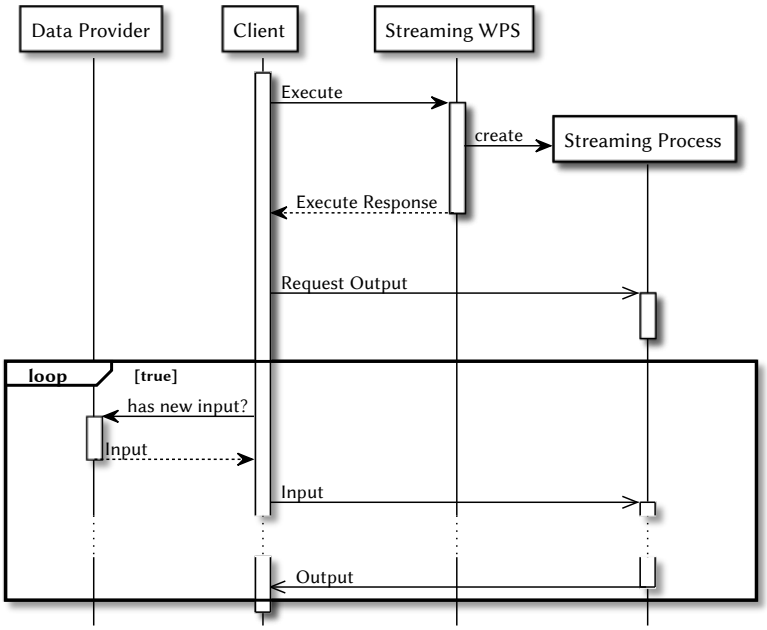


Figure 4.5: Sequence diagram of how to implement polling inputs for a streaming enabled WPS algorithm.

4.4 Dependencies

The definition of reference inputs in Section 4.3.3 implies a mechanism to resolve dependencies and to order the execution of streaming iterations. These are considered as tasks and can declare dependencies to other streaming iterations either by mapping an input to the output of another streaming iteration or by declaring an explicit dependency on another streaming iteration.

Dependencies can be best modeled using a Directed Acyclic Graph (DAG). A DAG is a structure $D = (V, E)$ consisting of a set of vertices (or nodes) V and edges (or arcs) E where every edge $e \in E$ is a ordered tuple $v_1 \rightarrow v_2$ with $v_1, v_2 \in V$. The distinct

vertices $v_1, \ldots, v_n \in V$ are called a path if for all successive vertices v_i, v_{i+1} exists an edge $v_i \to v_{i+1} \in E$. A directed graph is called acyclic if there exists no path in G with $v_1 = v_n$ (Jungnickel, 2012). A subgraph of a graph is the graph $G' = (V', E')$ with $V' \subseteq V$ and $E' = \{v_1 \to v_2 \in E | v_1, v_2 \in V'\}$. Two subgraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ are independent if $V_1 \cap V_2 = \emptyset$ and there exists no edge $v_1 \to v_2 \in E$ with $v_1 \in V_1 \wedge v_2 \in V_2$ or $v_2 \in V_1 \wedge v_1 \in V_2$.

In a dependency graph, vertices represent a task, package or other entity that has dependencies, and edges represent these dependencies (v_1 depends on v_2). Dependency graphs have to be acyclic as a cycle would introduce a cyclic dependency that can not be resolved.

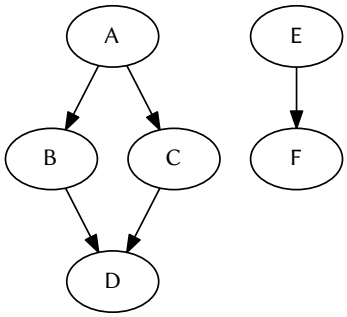


Figure 4.6: Example for a dependency graph consisting of two independent subgraphs. Arrows denoting a dependency between the nodes.

A system containing the tasks A, B, C, D, E, F and the dependencies $A \to B, A \to C, B \to D, C \to D$ and $E \to F$ will result in a DAG consisting of two independent subgraphs (see Figure 4.6).

The execution order of a dependency graph can be derived from the topological ordering of the graph: a "topological ordering, ord_D, of a directed acyclic graph $D = (V, E)$ maps each vertex to a priority value such that $ord_D(x) < ord_D(y)$ holds for all edges $x \to y \in E$" (Pearce and Kelly, 2007). A possible execution order is the list of all vertices sorted by descending ord_D. The topological order of a DAG can be computed using e.g. Breadth-first search (BFS) in linear time (Cormen et al., 2001). In most cases, the topological ordering is not unique; Figure 4.7 shows one possible execution order for the before mentioned

graph.

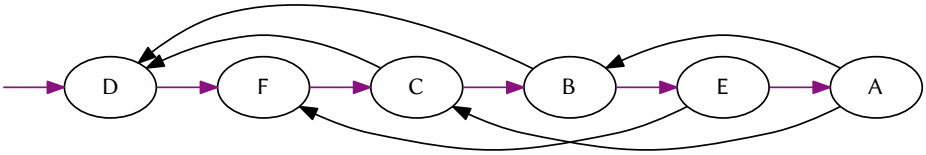


Figure 4.7: Possible execution/topological order of the dependency graph in Figure 4.6. Black arrows represent dependence to another vertex, colored arrows the execution order.

In contrast to conventional dependency systems like package managers, the Streaming WPS can not operate on a static graph of dependencies but on a graph to which vertices and edges are added constantly. Conventional topological sorting algorithms have to recompute the ordering for every insertion from scratch, which will have a big performance impact for the scenario of a great number of small streaming iterations. There exist a few dynamic topological sort algorithms that will maintain the topological order across edge and node insertions, and will only recompute the ordering if necessary.

Most dependency graphs generated using the Streaming WPS will probably consist of multiple independent subgraphs, no dependencies at all would be the most extreme example, or quite sparse graphs. For this, the algorithm described by Pearce and Kelly (2007) seems to be appropriate. Even though it is theoretically inferior to other algorithms for dynamic topological sorting (e.g. Alpern et al., 1990; Marchetti-Spaccamela et al., 1996), in practice it performs better especially on sparse graphs and on dense graphs only a constant factor slower than other algorithms (Pearce and Kelly, 2007).

Dependencies are of particular importance in case of execution failures. If the computation of a streaming iteration fails for whatever reason, all iterations that directly or indirectly depend on this iteration can not complete. As this also holds true for iterations that are supplied at a later time in the streaming process, the process can not proceed ignoring the error. Due to this, every error that occurs during the execution of a streaming iteration results in the termination of the streaming process. Dependencies also have a special meaning at the end of a streaming process, when a stop message is sent to notify the streaming process to accept no further inputs and finish pending streaming iterations. At

this point, all dependencies need to be able to be satisfied, which implies that all referenced input messages have been sent to the streaming process. In case a referenced input message is missing, the service is not able to complete gracefully and will fail. As references to future streaming iterations are allowed, prior to this point it is not possible for the Streaming WPS to determine if a reference may not be fulfilled. Because the service is not able to fail fast for incorrect references, clients using dependencies between streaming iterations have to pay careful attention to references.

It should also be noted, that the smallest unit that can be referenced in a streaming process is the output of a streaming iteration. Format specific references, e.g. to a particular feature inside a feature collection, are not possible using this protocol. Because of that, streaming process implementations need to be designed to not require smaller components or have to deploy an own referencing strategy (e.g. by supplying an additional input to identify the feature of the referenced collection). But, as this results in superfluous transfer of data, such solutions should be avoided. One may point out, that there is no way to reference input parameters of other streaming iterations, but this use case should be already covered by the WPS's own input reference parameters (see Section 4.3).

4.5 Process Description

The conventional process description mechanism of the WPS is not sufficient to describe streaming processes.

It consists of a *DescribeProcess* request issued to the WPS and the retrieval of one or more process descriptions of the specified process. These descriptions contain detailed descriptions of input and output parameters of the process and information about the supported formats, units of measurement or coordinate reference systems of each parameter. They also include details about allowed values, default value and multiplicity of input parameters (Open Geospatial Consortium, 2007d).

