Master Thesis

# Streaming Web-Services for Calculating Live Hydrological Derivatives

**Christian Autermann**

autermann@uni-muenster.de

Institute for Geoinformatics

University of Münster

**April 16, 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

# Listings

# 1. Introduction

Recent research has highlighted the relevance of lakes to global process such as the carbon cycle (Cole et al., 2007). Ecological studies on lakes have historically taken advantage of the "closed system" bounds to delineate a simplified ecosystem, but analyses that are formulated to answer societally relevant questions often must scale this single system science approach to hundreds, thousands, or millions of lakes (Downing et al., 2006). Therefore systems must be developed that can aggregate, analyze, and ultimately interpret hydrological data at large scales. Additionally, these analytical systems must be able to easily couple lake features with supporting data that define, for example, catchment properties, local climate, and anthropomorphic stressors. These data products are readily available as national coverages that can either be sampled and turned into model parameters, or turned into model drivers if they are time series products.

This work shall evaluate the existing tools (e.g. Lake Analyzer[1], see Read et al., 2011)), data models and the modeling frameworks used by USGS CIDA[2]. Modeling runs are based on online data brokers (such as the USGS's Geo Data Portal - GDP[3]) build upon Open Geospatial Consortium standards such as Catalogue Service for the Web (CSW), Web Processing Service (WPS), Web Map Service (WMS) and Web Coverage Service (WCS), but still rely on local algorithms, which 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 infrastructure for analyzing foundational data used by domain scientists is an important challenge given legacy and heterogeneous architectures. Therefore building on the existing infrastructure and corresponding demands of the use case shall be considered.

One approach for a scalable system is to move the modeling to a web-based processing framework, which should rely on public and interoperable standards in the given use case. Web processing allows to chain data brokers with translators, models, and eventually posthoc analysis of model runs. This chain provides specific information products to the user. Considering the amount of data (and future process scaling needs), such an analysis must 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 the lag for domain users of the system. Existing approaches to this problem shall be critically evaluated.

This thesis work comprises the evaluation, design and prototypical implementation of a

---

1. https://github.com/GLEON/Lake-Analyzer
2. http://cida.usgs.gov/
3. http://cida.usgs.gov/gdp/

lake analysis chain for live sensor data. This includes the evaluation of existing datamodels (mainly CSV/TSV) and a standardized way to convert existing domain specific applications written in MatLab[4] into streaming web services (possibly WPS algorithms) in favor of the currently used non standardized web frontend[5].

**Research Questions**

- How can large scale hydrological data be processed in a service-based processing chain?
- Do available web-processing interface definitions support a live data streaming scenario, what is missing?
- Can real-time data be integrated into the processing chain for a constant (streamed) analysis?
- How does the developed architecture perform in practical test with 1000s and 10000s of lake features?
- How can continued statistical quality assurance and quality control in the application area of lake ecology be modeled in a web service chain?
- Do existing standards (data models and service interfaces for data warehousing, processing and visualization) support a streaming analysis chain? What is missing?
- How can spatial dependencies between streamed features be considered?
- How can a analysis language commonly used by domain experts (in this case MatLab) be easily deployed in a web based processing chain?

---

4. http://www.mathworks.de/products/matlab/
5. http://lakeanalyzer.gleon.org/

# 2. Lake-Analyzer

# 3. Web Processing Service

The WPS (Open Geospatial Consortium, 2007) is the quasi standard for web based processing of spatiotemporal data (Foerster et al., 2012). It is an open service standard specified by the Open Geospatial Consortium (OGC) and is embedded in the OGC Web Services Common (OWS) 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 or cloud computing, as well as super computer processing are hidden behind a simple to use service interface and become accessible.

The WPS specifies mechanism 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 HTTP protocol. 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* methods, that can be used to request a detailed description of the service and it's capabilities. It offers a service identification and provider structures, which contain informations about the organization operating the WPS and informational meta data about the service instance and that can be used for service discovery. Besides that it contains detailed information about supported operations, bindings, languages and a list of available processes.

The detailed description of a single process may be requested 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 asynchronously or synchronously and how the results of the process should be encoded.

Typical interaction patterns of the Web Processing Service are depicted in Figure 1. During process discovery *GetCapabilities* and *DescribeProcess* are used to request a list of available processes and their description. Process executions takes place either synchronously or asynchronously by issuing a *Execute* request to a specific process. In the case of asynchronously process executions, the WPS returns URL to a *ExecuteResponse* which is continuously updated and which the client can regular request to get the current process status.
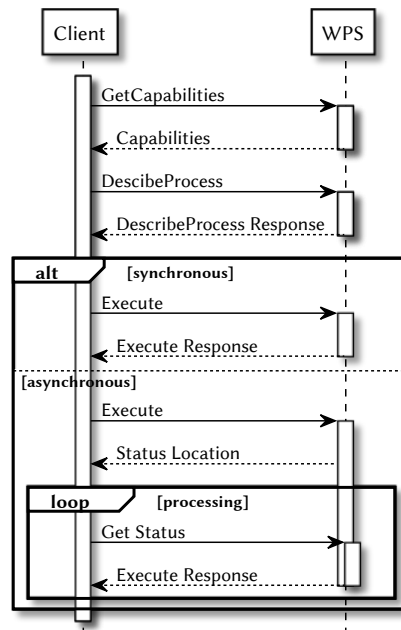


**Figure 1:** Typical interaction patterns of the Web Processing Service. Process discovery using *GetCapabilities* and *DescribeProcess*, synchronous and asynchronous process execution using *Execute*.