Because the Streaming WPS uses the WPS interface only to start a Streaming Process and the WPS interface does not provide any extension points for process descriptions, the *DescribeProcess* operation can only be used to describe the starting process, but not the input or output parameters of a streaming process.

In case of generic processes, e.g. processes that delegate to other WPS processes, information about input and output parameters is not even available prior to the execution of the streaming process. Furthermore input parameter cardinalities may change due to the use of static inputs. By this a valid input parameter for a delegate process may not be used in subsequent inputs because the maximal occurrence of the parameter is already exhausted using static input parameters. By this a process description for a streaming process will always be instance specific and can not be generated by the associated WPS process.

With knowledge of the delegate process a client may have enough information to facilitate the streaming process but for other streaming process there is no way for a generic client to know the input parameters of the process.

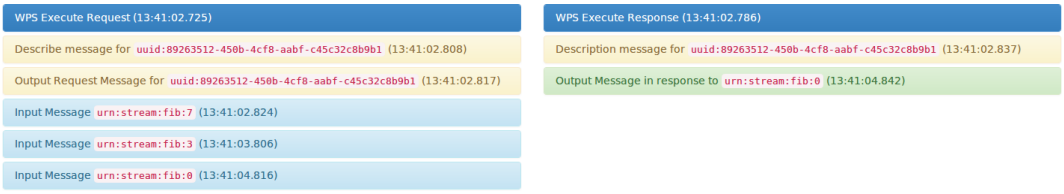
To compensate this shortcoming a method is needed to describe a Streaming Process instance at runtime.

4.6 Implementation

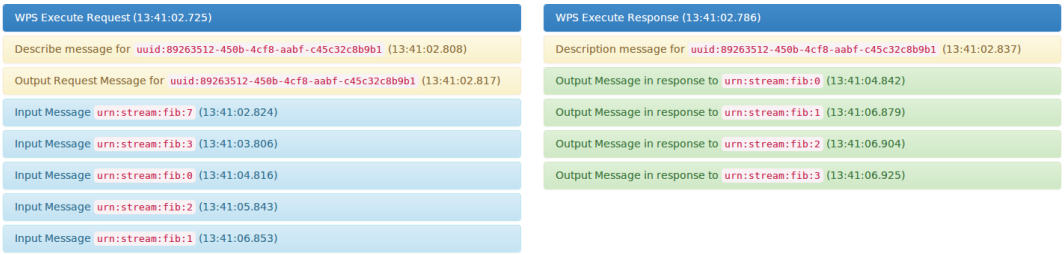
The Streaming WPS was prototypically, but feature complete implemented as a module for the 52°North WPS implementation. Besides offering a framework to develop all kinds of streaming enabled processes, it features a generic streaming process implementation that is able to convert every suitable algorithm (e.g. Douglas–Peucker) into a streaming enabled variant. The generic streaming process accepts the description of another WPS process and the URL of its endpoint as input parameters, and uses this remote process as an delegate for every streaming iteration. Additionally, it accepts a collection of static input parameters that are merged with the input parameters of a streaming iteration and are then transparently sent to the delegate process in every streaming iteration. Processes developed using the framework are able to maintain an internal state that can be mutated during execution, e.g. to create the sum of all streaming iterations, which is outputted as a final result. Due to the fact that the WPS standard defines a stateless protocol, whereas the protocol defined by the Streaming WPS encourages the use of an internal state, the possibilities of the generic streaming process are limited in this regard. Internal state can only be conveyed using the dependency mechanisms provided by the Streaming WPS, i.e. transporting state in input and output parameters.

An example for this can be visualized using the JavaScript based client library that was developed. The API abstracts message encoding and WebSocket interaction, allows to

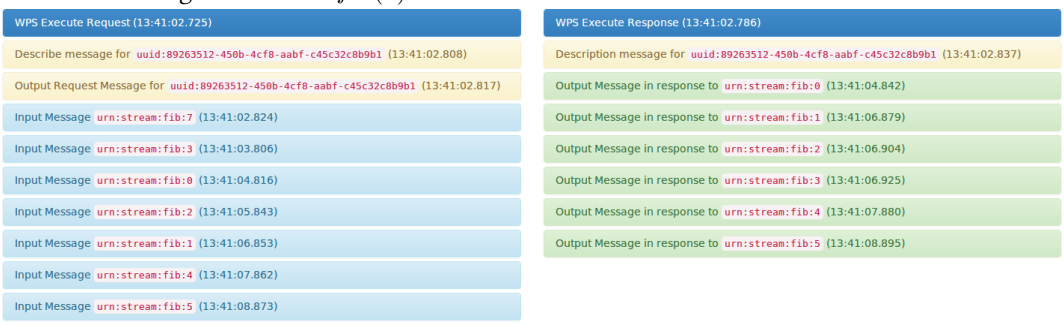
(a) Starting of the streaming process using a WPS *Execute* request, requesting of the streaming process's description and supplying of three input messages, of which only $fib(0) = 0$ can be processed immediately.



(b) $fib(1) = 1$ is supplied in an input message, and so the previously supplied request for $fib(2)$ and $fib(3)$ can be calculated. $fib(7)$ is still missing prerequisites.



(c) As the dependencies of $fib(4)$ and $fib(5)$ are fulfilled, they can directly be calculated. $fib(7)$ is still missing the result of $fib(6)$.



(d) At the time the request for $fib(6)$ arrives, all preconditions are met and $fib(6)$ and the final $fib(7)$ can be calculated. Once the result has arrived, the client stops the streaming process.

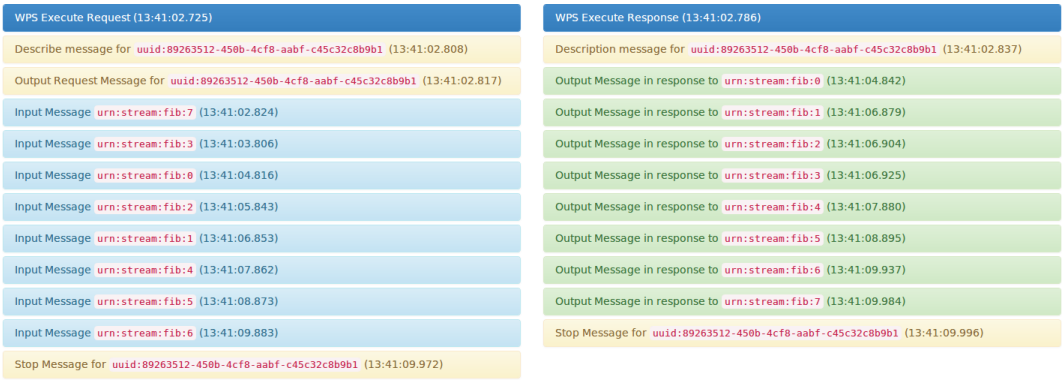


Figure 4.8: Calculation of the seventh Fibonacci number using the Streaming WPS, its accompanying JavaScript API and a simple addition WPS process as the streaming process's delegate.

start and stop streaming processes, to supply input messages, to request the process's description as well as to retrieve outputs. A basic demonstration of the Streaming WPS's capabilities can be seen in Figure 4.8, in which the seventh Fibonacci number is calculated using a streaming process. Fibonacci numbers are the values of the integer sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ defined inductively by the function $fib(n) = fib(n-1) + fib(n-2)$ with $fib(0) = 0$ and $fib(1) = 1$. $fib(n)$ is thereby called the *n-th Fibonacci number* (Bóna, 2011). Through their recursive definition, Fibonacci number calculation can be used to showcase the advanced dependency resolution capabilities of the Streaming WPS. The generic streaming process is started with a reference to another WPS process, which sole functionality is to add two integers and to return the result. Each Fibonacci number is then defined in its own streaming iteration by supplying an input message to the streaming process. To break down the principle of calculating Fibonacci numbers to the addition of two numbers, $fib(0)$ is defined as the addition of 0 and 0 and $fib(1)$ as the addition of 0 and 1. All following Fibonacci numbers are defined as the addition of the results of the two previous streaming iterations. By sending the input messages in a random order, the streaming process has to postpone calculation of a streaming iteration until the input messages of the two referenced streaming iterations have arrived and their results are available. This can be seen from the intermediate results containing all Fibonacci numbers previous to the one requested that arrive in order.

4.7 Streaming LakeAnalyzer WPS

The implementation of a streaming variant of the LakeAnalyzer process presented in Section 3.5 has to be divided in two different use cases. Processing of large data sets of several lakes can be easily accomplished using the generic streaming process implementation presented in the previous section. Configuration parameters for down-sampling rates, error correction and plotting configuration can be supplied using static input parameters, whereas the streaming input messages contain bathymetry, wind and water measurements and optionally salinity information. In this case, the greater data set of time series for multiple lakes is broken into smaller chunks spatially, whereas the temporal dimension is not modified.

By splitting a data set temporally, the analysis of very long time series or near real time buoys data is possible. In theory, streaming chunks can be reduced to a single point in

time by deactivating the down sampling code of the LakeAnalyzer. This would allow to offer characteristics as single values instead of CSV files. Because the LakeAnalyzer can not maintain state across streaming iterations but has to operate on a single point in time, error checking, outlier detection and temporal averaging would not be possible using the LakeAnalyzer as a delegate to the generic streaming process. By this, outputs like Lake or Wedderburn Number are more prone to error or may not be meaningful, e.g. mode-1 vertical seiche period. Development of a stateful streaming process that overcomes this issues would, however, neglect the efforts and possibilities already done in the LakeAnalyzer.

An easier solution to enable near real time or live analysis is to find an appropriate time frame adapted to the sampling rate of sensors deployed in a lake, which is large enough to allow temporal averaging and error checking but is small enough to produce continuous outputs. This approach could be realized using the generic streaming algorithm implementation: the process is initialized using a reference to the LakeAnalyzer process, and bathymetry, configuration parameters and plotting parameters are supplied as static inputs. Water temperature, wind speed and optionally level and salinity measurements are supplied in each streaming iteration. The supplied time frame should ideally be chosen such that after down sampling, smoothing and error correction only a single point in time is outputted.

Future extensions of the LakeAnalyzer, e.g. calculations of parameters that require the previous results, or analysis of chains of lakes in which lakes depend on each other, could also be easily realized. By implementing previous results or the results of a previous lake in the chain as additional inputs to the LakeAnalyzer, these dependencies can be easily resolved by the streaming process.

5 Discussion, Conclusion & Future Work

This thesis had to deal with two independent sub problems: on the one hand, the domain expert friendly deployment of MATLAB models and algorithms as interoperable web services and on the other hand, the efficient processing of large data sets and the analysis of live or near real-time data.

The OGC Web Processing Service presents an ideal solution for web based processing of spatiotemporal data in an interoperable manner and allows the reuse of developed models and algorithms using a standardized interface. The introduced MATLAB WPS facilitates domain experts to expose domain specific models as WPS processes easily. This allows the participation of MATLAB software in the Model Web (Geller and Melton, 2008), which promotes usage of models in processing chains and multi-model computations. The MATLAB WPS is not only of great importance for interoperable model development and sharing of research results, but thus also allows advanced processing chains that bridge the gap between different domains. By this, the MATLAB WPS supports the advance of research and can help to gain new insights by connecting domains, topics and themes that were previously considered independent.

Future improvements to the MATLAB WPS may include selective output generation so that only outputs requested by the client are generated. This can greatly reduce processing time and thus costs. Also the generation of different output formats has to be solved without introducing additional complexity. Both points may be solved by offering WPS process meta data to the MATLAB process (e.g. using a global variable). This should also include informations like the WPS's URL, etc. The LakeAnalyzer shows that MATLAB functions that use a multitude of inputs and/or outputs quickly become hard to read and thus hard to manage. The optional encoding of inputs and outputs as structures could improve maintenance efforts for these kind of processes.

Streaming of spatiotemporal data offers a great solution to the problem of processing large data sets or live environmental – and thus indefinite large – data sets. Previous approaches

are insufficient in regard to compatibility, efficiency and ultimately effectiveness. This is largely caused by the constraints and limitations imposed by the WPS standard. While being generic in terms of which processes can be encapsulated, it enforces strict constraints about how these processes are executed and how they can be described. Even though the asynchronous process execution offers a great leverage point for implementing streaming processes, the concept is too inflexible and lacks capabilities to transport intermediate outputs or even a reference to these. Processes are considered entities that can be executed and described, whereas process executions (or process instances) are not identifiable resources, but are defined as the inaccessible procedure of the *Execute* operation. Apart from the fact that it is impossible to supply subsequent input parameters to a process, other operations that could be considered useful in various use cases are not designated. Even though status responses of asynchronously executed processes can have the status *paused*, pausing/resuming or aborting of running processes can not be accomplished using the WPS interface and can not be implemented as long as process instances can not be identified. Furthermore, the WPS standard is fixed to HTTP and other transport layers that can be considered ideal for streaming, like WebSockets, are not envisaged. As the WPS specification itself, the process description format is simple and thus is easily adoptable but eventually lacks expressiveness. Dependencies between inputs or dependencies between specific outputs and inputs can not be expressed and extensions, e.g. the differentiation between intermediate and final results, are not foreseen.

The approach taken in this thesis to implement streaming processing of geospatial data by breaking out of the WPS standard has to be critically evaluated, as it also breaks compatibility to existing client solutions. But as the above-mentioned limitations show, it is ultimately required to bypass the WPS specification in order to enable true streaming processing. Nevertheless, client solutions need to be adjusted to streaming inputs and outputs. This is even true for previous approaches that target WPS compatibility. Implementing the here described concept of streaming by sending and receiving messages using WebSockets does not require more intrusive changes than the comprehension of playlist files, the continuous polling for new outputs, the provision of data fragments as resources that can be referenced, or the maintenance of an accessible playlist file. By reusing WPS terminologies and technologies, not only large parts of existing client solutions can be reused, but many existing process implementations can be transformed to streaming processes. Due to its advanced dependency handling and the capability to handle multiple streaming inputs and outputs, the presented Streaming WPS is also applicable to more use cases.

Future research should evaluate the not yet finished and published WPS 2.0 specification (Open Geospatial Consortium, 2014). Even though the standardization process is not transparent to non-OGC members, the public available change requests indicate a rethinking of the status of process executions (e.g. Open Geospatial Consortium, 2012c). Despite increasing the compatibility to the WPS standard, future research should concentrate on an in-depth performance evaluation of the Streaming WPS. Additionally, the generic streaming process should be enhanced to be able to request non-default formats from delegates. Future development should also conceptualize a mechanism to declare which outputs should be outputted, which are only for internal use, or which are not required at all. The current approach of identical input and output definitions for referenced inputs should be enhanced to enable a conversion between different formats. Dependency resolution mechanisms should be enhanced so that true indefinite processing of live data becomes possible. The current approach indefinitely keeps references to previous streaming iterations with the result that at one point the dependency graph would not fit into the server's memory. An automatic eviction of old streaming iterations or the optional deactivation of dependency resolution should be examined. Web browser clients that connect only occasionally to a streaming process should also be able to request outputs that were generated prior to the output request message and not only upcoming outputs. For this, a (temporal restricted) replay queue could be developed.

The most important topic for future research is the development of appropriate client solutions that are able to supply inputs to a streaming process and are able to visualize their outputs. Besides that, similar extensions to OGC web services for data warehousing (e.g. WCS or SOS) need to be developed and existing approaches (e.g. for the WFS, see Aydin et al., 2006) need to be integrated.

This thesis showed on the example of LakeAnalyzer how MATLAB analysis software can be exposed as interoperable WPS processes and how streaming can be used to process (indefinite) large data sets in a service-oriented environment, but forthcoming work has yet to demonstrate how the LakeAnalyzer and its streaming variant can be integrated into existing work flows or environments, and which advanced models and processing chains can be developed using the MATLAB and Streaming WPS.