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 GML 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 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 is data, that 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 in arbitrary dimension, that is described by a coordinate reference system (CRS).

TODO: Überleitungssatz zum Matlab-WPS

## 4. Matlab WPS

MATLAB[6] 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)[7], today MATLAB is widespread across different domains in academics, engineering and industry. The base program is extensible 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 using *Matlab Central*, a repository of user contributions, mostly licensed under the two-clause BSD license[8][9].

Creating a specific WPS process implementation for the Lake Analyzer would be possible, but considering the wide spread usage of MATLAB based scripts and applications, a generic solution, that enables the easy deployment of MATLAB based functionalities as Web Processing Service 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 Lake Analyzer 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 a extensive programming experience beyond MATLAB. Domain expert 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 of other programming languages or an comprehensive expertise in web services or their development. To accomplish this, no switch from MATLAB to other languages should be needed and rather complex and verbose process descriptions should not be manually be written, but automatically generated. A key goal of the MATLAB WPS to expose existing models and algorithms as WPS processes and so the procedure to convert a MATLAB script or function should not require intrusive changes to be compatible with the MATLAB WPS.

There exist previous approaches to offer data analysis and modeling languages like MATLAB as WPS processes. Specially emphasized should be the *WPS4R* (Hinz et al., 2013)

---

6. http://www.mathworks.com/products/matlab/
7. http://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html
8. http://opensource.org/licenses/bsd-license.php
9. http://www.mathworks.com/matlabcentral/FX_transition_faq.html

project, that creates WPS processes from scripts written for the statistical analysis environment *GNU R* (R Core Team, 2014). Written as module for the 52°North WPS implementation, it shares many requirements and challenges with a MATLAB WPS. R is also a environment used mostly by domain experts and features a massive amount of existing models and algorithm implementations 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 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",
5  #          abstract = "the number to be used for factorization",
   #          value = 1, minOccurs = 0, maxOccurs = 1;

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

WPS4R takes a 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, a annotation mechanism was developed, that 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 1). During process execution, WPS4R will populate the described input parameter variables using WPS inputs, execute the script and will later read the specified output variables from the R session and transform them to WPS outputs. The usage of annotation 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, while complex inputs are transferred as files to a temporary working directory. Complex input and output parameter 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 connection, 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 to e.g. generate documentation (e.g JavaDoc (?)), these often are 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 first can be detected during 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 it own file. By this, MATLAB programs tend to consist of multiple files, and can not easily deployed as single script file.

…

In contrast to R, MATLAB offers multiple return values of functions as a native language feature. 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 parsing of the 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 and 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 in a headless mode (without any user interface) can take considerable time. This requires a effective usage MATLAB instance, especially the reuse of already started MATLAB instances to reduce latency of process executions. Complex inputs should be usable inside of MATLAB without restriction to any format, and without the need to change any configuration files.

... This chapter will outline the conceptualization and implementation of a MATLAB WPS ...

**Listing 2:** MATLAB example function that calculates statistical characteristics (mean and standard deviation) of an input vector.

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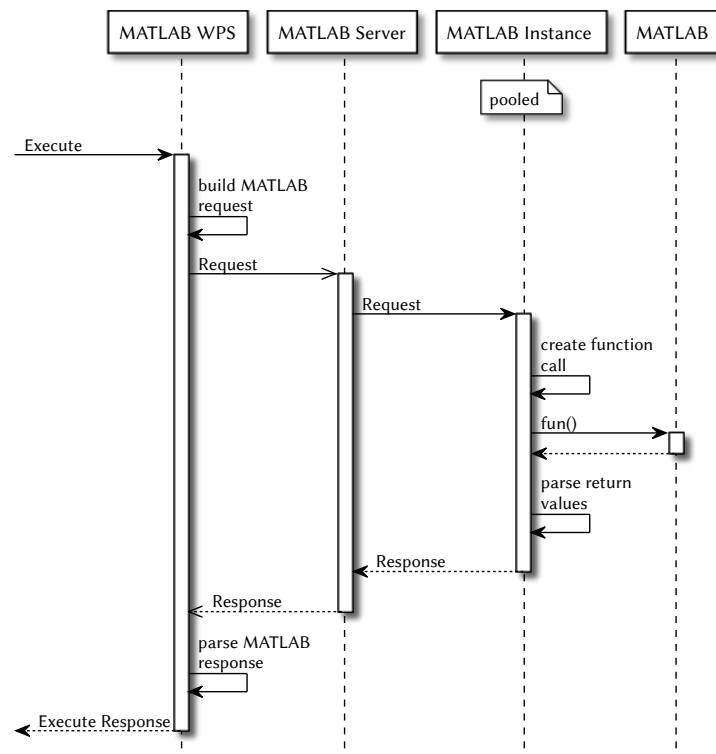


**Figure 2:** Sequence diagram of the Matlab WPS.

- matlab function <-> wps process
- not format specific
- no conversion of complex inputs/outputs

- single output formats
- matlab program has to parse inputs
- easy to publish existing scripts and functions as WPS processes
- multi-tier implementation
  - Matlab WPS
    * Translates WPS Execute requests to Matlab client requests
    * Translates Matlab client responses to WPS Execute responses
    * configuration with YAML file to create description and translate inputs/outputs
  - Matlab Client
    * WebSocket client to access the Matlab server.
    * offers simple request building API
  - Matlab Server
    * WebSocket server that pools multiple Matlab Instances
    * delegates requests to free instances
  - Matlab Instance
    * a Java wrapper around a Matlab instance
  - Matlab
    * A headless instance of the Matlab software

## 4.1. 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 quotations or braces, while optionally allowing these. The structure of YAML has close resemblance with JSON (which is actually a valid subset of YAML since version 1.2), feature the basic types of scalars, sequences and associative arrays (maps), but has 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. Configuration for the MATLAB WPS can contain multiple process configuration expressed as associative array, that 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 shows an example process configuration for the function displayed in Listing 2. The process description generated from the YAML configuration can be found in Listing 4.