Bibliography

B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

G. Aydin, G. C. Fox, H. Gadgil, and M. E. Pierce. Streaming data services to support archival and real-time geographical information system grids. In *Sixth Annual NASA Earth Science Technology Conference*, 2006.

B. Baranski. Grid computing enabled web processing service. *Proceedings of the 6th Geographic Information Days, IfGI prints*, 32:243–256, 2008.

B. Baranski, T. Foerster, B. Schäffer, and K. Lange. Matching INSPIRE quality of service requirements with hybrid clouds. *Transactions in GIS*, 15(s1):125–142, 2011.

O. Ben-Kiki, C. Evans, and B. Ingerson. YAML Ain't Markup Language (YAML™) 1.2, Oct. 2009. URL http://www.yaml.org/spec/1.2/spec.html. Last retrieved May 4, 2014.

P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C recommendation, W3C, Oct. 2004. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

M. Bóna. *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory*. World Scientific Publishing Co. Pte. Ltd, third edition, 2011.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture Volume 1: A System of Patterns. 1996.

J. J. Cole, Y. T. Prairie, N. F. Caraco, W. H. McDowell, L. J. Tranvik, R. G. Striegl, C. M. Duarte, P. Kortelainen, J. A. Downing, J. J. Middelburg, and J. Melack. Plumbing the global carbon cycle: Integrating inland waters into the terrestrial carbon budget. *Ecosystems*, 10(1):172–185, 2007. ISSN 1432-9840.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

A. Deveria. Can I use... Compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers. Website, 2014. http://caniuse.com/.

L. Di, A. Chen, W. Yang, and P. Zhao. The integration of grid technology with OGC web services (OWS) in NWGISS for NASA EOS data. *GGF8 & HPDC12*, pages 24–27, 2003.

E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

J. A. Downing, Y. T. Prairie, J. J. Cole, C. M. Duarte, L. J. Tranvik, R. G. Striegl, W. H. McDowell, P. Kortelainen, N. F. Caraco, J. M. Melack, and J. J. Middelbiurg. The global abundance and size distribution of lakes, ponds, and impoundments. *Limnology and Oceanography*, 51(5):2388–2397, 2006. ISSN 1939-5590.

European Computer Machinery Association. Standard ECMA-334: C# Language Specification, June 2006.

I. Fette and A. Melnikov. The Websocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. URL http://www.ietf.org/rfc/rfc6455.txt.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Dec. 1999. URL http://www.ietf.org/rfc/rfc2616.txt.

T. Foerster, B. Baranski, and H. Borsutzky. Live geoinformation with standardized geoprocessing services. In *Bridging the Geographic Information Sciences*, pages 99–118. Springer, 2012.

G. N. Geller and F. Melton. Looking forward: Applying an ecological model web to assess impacts of climate change. *Biodiversity*, 9(3-4):79–83, 2008.

I. Hickson. The WebSocket API. W3C Candidate Recommendation, W3C, Sept. 2012. URL http://www.w3.org/TR/2012/CR-websockets-20120920/.

M. Hinz, D. Nüst, B. Proß, and E. Pebesma. Spatial Statistics on the Geospatial Web. In *Short Paper Proceedings of the 16th AGILE Conference on Geographic Information Science, Leuven, Belgium*, May 2013.

G. E. Hutchinson. *A Treatise on Limnology: Vol. I. Geography, Physics and Chemistry*. John Wiley & Sons, Inc., New York, 1957.

S. B. Idso. On the concept of lake stability. *Limnology and Oceanography*, 18:681–683, 1973.

IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, Aug. 2008.

J. Imberger and J. C. Patterson. Physical limnology. *Advances in Applied Mechanics*, 27: 303–475, 1990.

S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006. URL http://www.ietf.org/rfc/rfc4648.txt.

JSR-175 Experts Group. A Metadata Facility for the Java™ Programming Language, Sept. 2004. URL https://jcp.org/en/jsr/detail?id=175.

D. Jungnickel. *Graphs, Networks and Algorithms*. Number 5 in Algorithms and Computation in Mathematics. Springer, fourth edition, 2012.

Y. Lafon, M. Gudgin, M. Hadley, M. Moreau, N. Mendelsohn, A. Karmarkar, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, Apr. 2007. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

S. Lanig, A. Schilling, B. Stollberg, and A. Zipf. Towards standards-based processing of digital elevation models for grid computing through web processing service (WPS). In *Computational Science and Its Applications–ICCSA 2008*, pages 191–203. Springer, 2008.

P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005. URL http://www.ietf.org/rfc/rfc4122.txt.

A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.

S. Monismith. An experimental study of the upwelling response of stratified reservoirs to surface shear-stresses. *Journal of Fluid Mechanics*, 171:407–439, 1986.

National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3. Technical report, Department of Commerce, Aug. 2008.

Open Geospatial Consortium. OpenGIS® Web Map Server. Implementation Specification, OGC, Mar. 2006. URL http://portal.opengeospatial.org/files/06-042.

Open Geospatial Consortium. OpenGIS® Catalogue Services Specification. Implementation Specification, OGC, Feb. 2007a. URL http://portal.opengeospatial.org/files/07-006r1.

Open Geospatial Consortium. OpenGIS® Geography Markup Language (GML) Encoding Standard. Implementation Specification, OGC, Aug. 2007b. URL https://portal.opengeospatial.org/files/07-036.

Open Geospatial Consortium. OGC Web Services Common Specification. Implementation Specification, OGC, Feb. 2007c. URL http://portal.opengeospatial.org/files/06-121r3.

Open Geospatial Consortium. OpenGIS® Web Processing Service. Implementation Specification, OGC, June 2007d. URL http://portal.opengeospatial.org/files/05-007r7.

Open Geospatial Consortium. OpenGIS Web Feature Service 2.0 Interface Standard. Implementation Specification, OGC, Nov. 2010. URL https://portal.opengeospatial.org/files/09-025r1.

Open Geospatial Consortium. OGC® Sensor Observation Service Interface Standard. Implementation Specification, OGC, Apr. 2012a. URL http://portal.opengeospatial.org/files/12-006.

Open Geospatial Consortium. OGC® WCS 2.0 Interface Standard- Core: Corrigendum. Implementation Specification, OGC, July 2012b. URL https://portal.opengeospatial.org/files/09-110r4.

Open Geospatial Consortium. Web Processing Service Change Request – methods for controlling, and checking the status of asynchronous process. Change Request, OGC, July 2012c. URL http://portal.opengeospatial.org/files/09-109r1.

Open Geospatial Consortium. Web Processing Service 2.0 SWG, 2014. URL http://www.opengeospatial.org/projects/groups/wps2.0swg. Last retrieved May 4, 2014.

Oracle Corporation. Javadoc Tool Home Page, May 2013. URL http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html. Last retrieved May 4, 2014.

R. Pantos and W. May. HTTP Live Streaming. Internet Draft (Informational), Oct. 2013. URL http://tools.ietf.org/id/draft-pantos-http-live-streaming-12.txt.

D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics*, 11:1–7, 2007. ISSN 1084-6654.

R Core Team. The R Project for Statistical Computing, 2014. URL http://www.r-project.org/. Last retrieved May 4, 2014.

J. S. Read and K. Muraoka. *LakeAnalyzer – Ver. 3.3 User Manual*. Global Lake Ecological Observatory Network, Mar. 2011.

J. S. Read, D. P. Hamilton, I. D. Jones, K. Muraoka, L. A. Winslow, R. Kroiss, C. H. Wu, and E. Gaiser. Derivation of lake mixing and stratification indices from high-resolution lake buoy data. *Environmental Modelling & Software*, 26(11):1325–1336, 2011. ISSN 1364-8152.

J. S. Read, L. A. Winslow, G. A. Hansen, J. Van Den Hoek, C. D. Markfort, and N. Booth. Upscaling aquatic ecology: Linking continental data products to distributed lake models, 2013. Poster, EarthCube Inland Waters meeting.

T. Rogers, M. Hadley, and M. Gudgin. Web Services Addressing 1.0 - Core. W3C recommendation, W3C, May 2006. URL http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. URL http://www.ietf.org/rfc/rfc3261.txt.