**Listing 3:** Matlab process configuration describing the function in Listing 2.

```yaml
---
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.
...
```

Top level attributes are describing the process itself, while *inputs* holds a sequence of input descriptions and *outputs* a sequence of output description 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 to their equivalent in the OGC name space. *maxOccurs* holds either a integral number or the special value *unbounded*, that will be translated to the platform specific maximum possible value (typically the greatest possible integer value) and data types 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*.

The keyword *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 an human readable format, that is way easier to maintain than custom

**Listing 4:** Process description generated from the configuration in Listing 3 (see Appendix B for omitted XML namespaces).

```xml
<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>
```

annotations in in line comments. It results in a less error prone procedure for unexperienced domain experts, while advanced users are able to benefit from advanced YAML features and future enhancements and additions can be easily implemented backwards compatible.

## 4.2. Type Mapping

## 4.3. Pooling

- matlab instances are pooled
- reduced starting time of instances
- limitation of instances

Table 1: Type Mapping between Matlab and WPS Data

| | Data | Matlab Type | |
| --- | --- | --- | --- |
| | | For single inputs | For multiple inputs |
| **Complex** | *any* | String | Cell |
| **Bounding Box** | - | - | - |
| **Literal** | xs:int | Numeric | Array |
| | xs:boolean | Numeric | Array |
| | xs:dateTime | Numeric | Array |
| | xs:double | Numeric | Array |
| | xs:float | Numeric | Array |
| | xs:byte | Numeric | Array |
| | xs:short | Numeric | Array |
| | xs:int | Numeric | Array |
| | xs:long | Numeric | Array |
| | xs:string | String | Cell |
| | xs:anyURI | String | Cell |

## 4.4. License Issues

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

14

*line, or any of the licensed Programs with code generation capabilities. In addi-
tion, 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. Pro-
grams licensed under the Network Named User and Standalone Named User Ac-
tivation 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 Pro-
grams 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 additional authentication mechanism in
place in order to restrict the 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 a open
network, while users of the second class of licenses are still allowed to deploy them with
an additional authentication mechanism. Using a pool of MATLAB instances on a remote
server on the other hand introduce additional problems in regard of the license. In theory
these MATLAB 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
definition, it may be considered a license violation the 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.

## 4.5. Implementation

## 4.6. Lake-Analyzer WPS

## 5. 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 describe a 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, while 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 operation systems, over interprocess communications 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), over general-purpose computing on graphics processing units (GPGPU) to modern media streaming solutions like RTP and RTCP (Schulzrinne et al., 2003), RTSP (Schulzrinne et al., 1998) or SIP (Rosenberg et al., 2002). The concept can be best shown on it's 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 coding/compression format used to encode the media file, it is not possible to play the file until the download is finished. Media streaming reduces time to start playing drastically by sending smaller parts of the media file over the network (e.g. one or more single frames). Suitable player are now able to play this stream of frames long before the complete file is transmitted. Besides th 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/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 programs are able to process very large or infinite datasets, because the complete dataset 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) 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 datasets tend to become rather large and the analysis of real-time data can have great benefits, especially as spatiotemporal data is often an ideal candidate to streaming, as 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 has to especially considered when transferring the concept of streaming to spatiotemporal processing. Algorithms used in streaming are required to operate on smaller chunks of the complete dataset and computations, that require global knowledge are not expected any advantage from streaming. E.g. 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 3 (a)). Characteristic for input streaming (b) 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 (c). Combining these two approaches results in the third category, full input and output streaming, in which input, processing and output take place concurrently. Despite their respective 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 datasets by processing each input data chunk separately and outputting a output data chunk for each of them. Through this the analysis of live sensor data can be accomplished. Each of these categories of processing demands different requirements from the process or algorithm. To create a stream the dataset needs to divisible 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 result in no benefits for algorithms requiring global knowledge of the dataset, because they can not start processing prior to all data chunks have 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.