W. Schmidt. Über die Temperatur-und Stabilitätsverhältnisse von Seen. *Geografiska annaler*, pages 145–177, 1928.

H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), Apr. 1998. URL http://www.ietf.org/rfc/rfc2326.txt.

H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Intenet Standard), July 2003. URL http://www.ietf.org/rfc/rfc3550.txt.

The MathWorks, Inc. The MathWorks, Inc. Software License Agreement, Sept. 2013.

The MathWorks, Inc. MATLAB Documentation Center, 2014. URL http://www.mathworks.com/help/matlab/index.html. Last retrieved May 4, 2014.

R. O. R. Y. Thompson and J. Imperger. Response of a numerical model of a stratified lake to wind stress. In *Proceedings of the 2nd Int. Symp. Stratified Flows, Trondheim, Norway*, volume 1, pages 562–570, June 1980.

S. Urbanek. Rserve – A fast way to provide R functionality to applications. In K. Hornik, F. Leisch, and A. Zeileis, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing, Vienna, Austria*, Mar. 2003.

A. van Kesteren et al. Cross-Origin Resource Sharing. W3C Recommendation, W3C, Jan. 2014. URL http://www.w3.org/TR/2014/REC-cors-20140116/.

Appendix

A Lake Analyzer Process Wrapper Function

```
function [ results,   resultsWtr,  StFig,     uStFig,     ...
           LnFig,     WFig,        wTempFig,  wndSpdFig,  ...
           metaTFig,  metaBFig,    thermDFig, SthermDFig, ...
           SmetaBFig, SmetaTFig,   SuStFig,   SLnFig,     ...
           SWFig,     N2Fig,       SN2Fig,    T1Fig,      ...
           ST1Fig ] = Run_LA_WPS(...
            bthFileName, lvlFileName, wndFileName, wtrFileName, salFileName,   ...
            outputResolution, totalDepth, windHeight, windAveraging,           ...
            layerAveraging, outlierWindow, maxWaterTemp,  minWaterTemp,        ...
            maxWindSpeed,  minWindSpeed, metaMinSlope,  mixedTempDifferential, ...
            figRes, figUnits, figWidth, figHeight, leftMargin, rightMargin,    ...
            topMargin, botMargin, fontName, fontSize, heatMapMin, heatMapMax)

  pltMods = struct('figUnits',    figUnits,            ...
                   'figWidth',    num2str(figWidth),   ...
                   'figHeight',   num2str(figHeight),  ...
                   'leftMargin',  num2str(leftMargin), ...
                   'rightMargin', num2str(rightMargin), ...
                   'topMargin',   num2str(topMargin),  ...
                   'botMargin',   num2str(botMargin),  ...
                   'figType',     'png',               ...
                   'figRes',      num2str(figRes),     ...
                   'fontName',    fontName,            ...
                   'fontSize',    num2str(fontSize),   ...
                   'heatMapMin',  num2str(heatMapMin), ...
                   'heatMapMax',  num2str(heatMapMax));

  inFileNames = struct('bthFileName', bthFileName, ...
                       'wndFileName', wndFileName, ...
                       'wtrFileName', wtrFileName);

  % null inputs are encoded as NaN
  if ~isnan(salFileName)
      inFileNames.salFileName = salFileName;
  end
  if ~isnan(lvlFileName)
      inFileNames.lvlFileName = lvlFileName;
  end

  outFileNames = struct('results',    [tempname '.out'], ...
                        'resultsWtr', [tempname '.out'], ...
                        'StFig',      [tempname '.out'], ...
                        'uStFig',     [tempname '.out'], ...
                        'LnFig',      [tempname '.out'], ...
                        'WFig',       [tempname '.out'], ...
```

```
                        'wTempFig',   [tempname '.out'], ...
                        'wndSpdFig',  [tempname '.out'], ...
                        'metaTFig',   [tempname '.out'], ...
                        'metaBFig',   [tempname '.out'], ...
                        'thermDFig',  [tempname '.out'], ...
                        'SthermDFig', [tempname '.out'], ...
                        'SmetaBFig',  [tempname '.out'], ...
                        'SmetaTFig',  [tempname '.out'], ...
                        'SuStFig',    [tempname '.out'], ...
                        'SLnFig',     [tempname '.out'], ...
                        'SWFig',      [tempname '.out'], ...
                        'N2Fig',      [tempname '.out'], ...
                        'SN2Fig',     [tempname '.out'], ...
                        'T1Fig',      [tempname '.out'], ...
                        'ST1Fig',     [tempname '.out']);

  LA({'St', 'uSt', 'Ln', 'W', 'wTemp', 'wndSpd', 'metaT', 'metaB', 'thermD', ...
      'SthermD', 'SmetaB', 'SmetaT', 'SuSt', 'SLn', 'SW', 'N2', 'SN2', 'T1', ...
      'ST1'},                                                                 ...
    outputResolution, totalDepth, windHeight, windAveraging, layerAveraging, ...
    outlierWindow, maxWaterTemp, minWaterTemp, maxWindSpeed, minWindSpeed,   ...
    metaMinSlope, mixedTempDifferential, inFileNames, outFileNames, pltMods, ...
    1, 1);

  results     = outFilenames.results
  resultsWtr  = outFilenames.resultsWtr
  StFig       = outFilenames.StFig
  uStFig      = outFilenames.uStFig
  LnFig       = outFilenames.LnFig
  WFig        = outFilenames.WFig
  wTempFig    = outFilenames.wTempFig
  wndSpdFig   = outFilenames.wndSpdFig
  metaTFig    = outFilenames.metaTFig
  metaBFig    = outFilenames.metaBFig
  thermDFig   = outFilenames.thermDFig
  SthermDFig  = outFilenames.SthermDFig
  SmetaBFig   = outFilenames.SmetaBFig
  SmetaTFig   = outFilenames.SmetaTFig
  SuStFig     = outFilenames.SuStFig
  SLnFig      = outFilenames.SLnFig
  SWFig       = outFilenames.SWFig
  N2Fig       = outFilenames.N2Fig
  SN2Fig      = outFilenames.SN2Fig
  T1Fig       = outFilenames.T1Fig
  ST1Fig      = outFilenames.ST1Fig
end
```