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
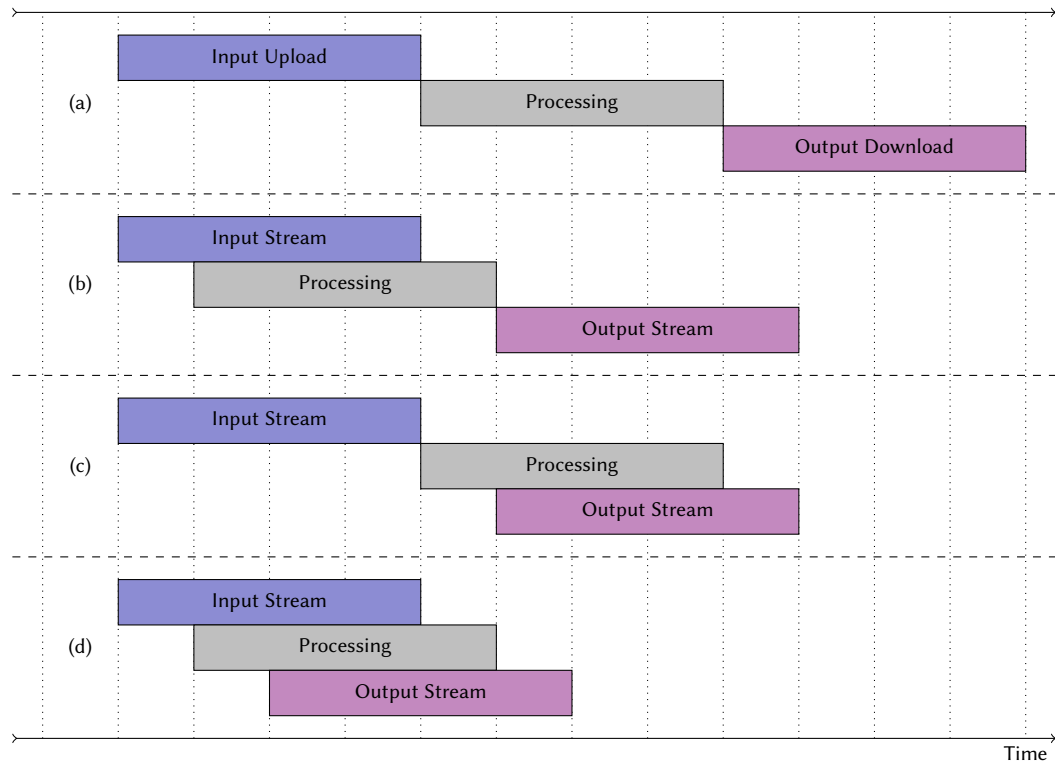
**Figure 3:** Four different types of processing data: (a) conventional processing, (b) streaming input data (c) streaming output data, (d) full input and output streaming (based on Foerster et al., 2012).

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 (Foerster et al., 2012) 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). As the WPS specification is not designed to be extensible, the elements content is restricted to a simple string and can not contain complex Extensible Markup Language (XML) structures. Furthermore the elements definition states, that it should 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.*
>
> − Open Geospatial Consortium (2007)

19

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.
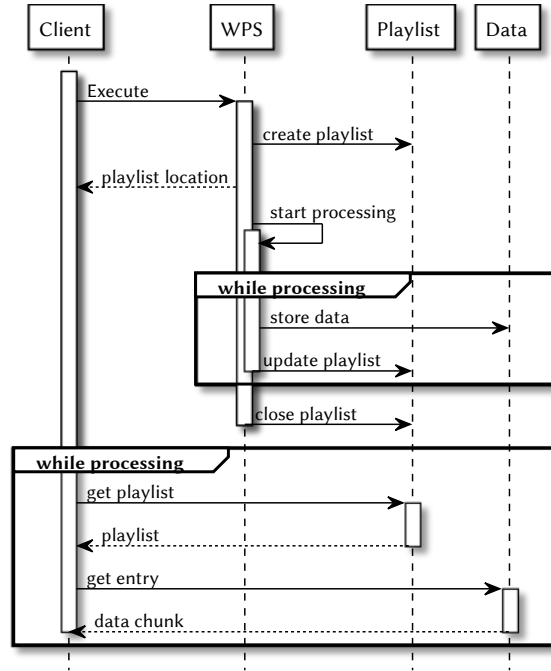


**Figure 4:** Sequence diagram of the playlist-based streaming enabled WPS (Foerster et al., 2012).

Input parameters may also be supplied using a playlist file. The coordination of several streaming inputs is either not possible or heavily dependent on the streaming enabled process. A process accepting two streamed datasets, that are combined during processing, has to decide which data chunks it has to combine and .... Even in the simplest case of combining chunks of both streams with the same index can have serious implications in the use case of live analysis. If a data chunk gets lost, the process, either due to hardware or network failure, the process will combine chunks, that are not related. In continuous process, this error can not be detected, as to indefinite streams of data will always have matching indexes. Use cases in which the rate of incoming data between streams differ or data chunks depend on other chunks are very hard to model and will result in highly specialized processes, that depend not only on the structure and format of input data, but also on the data source, and thus it's incoming rate. 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 other that it was 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. The usage of playlist to transport data from the client to the server, in contrast to transporting data from the server to the client, for which the origin in media streaming playlist was developed, is additionally questionable. Clients need the capability to publish files as resources, that a accessible using a 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 has to store 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 3 (c)). Splitting spatiotemporal data into smaller chunks is not as trivial as e.g. splitting a audio or video stream in single frames and so the process implementations become heavily format dependent and dependencies between data chunk can only expressed as part of the data, 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 3 (a)) to enable input only streaming (Figure 3 (b)), output only streaming (Figure 3 (c)), and full input/output streaming (Figure 3 (d)), for which input parameters are supplied subsequently and output data chunks is published as it becomes available. To accomplish this, it 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 process should be accessible from the same environments as conventional WPS processes. This especially includes web browser environments, that are particular 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 and it's processing and analysis often can not be treated independent to 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 dataset may not profit from a streaming enabled WPS 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 dataset, data chunks should be processed in parallel if possible. As this may result in a undefined order of outputted data chunks, client 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 streaming enabled WPS fulfilling the requirements, that is compatible to the standard. Previous solutions compromised functionality for sake of (incomplete) compatibility with the inflexible standard. In order to enable true, browser compatible, streaming this approach 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 specification.