B Lake Analyzer Process Configuration

```yaml
---
connection: ws://localhost:7000
identifier: org.gleon.LakeAnalyzer
version: 1.0.0
title: Lake Analyzer
abstract: Lake Analyzer
function: Run_LA_WPS
inputs:
  # input files
  - identifier: bathymetry
    title: Bathymetry
    abstract: >
      A bathymetry file is a tab delimited text file with extension of [.bth].
      The file starts from one line header and followed by the hypsographic data
      at each depth. Depths must start from zero (i.e. surface) with a unit of
      meters, and hypsographic curve data with area as square meters is followed
      by comma delimiter. If the hypsographic curve is not concluded with zero
      at the bottom, LakeAnalyzer program automatically assigns zero to the
      bottom depth which was defined during the configuration process.
      LakeAnalyzer linearly interpolates the given hypsographic curve. Change to
      the hypsographic curve due to surface elevation change is not supported by
      the current version of the LakeAnalyzer.
    type: &csv
      mimeType: text/csv
  - identifier: waterLevel
    title: Water Level
    abstract: >
      The Water Level file is a tab delimited text file with the file extension
      of [.lvl]. Water level input is optional for all the output parameters. It
      is useful for estuaries and lake with significant level changes which
      affect hypsographic curve of the water body. The effect of water level
      fluctuation to the bathymetry area are calculated when calculating
      stabilities. The water level file contains one header [DateTime
      level(positive Z down)]. From the second line, date/time information with
      the format of [yyyy-mm-dd HH:MM], and water level from the highest
      elevation area measurement available (original depth is the surface level
      stated in the .bth file) should be described. Level depths must be equal
      or greater than 0.
    type: *csv
    minOccurs: 0
  - identifier: windSpeed
    title: Wind Speed
    abstract: >
      The wind speed file is a tab delimitated text file with extension of
      [.wnd]. Wind speed data are used for uStar, Lake Number, and Wedderburn
      Number calculations. Time scale and resolution of the wind speed must
      match the water temperature input parameters. The file starts from one
      line header [dateTime windSpeed]. From the second line, date/time
```

```
    information with the format of [yyyy-mm-dd HH:MM], and wind speed data in
    m/s should be described.
  type: *csv
- identifier: waterTemperature
  title: Water Temperature
  abstract: >
    The water temperature file is a tab delimitated text file with a file
    extension of [.wtr]. The file should contain one header which starts from
    DateTime, followed by individual thermister depths in meters with format
    of [temp5]. LakeAnalyzer uses header information to acquire thermister
    depth. Temperature data should be inserted from the following line. The
    data starts from the date/time input parameters, which should be formatted
    as [yyyy-mm-dd HH:MM].
  type: *csv
- identifier: salinity
  title: Salinity
  abstract: >
    The salinity file is a tab delimitated text file with the file extension
    of [.sal]. Salinity input is optional for all the output parameters. If
    the program locates the salinity file in the correct directory, the effect
    of salinity on the density is calculated during the process. Salinity time
    can be independent to the other input files. The salinity file contains
    one header line starting from DateTime, and followed by depths of
    measurements in format of [salinity2.0]. The second line is the beginning
    of the actual data input parameters, starting from date/time in format
    [yyyy-mm-dd HH:MM]. After tab separation, salinity should be indicated
    Practical Salinity Scale (PSS) units.
  type: *csv
  minOccurs: 0
# .lke file contents
- identifier: outputResolution
  title: Output Resolution
  abstract: >
    Output resolution specifies the time-step of the calculations made for
    Lake Analyzer. If the temporal resolution of the input data is coarser
    than the entry for this input, calculations will be made according to
    input data resolution.
  type: int
  unit: s
- identifier: totalDepth
  title: Total Depth
  abstract: >
    Total depth must be greater or equal to than the maximum depth given in
    the .bth file. If the total depth is not included in the .bth file, it is
    assumed that the area at total depth is 0 (m2) and the depth area curve
    is linearly interpolated from this depth to the values in the .bth file.
  type: double
  unit: m
- identifier: windHeight
  title: Wind Height
  abstract: >
    Height from surface for wind measurement. Height of wind measurement is
    used for the wind speed correction factor.
  type: double
  unit: m
- identifier: windAveraging
  title: Wind Averaging
  abstract: >
    Wind averaging is the backwards-looking smoothing window used for the
    calculation of uSt and SuSt. This calculation allows for the relevant wind
    duration to influence the calculation of wind-derived parameters.
  type: int
  unit: s
```

```yaml
- identifier: layerAveraging
  title: Layer Averaging
  abstract: >
    Thermal averaging is the smoothing window used for metaT, metaB,
    thermD, SmetaT, SmetaB, and SthermD. Temporal smoothing for thermal layers
     is intended to minimize the effects of internal waves on these parameters.
  type: int
  unit: s
- identifier: outlierWindow
  title: Outlier Window
  abstract: >
    Outlier window is the window size (seconds) for outlier removal, where
    measurements outside of the bounds ( $\mu \pm 2.5 \cdot \sigma$ ) based on the standard
    deviation and the mean inside the outlier window are removed. Outlier
    removal is performed on .wtr and .wnd files prior to down-sampling (if
    applicable).
  type: int
  unit: s
- identifier: maxWaterTemp
  title: Maximum Water Temperature
  abstract: >
    Maximum allowed water temperature, where all values of .wtr file not
    fitting this criteria are removed before outlier checking.
  type: double
  unit: °C
  minOccurs: 0
- identifier: minWaterTemp
  title: Minimum Water Temperature
  abstract: >
    Minimum allowed water temperature, where all values of .wtr file not
    fitting this criteria are removed before outlier checking.
  type: double
  unit: °C
  minOccurs: 0
- identifier: maxWindSpeed
  title: Maximum Wind Speed
  abstract: >
    Maximum allowed wind speed, where all values of .wnd file not
    fitting this criteria are removed before outlier checking.
  type: double
  unit: m/s
  minOccurs: 0
- identifier: minWindSpeed
  title: Minimum Wind Speed
  abstract: >
    Minimum allowed wind speed, where all values of .wnd file not
    fitting this criteria are removed before outlier checking.
  type: double
  unit: m/s
  minOccurs: 0
- identifier: metaMinSlope
  title: Minimum Metalimnion slope
  abstract: >
    Minimum slope for the range of the metalimnion, which is used to
    calculated values of metaT, metaB, SmetaT, and SmetaB.
  type: double
  unit: (kg/m^(-3))/m
- identifier: mixedTempDifferential
  title: Mixed Temperature Differential
  abstract: >
    Minimum surface to bottom thermistor temperature differential before
    the case of 'mixed' is applied. When 'mixed' is true, all thermal layer
    calculations are no longer applicable, and values are given as the depth
```

```
      of the bottom thermistor.
    type: double
    unit: °C
  # .plt file contents
  - identifier: figRes
    abstract: Resolution of the figure
    title: Plot Resolution
    type: int
    values: [ 50, 100, 200, 300, 400, 500 ]
    unit: dpi
  - identifier: figUnits
    title: Figure Units
    abstract: Units of measure for figure size
    type: string
    values: [ inches, centimeters, points ]
  - identifier: figWidth
    title: Figure Width
    abstract: Width of figure (relative to figUnits)
    type: double
  - identifier: figHeight
    title: Figure Height
    abstract: Height of figure (relative to figUnits)
    type: double
  - identifier: leftMargin
    title: Left Margin
    abstract: Space between left edge of figure and y-axis (relative to figUnits)
    type: double
  - identifier: rightMargin
    title: Right Margin
    abstract: Space between right edge of figure and right axis
    type: double
  - identifier: topMargin
    title: Top Margin
    abstract: >
      Space between the top edge of the figure and the top of the plot axis
    type: double
  - identifier: botMargin
    title: Bottom Margin
    abstract: >
      Space between the bottom edge of the figure and the bottom of the
      plot x-axis
    type: double
  - identifier: fontName
    title: Font Name
    abstract: Font name for plot text
    type: string
    values: [ Arial, Times New Roman, Helvetica ]
  - identifier: fontSize
    title: Font Size
    abstract: Font sive for plot text
    type: int
    values: [ 8, 9, 10, 11, 12, 14 ]
  - identifier: heatMapMin
    title: Minimum Heat Map Value
    abstract: Value that represents the minimum heatmap color
    type: double
  - identifier: heatMapMax
    title: Maximum Heat Map Value
    abstract: Value that represents the maximum heatmap color
    type: double
outputs:
  - identifier: results
    title: Raw Results
```

```
    type: *csv
  - identifier: results_wtr
    title: Raw Results
    type: *csv
  - identifier: N2
    title: Buoyancy frequency
    type: &png
      mimeType: image/png
      encoding: Base64
  - identifier: SN2
    title: Parent buoyancy frequency
    type: *png
  - identifier: Ln
    title: Lake number
    type: *png
  - identifier: SLn
    title: Parent lake number
    type: *png
  - identifier: metaB
    title: Metalimnion bottom depth
    type: *png
  - identifier: SmetaB
    title: Parent metalimnion bottom depth
    type: *png
  - identifier: metaT
    title: Metalimnion top depth
    type: *png
  - identifier: SmetaT
    title: Parent metalimnion top depth
    type: *png
  - identifier: T1
    title: Mode one vertical seiche period
    type: *png
  - identifier: ST1
    title: Parent mode one vertical seiche period
    type: *png
  - identifier: St
    title: Schmidt stability
    type: *png
  - identifier: thermD
    title: Thermocline depth
    type: *png
  - identifier: SthermD
    title: Parent thermocline depth
    type: *png
  - identifier: uSt
    title: u star (turblent velocity scale from wind)
    type: *png
  - identifier: SuSt
    title: Parent u star (turblent velocity scale from wind)
    type: *png
  - identifier: wTemp
    title: Water temperature
    type: *png
  - identifier: W
    title: Wedderburn number
    type: *png
  - identifier: SW
    title: Parent Wedderburn number
    type: *png
  - identifier: wndSpd
    title: Wind speed
    type: *png
```

C Lake Analyzer Process Description

See Appendix E for omitted XML namespaces.

```
<ProcessDescription statusSupported="true" storeSupported="true"
    wps:processVersion="1.0.0">
  <ows:Identifier>org.gleon.LakeAnalyzer</ows:Identifier>
  <ows:Title>Lake Analyzer</ows:Title>
  <ows:Abstract>Lake Analyzer</ows:Abstract>
  <DataInputs>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>bathymetry</ows:Identifier>
      <ows:Title>Bathymetry</ows:Title>
      <ows:Abstract>A bathymetry file is a tab delimited text file with extension
          of [.bth]. The file starts from one line header and followed by the
          hypsographic data at each depth. Depths must start from zero (i.e.
          surface) with a unit of meters, and hypsographic curve data with area
          as square meters is followed by comma delimiter. If the hypsographic
          curve is not concluded with zero at the bottom, LakeAnalyzer program
          automatically assigns zero to the bottom depth which was defined during
           the configuration process. LakeAnalyzer linearly interpolates the
          given hypsographic curve. Change to the hypsographic curve due to
          surface elevation change is not supported by the current version of the
           LakeAnalyzer.</ows:Abstract>
      <ComplexData>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>waterLevel</ows:Identifier>
      <ows:Title>Water Level</ows:Title>
      <ows:Abstract>The Water Level file is a tab delimited text file with the
          file extension of [.lvl]. Water level input is optional for all the
          output parameters. It is useful for estuaries and lake with significant
           level changes which affect hypsographic curve of the water body. The
          effect of water level fluctuation to the bathymetry area are calculated
           when calculating stabilities. The water level file contains one header
           [DateTime level(positive Z down)]. From the second line, date/time
          information with the format of [yyyy-mm-dd HH:MM], and water level from
           the highest elevation area measurement available (original depth is
```

```
          the surface level stated in the .bth file) should be described. Level
          depths must be equal or greater than 0.</ows:Abstract>
      <ComplexData>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>windSpeed</ows:Identifier>
      <ows:Title>Wind Speed</ows:Title>
      <ows:Abstract>The wind speed file is a tab delimitated text file with
          extension of [.wnd]. Wind speed data are used for uStar, Lake Number,
          and Wedderburn Number calculations. Time scale and resolution of the
          wind speed must match the water temperature input parameters. The file
          starts from one line header [dateTime windSpeed]. From the second line,
           date/time information with the format of [yyyy-mm-dd HH:MM], and wind
          speed data in m/s should be described.</ows:Abstract>
      <ComplexData>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>waterTemperature</ows:Identifier>
      <ows:Title>Water Temperature</ows:Title>
      <ows:Abstract>The water temperature file is a tab delimitated text file with
           a file extension of [.wtr]. The file should contain one header which
          starts from DateTime, followed by individual thermister depths in
          meters with format of [temp5]. LakeAnalyzer uses header information to
          acquire thermister depth. Temperature data should be inserted from the
          following line. The data starts from the date/time input parameters,
          which should be formatted as [yyyy-mm-dd HH:MM].</ows:Abstract>
      <ComplexData>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>salinity</ows:Identifier>
```

```
      <ows:Title>Salinity</ows:Title>
      <ows:Abstract>The salinity file is a tab delimitated text file with the file
           extension of [.sal]. Salinity input is optional for all the output
          parameters. If the program locates the salinity file in the correct
          directory, the effect of salinity on the density is calculated during
          the process. Salinity time can be independent to the other input files.
           The salinity file contains one header line starting from DateTime, and
           followed by depths of measurements in format of [salinity2.0]. The
          second line is the beginning of the actual data input parameters,
          starting from date/time in format [yyyy-mm-dd HH:MM]. After tab
          separation, salinity should be indicated Practical Salinity Scale (PSS)
           units.</ows:Abstract>
      <ComplexData>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>outputResolution</ows:Identifier>
      <ows:Title>Output Resolution</ows:Title>
      <ows:Abstract>Output resolution specifies the time-step of the calculations
          made for Lake Analyzer. If the temporal resolution of the input data is
           coarser than the entry for this input, calculations will be made
          according to input data resolution.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <UOMs>
          <Default>
            <ows:UOM>s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>totalDepth</ows:Identifier>
      <ows:Title>Total Depth</ows:Title>
      <ows:Abstract>Total depth must be greater or equal to than the maximum depth
           given in the .bth file. If the total depth is not included in the .bth
           file, it is assumed that the area at total depth is 0 (m2) and the
          depth area curve is linearly interpolated from this depth to the values
           in the .bth file.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>m</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>m</ows:UOM>
          </Supported>
        </UOMs>
```

```
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>windHeight</ows:Identifier>
      <ows:Title>Wind Height</ows:Title>
      <ows:Abstract>Height from surface for wind measurement. Height of wind
           measurement is used for the wind speed correction factor.</ows:Abstract
           >
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>m</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>m</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>windAveraging</ows:Identifier>
      <ows:Title>Wind Averaging</ows:Title>
      <ows:Abstract>Wind averaging is the backwards-looking smoothing window used
           for the calculation of uSt and SuSt. This calculation allows for the
           relevant wind duration to influence the calculation of wind-derived
           parameters.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <UOMs>
          <Default>
            <ows:UOM>s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>layerAveraging</ows:Identifier>
      <ows:Title>Layer Averaging</ows:Title>
      <ows:Abstract>Thermal averaging is the smoothing window used for metaT,
           metaB, thermD, SmetaT, SmetaB, and SthermD. Temporal smoothing for
           thermal layers is intended to minimize the effects of internal waves on
            these parameters.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <UOMs>
          <Default>
            <ows:UOM>s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
```

```
      <ows:Identifier>outlierWindow</ows:Identifier>
      <ows:Title>Outlier Window</ows:Title>
      <ows:Abstract>Outlier window is the window size (seconds) for outlier
           removal, where measurements outside of the bounds ( $\mu \pm 2.5 \cdot \sigma$ ) based
           on the standard deviation and the mean inside the outlier window are
           removed. Outlier removal is performed on .wtr and .wnd files prior to
           down-sampling (if applicable).</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <UOMs>
          <Default>
            <ows:UOM>s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>maxWaterTemp</ows:Identifier>
      <ows:Title>Maximum Water Temperature</ows:Title>
      <ows:Abstract>Maximum allowed water temperature, where all values of .wtr
           file not fitting this criteria are removed before outlier checking.</
           ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>°C</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>°C</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>minWaterTemp</ows:Identifier>
      <ows:Title>Minimum Water Temperature</ows:Title>
      <ows:Abstract>Minimum allowed water temperature, where all values of .wtr
           file not fitting this criteria are removed before outlier checking.</
           ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>°C</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>°C</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>maxWindSpeed</ows:Identifier>
      <ows:Title>Maximum Wind Speed</ows:Title>
      <ows:Abstract>Maximum allowed wind speed, where all values of .wnd file not
           fitting this criteria are removed before outlier checking.</
```

```
          ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>m/s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>m/s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="0" maxOccurs="1">
      <ows:Identifier>minWindSpeed</ows:Identifier>
      <ows:Title>Minimum Wind Speed</ows:Title>
      <ows:Abstract>Minimum allowed wind speed, where all values of .wnd file not
          fitting this criteria are removed before outlier checking.</
          ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>m/s</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>m/s</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>metaMinSlope</ows:Identifier>
      <ows:Title>Minimum Metalimnion slope</ows:Title>
      <ows:Abstract>Minimum slope for the range of the metalimnion, which is used
          to calculated values of metaT, metaB, SmetaT, and SmetaB.</ows:Abstract
          >
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
            <ows:UOM>(kg/m^(-3))/m</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>(kg/m^(-3))/m</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>mixedTempDifferential</ows:Identifier>
      <ows:Title>Mixed Temperature Differential</ows:Title>
      <ows:Abstract>Minimum surface to bottom thermistor temperature differential
          before the case of 'mixed' is applied. When 'mixed' is true, all
          thermal layer calculations are no longer applicable, and values are
          given as the depth of the bottom thermistor.</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <UOMs>
          <Default>
```

```
            <ows:UOM>°C</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>°C</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>figRes</ows:Identifier>
      <ows:Title>Plot Resolution</ows:Title>
      <ows:Abstract>Resolution of the figure</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <UOMs>
          <Default>
            <ows:UOM>dpi</ows:UOM>
          </Default>
          <Supported>
            <ows:UOM>dpi</ows:UOM>
          </Supported>
        </UOMs>
        <ows:AllowedValues>
          <ows:Value>50</ows:Value>
          <ows:Value>100</ows:Value>
          <ows:Value>200</ows:Value>
          <ows:Value>300</ows:Value>
          <ows:Value>400</ows:Value>
          <ows:Value>500</ows:Value>
        </ows:AllowedValues>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>figUnits</ows:Identifier>
      <ows:Title>Figure Units</ows:Title>
      <ows:Abstract>Units of measure for figure size</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:string"/>
        <ows:AllowedValues>
          <ows:Value>inches</ows:Value>
          <ows:Value>centimeters</ows:Value>
          <ows:Value>points</ows:Value>
        </ows:AllowedValues>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>figWidth</ows:Identifier>
      <ows:Title>Figure Width</ows:Title>
      <ows:Abstract>Width of figure (relative to figUnits)</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AllowedValues/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>figHeight</ows:Identifier>
      <ows:Title>Figure Height</ows:Title>
      <ows:Abstract>Height of figure (relative to figUnits)</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AllowedValues/>
      </LiteralData>
```

```
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>leftMargin</ows:Identifier>
      <ows:Title>Left Margin</ows:Title>
      <ows:Abstract>Space between left edge of figure and y-axis (relative to
          figUnits)</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>rightMargin</ows:Identifier>
      <ows:Title>Right Margin</ows:Title>
      <ows:Abstract>Space between right edge of figure and right axis</
          ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>topMargin</ows:Identifier>
      <ows:Title>Top Margin</ows:Title>
      <ows:Abstract>Space between the top edge of the figure and the top of the
          plot axis</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>botMargin</ows:Identifier>
      <ows:Title>Bottom Margin</ows:Title>
      <ows:Abstract>Space between the bottom edge of the figure and the bottom of
          the plot x-axis</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>fontName</ows:Identifier>
      <ows:Title>Font Name</ows:Title>
      <ows:Abstract>Font name for plot text</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:string"/>
        <ows:AllowedValues>
          <ows:Value>Arial</ows:Value>
          <ows:Value>Times New Roman</ows:Value>
          <ows:Value>Helvetica</ows:Value>
        </ows:AllowedValues>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>fontSize</ows:Identifier>
      <ows:Title>Font Size</ows:Title>
      <ows:Abstract>Font sive for plot text</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:int"/>
        <ows:AllowedValues>
          <ows:Value>8</ows:Value>
          <ows:Value>9</ows:Value>
```

```
          <ows:Value>10</ows:Value>
          <ows:Value>11</ows:Value>
          <ows:Value>12</ows:Value>
          <ows:Value>14</ows:Value>
        </ows:AllowedValues>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>heatMapMin</ows:Identifier>
      <ows:Title>Minimum Heat Map Value</ows:Title>
      <ows:Abstract>Value that represents the minimum heatmap color</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>heatMapMax</ows:Identifier>
      <ows:Title>Maximum Heat Map Value</ows:Title>
      <ows:Abstract>Value that represents the maximum heatmap color</ows:Abstract>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
  </DataInputs>
  <ProcessOutputs>
    <Output>
      <ows:Identifier>results_wtr</ows:Identifier>
      <ows:Title>Raw Results</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>results</ows:Identifier>
      <ows:Title>Raw Results</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>text/csv</MimeType>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>N2</ows:Identifier>
      <ows:Title>Buoyancy frequency</ows:Title>
      <ComplexOutput>
```

```
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SN2</ows:Identifier>
      <ows:Title>Parent buoyancy frequency</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>Ln</ows:Identifier>
      <ows:Title>Lake number</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SLn</ows:Identifier>
      <ows:Title>Parent lake number</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
```

```
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>metaB</ows:Identifier>
      <ows:Title>Metalimnion bottom depth</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SmetaB</ows:Identifier>
      <ows:Title>Parent metalimnion bottom depth</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>metaT</ows:Identifier>
      <ows:Title>Metalimnion top depth</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SmetaT</ows:Identifier>
      <ows:Title>Parent metalimnion top depth</ows:Title>
      <ComplexOutput>
```

```
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>T1</ows:Identifier>
      <ows:Title>Mode one vertical seiche period</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>ST1</ows:Identifier>
      <ows:Title>Parent mode one vertical seiche period</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>St</ows:Identifier>
      <ows:Title>Schmidt stability</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
```

```
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>thermD</ows:Identifier>
      <ows:Title>Thermocline depth</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SthermD</ows:Identifier>
      <ows:Title>Parent thermocline depth</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>uSt</ows:Identifier>
      <ows:Title>u star (turblent velocity scale from wind)</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SuSt</ows:Identifier>
      <ows:Title>Parent u star (turblent velocity scale from wind)</ows:Title>
      <ComplexOutput>
```

```
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>wTemp</ows:Identifier>
      <ows:Title>Water temperature</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>W</ows:Identifier>
      <ows:Title>Wedderburn number</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>SW</ows:Identifier>
      <ows:Title>Parent Wedderburn number</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
```

```
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
    <Output>
      <ows:Identifier>wndSpd</ows:Identifier>
      <ows:Title>Wind speed</ows:Title>
      <ComplexOutput>
        <Default>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Default>
        <Supported>
          <Format>
            <MimeType>image/png</MimeType>
            <Encoding>Base64</Encoding>
          </Format>
        </Supported>
      </ComplexOutput>
    </Output>
  </ProcessOutputs>
</ProcessDescription>
```