## 5.1. Protocol

As the the WPS specification is not flexible enough to model a full streaming scenario, the WPS has to be bypassed. For this a more flexible interaction model was developed, that 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. While the execution of the streaming enabled algorithm is fully supported by the WPS specification, 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. This is needed to *push* messages to clients instead of letting the clients constantly request updates.
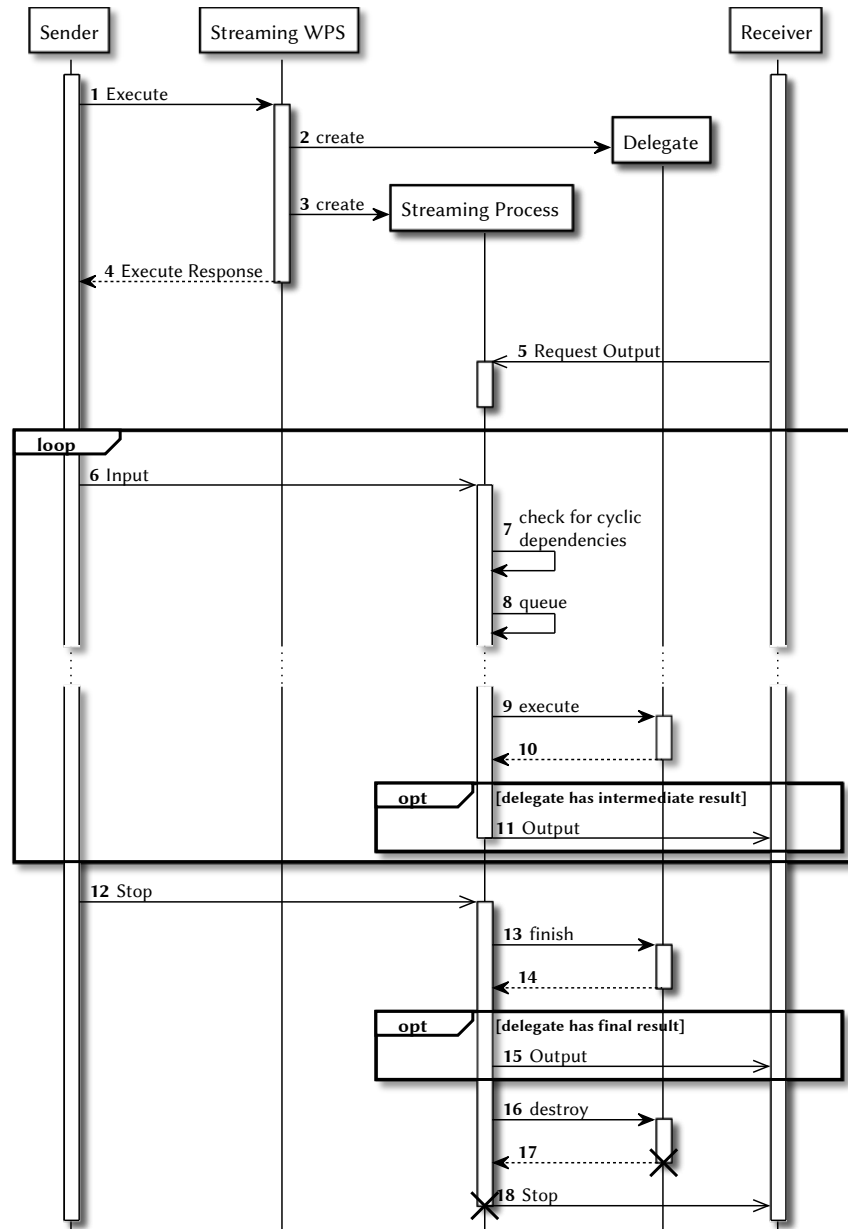


**Figure 5:** Sequence diagram of typical interaction pattern with a streaming enabled WPS algorithm using two distinct clients for sending and receiving data.

The detailed interaction protocol is depicted in Figure 5. A client (*Sender*) issues a Execute to a streaming enabled WPS algorithm (step 1). The algorithm will instantiate a delegate (step 2), that is responsible for processing data chunks, and a streaming process (step 3),

that 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[10] (*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 will receive 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 will check the inputs for validity (step 7) and will queue 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 5.3.3 and 5.4). When there are free capacities to process the data and all other requirements are met, the delegate will be tasked to process the data (step 9). The delegate implementation can return a intermediate result in step 10, which will be 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 for ever (or at least until some timeout interferes), the client has to stop the streaming process explicitly (step 12). This will cause the streaming process to stop accepting inputs, to process all not yet processed inputs and to request a last potential output from the delegate (step 13 & 14), which will be forwarded to all listening clients (hyperref[fig:sd:swps]step 15). After this it will destruct the delegate (steps 16 & 17) and will notify all registered listeners, that there will be no further outputs become available by publish forwarding the stop message (step 18). The streaming process will destroy itself after this.

A detailed description of the various messages of this protocol can be found in section 5.2.

The protocol permits various streaming usage scenarios. A delegate, that produces a output for every input message creates a full input/output streaming process (see Figure 3 (d)), a delegate that produces only a final output results in a input only streaming process (see Figure 3 (b)). By suppling a single input message and repeating step 11, a suitable delegate may create a output streaming process (see Figure 3 (c)) and, although not reasonable, even the traditional processing approach depicted in Figure 3 (a) can be simulated by passing all inputs in a single input message and producing a single output message.

---

10. 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).