D Source Code

All source code, including prototypical implementations of the described Streaming WPS and MATLAB WPS, is freely available under an open source license:

Streaming WPS	Extension for the 52°North WPS to allow streaming of input and output parameters over WebSockets. GNU General Public License Version 3 https://github.com/autermann/streaming-wps
MATLAB WPS	Extension for the 52°North WPS to offer MATLAB functions and scripts as OGC Web Processing Service algorithms. GNU General Public License Version 2 https://github.com/autermann/matlab-wps
streaming-wps-js	Streaming WPS JavaScript Bindings Apache License Version 2.0 https://github.com/autermann/streaming-wps-js
WPS Commons	52°North WPS convenience classes and bootstrapping code. GNU General Public License Version 2 https://github.com/autermann/wps-commons
matlab-connector	MATLAB function execution on (pooled) remote MATLAB instances. GNU General Public License Version 3 https://github.com/autermann/matlab-connector
LakeAnalyzer	MATLAB source code for Lake Analyzer GNU General Public License Version 2 https://github.com/autermann/Lake-Analyzer
YAML API	A Jackson-like API to read and create YAML nodes (based on SnakeYAML). Apache License Version 2.0 https://github.com/autermann/yaml

E XML Namespaces

For clarity XML name spaces are omitted in XML Listings. Their respective value can be found in the following table:

Prefix	Namespace
xlink	http://www.w3.org/1999/xlink
xml	http://www.w3.org/XML/1998/namespace
xs	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
soap	http://www.w3.org/2003/05/soap-envelope
wsa	http://www.w3.org/2005/08/addressing
ows	http://www.opengis.net/ows/1.1
wps	http://www.opengis.net/wps/1.0.0
stream	https://github.com/autermann/streaming-wps

Plagiarism Statement

Hiermit versichere ich, dass die vorliegende Arbeit über *Streaming Web-Services for Calculating Live Hydrological Derivatives* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, den 4. Mai 2014 _____

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Münster, den 4. Mai 2014 _____