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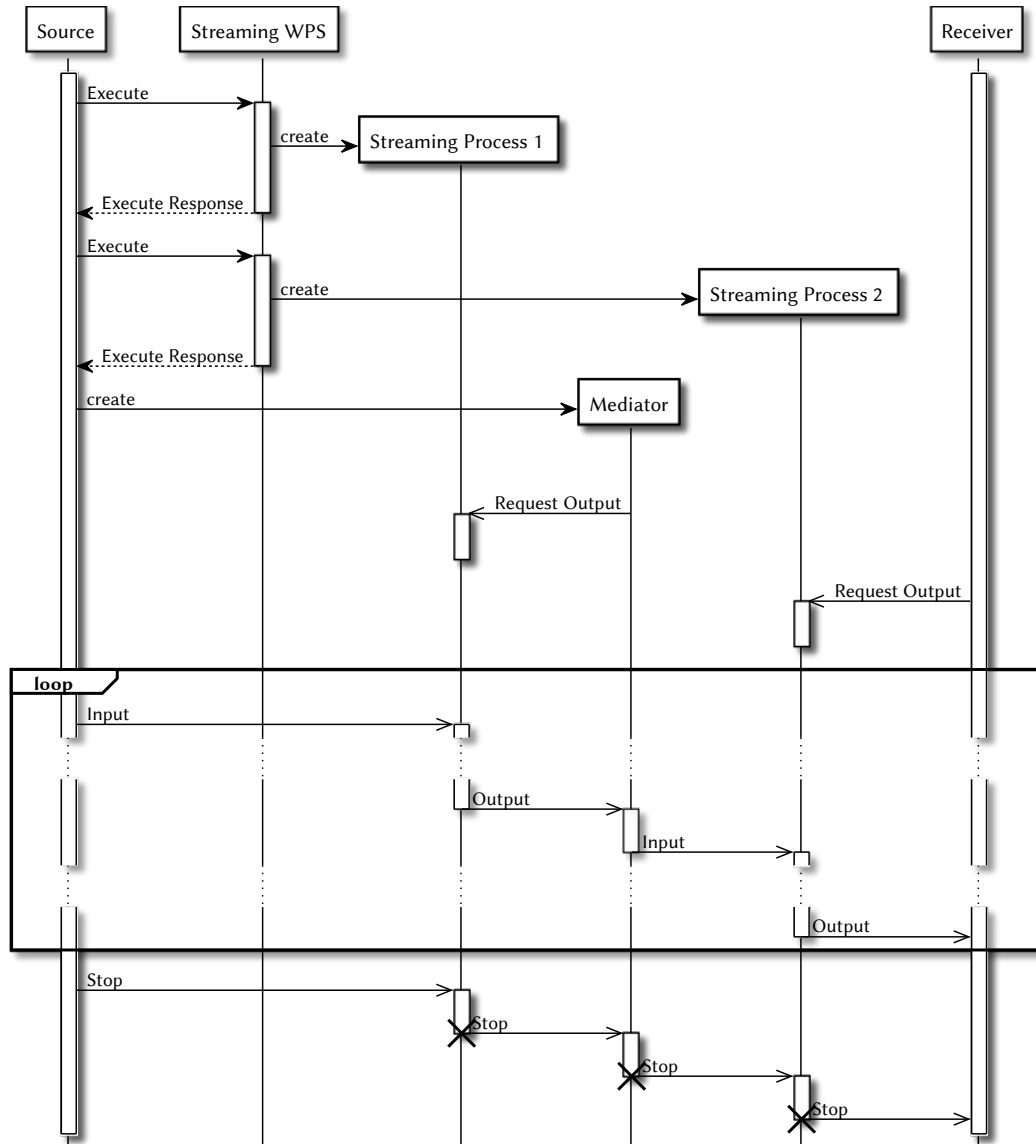


**Figure 6:** Sequence diagram of chaining two streaming processes using a generic mediator between the processes to translate output to input messages.

The protocol also enables the chaining of processing steps. This can be realized in two ways: a delegate itself may represent a WPS process chain and thus chain every processing step or several streaming process are chained itself. A simple mediator translating

input messages to output messages (see Figure 6). This mediator can be realized using a dedicated streaming enabled algorithm accepting a 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 simply connect to the second streaming process and will received the data process by the chain. By requesting the outputs first streaming process even intermediate result of the chain are accessible.

## 5.2. 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, that introduces 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 a 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 and thus can be easily adopted, is a ideal candidate for this. In combination with Web Services Addressing (WSA) (Rogers et al., 2006) it creates a XML based message framework, that allows asynchronous requests and responses over a 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, that can be referenced with arbitrary semantics (e.g. reply).

The Streaming WPS defines seven SOAP messages.

**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 a identifier, which references the streaming process to which the inputs should be supplied. An example can be seen in Listing 5, possible inputs can be seen in section 5.3.

**Listing 5:** Example for a Streaming WPS input message (see Appendix B 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>
5      <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>
10       <stream:Inputs>
           [...]
         </stream:Inputs>
       </stream:InputMessage>
     </soap:Body>
15 </soap:Envelope>
```

**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 a arbitrary number of outputs and the identifier of the process, that produced the outputs. Output messages containing intermediate result 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 5.3.3 and 5.4) the corresponding output messages are also referenced. An example can be seen in Listing 6.

**Output Request Message**   A output request message is used by client to let a streaming process know, that it would like to receive outputs from the 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 a 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 register to one or more streaming processes using their corresponding identifiers. An example can be seen in Listing 7.

**Stop Message**   As streaming process can run indefinitely long, input supplying clients need to be able to let the streaming process know, that there will be no further inputs become available. To achieve this a stop message (see Listing 8) is send to the streaming process. The process will propagate the stop message to all listening clients to let them know there will be no further outputs. Before the stop message is propagated all streaming

**Listing 6:** Example for a Streaming WPS output message (see Appendix B for omitted XML namespaces).

```xml
<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>
```

**Listing 7:** Example for a Streaming WPS output request message (see Appendix B for omitted XML namespaces).

```xml
<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>
```

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 5.3.3 and 5.4) the streaming process will fail with an error message.

**Listing 8:** Example for a Streaming WPS stop message (see Appendix B 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>
 5    </soap:Header>
      <soap:Body>
        <stream:StopMessage>
          <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</stream:ProcessID>
        </stream:StopMessage>
10    </soap:Body>
    </soap:Envelope>
```

**Error Message**   Errors are transported, as in the WPS specification, using OWS exception reports (Open Geospatial Consortium, 2007). 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 send to the client that send 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 a output request message. In contrast to failures during input validation, due to constraints imposed by dependencies (see sections 5.3.3 and 5.4), 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 obtained from the reply relation encoded using WSA. An example of an error message can be found in Listing 9.

**Describe & Description Message**   Describe messages are directly adopted from the WPS Describe Process operation. Due to conditions described in section 5.5 a client needs to able to retrieve a description from a running streaming process. The message simply contains the identifier of the process the clients wants to have the description from (an example can be seen in Listing 10). The reply resembles a Describe Process response and is encoded in a description message referencing the describe message and containing the streaming process description and (see Listing 11).

29

**Listing 9:** Example for a Streaming WPS error message (see Appendix B 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>
5          <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>
10             <ows:Exception exceptionCode="RemoteComputationError">
                   <ows:ExceptionText>Remote computation failed</ows:ExceptionText>
               </ows:Exception>
           </stream:ErrorMessage>
       </soap:Body>
15 </soap:Envelope>
```

**Listing 10:** Example for a Streaming WPS describe message (see Appendix B 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>
5    </soap:Header>
     <soap:Body>
       <stream:DescribeMessage>
         <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</stream:ProcessID>
       </stream:DescribeMessage>
10   </soap:Body>
   </soap:Envelope>
```

## 5.3. 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 section 3).

### 5.3.1. 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 12).

A conventional algorithm to compute the histogram of a raster (e.g. a satellite image) needs

**Listing 11:** Example for a Streaming WPS description message (see Appendix B for omitted XML namespaces).

```xml
<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"
          statusSupported="false" storeSupported="true">
        <ows:Identifier>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</ows:Identifier>
        <ows:Title>com.github.autermann.wps.streaming.example.AddAlgorithm</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>
```

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 it's 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.

31

**Listing 12:** Example for a Streaming WPS streaming inputs (see Appendix B for omitted XML namespaces).

```xml
<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>
```

### 5.3.2. Static Inputs

Algorithms that operate 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 13). 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.

For example, a traditional process implementation of the Douglas–Peucker algorithm (Douglas and Peucker, 1973) would require a feature collection and a $\epsilon$ value as inputs. In a

**Listing 13:** Example for a Streaming WPS static inputs (see Appendix B for omitted
XML namespaces).

```xml
<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>
```

streaming environment, one would model the $\epsilon$ 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 and the feature collection would be split
into several streaming inputs and supplied in independent streaming iterations.

### 5.3.3. 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 dataset or even only about the result of their processing. To model these
dependencies between streaming iterations, reference inputs can be used (see Listing 14).

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 processed (see section 5.4).

**Listing 14:** Example for a Streaming WPS reference input (see Appendix B 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 a river depends on would be supplied as additional reference inputs.

### 5.3.4. 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, that is requested continuously. To not miss inputs, when they become available a playlist file, as described in previous approaches (Foerster et al., 2012) would be need. 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 one can define the polling frequency used to retrieve the playlist, how can multiple polling inputs be declared, and 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 typical knows of the rate data becomes available and so can choose a 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 7: the client polls a data provider (e.g. a Sensor Observation Service (SOS)) to check if new data is available and convert this data into a streaming input for the Streaming WPS.
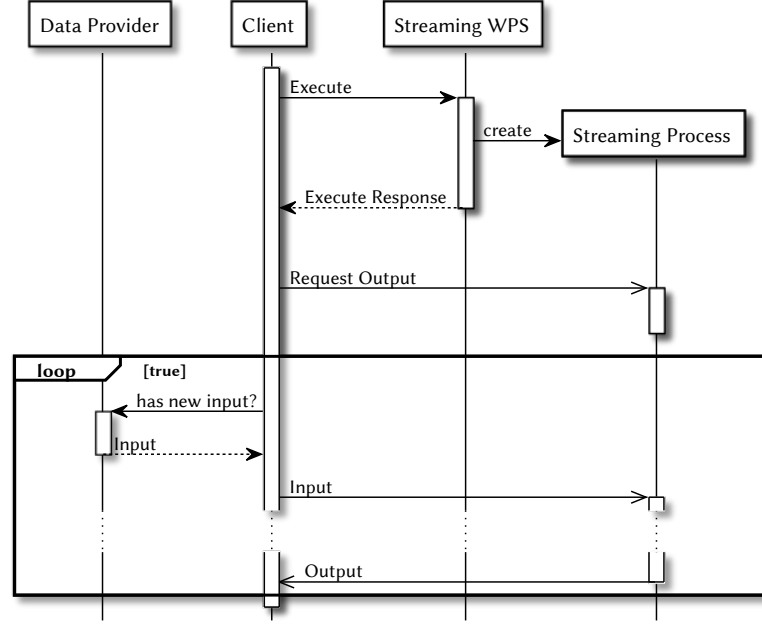


**Figure 7:** Sequence diagram of how to implement polling inputs for a streaming enabled WPS algorithm.
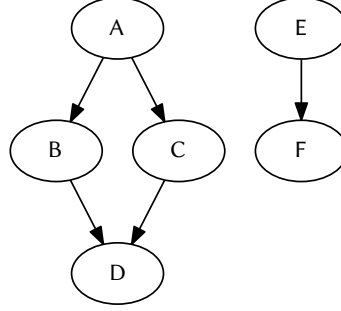
## 5.4. Dependencies

The definition of Reference Inputs in Section 5.3.3 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 a 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 pair $v_1 \rightarrow v_2$ 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 a edge $v_i \rightarrow v_{i+1} \in E$. A directed graph is called acyclic if there exists no path in $G$ with $v_1 = v_n$. A subgraph of a graph is the graph $G' = (V', E')$ with $V' \subseteq V$ and $E' = \{v_1 \rightarrow v_2 \in E | v_1, v_2 \in V'\}$. 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 $v_1 \rightarrow v_2 \in E$ 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 8:** 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 $A \rightarrow B, A \rightarrow C, B \rightarrow D, C \rightarrow D$ and $E \rightarrow F$ will result in a DAG consisting of two independent subgraphs (see Figure 8).

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 $x \rightarrow y \in E$" (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 9 shows one possible execution order for the before mentioned graph.

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 few dynamic topological sort algorithms that will maintain the topological order across edge and node insertions and will only recompute the ordering if necessary.
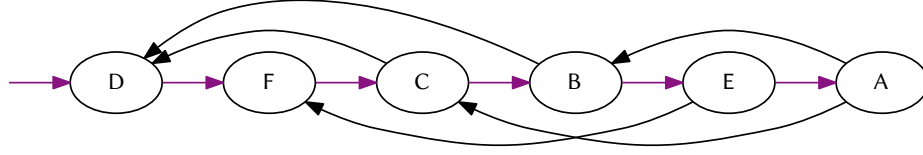
**Figure 9:** Possible execution/topological order of the dependency graph in Figure 8. Black arrows represent dependence to another vertex, colored arrows the execution order.

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 it is theoretically it is inferior to other algorithms for dynamic topological sorting, it especially performs better 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 result 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 the service is not able to complete gracefully and fail. As references to future streaming iterations are allowed, prior to this point, it is not possible for the Streaming WPS to determine if reference may not be fulfilled. As 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 referenceable unit for 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 and streaming process implementations need to be designed to not need smaller components or have to deploy a own referencing strategy (e.g. by additionally 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 now way to reference input parameters of other streaming iterations, but this use case should be already covered by the WPS' own input reference parameters (see section 5.3).

## 5.5. 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, 2007).

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 has 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.

- other process description formats
- differentiation between intermediate results and final result

## 5.6. WPS Specification Shortcomings

- different procedure description format, like in the sensorweb
- process instance need to be identifiable
- WSDL like description language of wps processes
- differentiation between continous outputs and final results
- allow different transport layers (like websockets)
- allowing subsequent input parameters

## 5.7. Implementation

- Server:
    - based on the 52°North WPS
    - includeable module
    - default implementation uses another WPS process as delegate
- Client
    - small JavaScript library
    - abstracts the message generation and WebSocket interaction
    - may be used to start generic delegation processes

## 5.8. Streaming Lake-Analyzer WPS

- simple application of the Streaming WPS and Matlab WPS
- LakeAnalyzer may need further adjustments to allow live analysis
- remove down sampling code
- operate on single point in time
- etc

## 5.9. Limitations

- No input/output conversion
- Only default format is requested from delegate
- process will not fail fast in under every condition
    - inputs first are checked at execution time
- receivers are only provided with upcoming
    - no replay queue

# 6. Future Work

# 7. Conclusion

# References

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

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.

E.W. 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), December 2011. URL http://www.ietf.org/rfc/rfc6455.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.

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.

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

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, April 2007. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

Open Geospatial Consortium. OpenGIS Web Processing Service. Implementation Specification, OGC, June 2007. URL http://portal.opengeospatial.org/files/?artifact_id=24151.

R. Pantos and W. May. HTTP Live Streaming. Internet Draft (Informational), October 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/.

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.

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.

H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 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 (INTERNET STANDARD), July 2003. URL http://www.ietf.org/rfc/rfc3550.txt.

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

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*, March 2003.

# A. Source Code

| | |
|---|---|
| **Streaming WPS** | Extension for the 52°North WPS to allow of Inputs and Outputs over WebSockets. |
| | 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. |
| | https://github.com/autermann/matlab-wps |
| **streaming-wps-js** | Streaming WPS JavaScript Bindings |
| | https://github.com/autermann/streaming-wps-js |
| **WPS Commons** | 52°North WPS convenience classes and bootstrapping code. |
| | https://github.com/autermann/wps-commons |
| **Matlab Connector** | Matlab function execution on (pooled) remote Matlab instances. |
| | https://github.com/autermann/matlab-connector |
| **Lake-Analyzer** | Matlab source code for Lake Analyzer |
| | https://github.com/autermann/Lake-Analyzer |
| **YAML API** | A Jackson-like API to read and create YAML nodes (based on SnakeYAML). |
| | https://github.com/autermann/yaml |

# B. 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 |

## Plagiatserklärung des Studierenden

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 5. 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 5. Mai 2014    _____