Master Thesis

Streaming Web-Services for Calculating Live Hydrological Derivatives

Christian Autermann
autermann@uni-muenster.de
Institute for Geoinformatics
University of Münster

May 5, 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

2. Lake-Analyzer

3. Foundations

4. Matlab WPS

- weakly typed language
- functions with multiple return values
- previous approaches: WPS4R
  - heavily format specific
    - parsing of GML/etc in the WPS and translation to R structures
    - configuration as comments in R scripts
    - focussing on scripts and not on functions

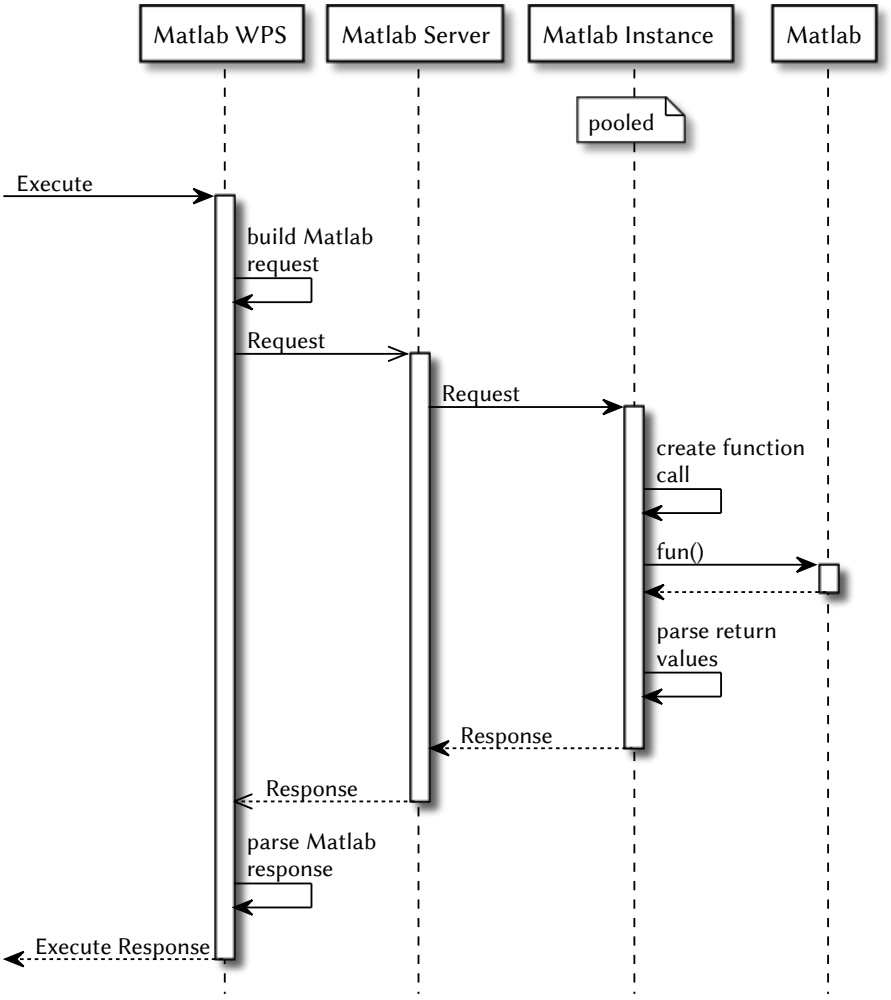

Figure 1: Sequence diagram of the Matlab WPS.

- 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

Listing 1: Matlab example function that represents a simple addition.

```
function result = add(a, b)
  result = a + b
end
```

- Can not be used to offer any function as process
- would not conform to Mathworks license
- configuring of a single function as a process
- configuration YAML file

Listing 2: Matlab process configuration describing the function in Listing 1.

```yaml
---
function: add
connection: local
identifier: matlab.add
version: 1.0.0
inputs:
  - identifier: a
    type: double
  - identifier: b
    type: double
outputs:
  - identifier: result
    type: double
...
```

Listing 3: Process description generated from the configuration in Listing 2 (see Appendix B for omitted XML namespaces).

```xml
<ProcessDescription wps:processVersion="1.0.0">
  <ows:Identifier>matlab.add</ows:Identifier>
  <ows:Title>matlab.add</ows:Title>
  <DataInputs>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>a</ows:Identifier>
      <ows:Title>a</ows:Title>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
    <Input minOccurs="1" maxOccurs="1">
      <ows:Identifier>b</ows:Identifier>
      <ows:Title>b</ows:Title>
      <LiteralData>
        <ows:DataType ows:reference="xs:double"/>
        <ows:AnyValue/>
      </LiteralData>
    </Input>
  </DataInputs>
  <ProcessOutputs>
    <Output>
      <ows:Identifier>result</ows:Identifier>
      <ows:Title>result</ows:Title>
      <LiteralOutput>
        <ows:DataType ows:reference="xs:double"/>
      </LiteralOutput>
    </Output>
  </ProcessOutputs>
</ProcessDescription>
```

4.2. Type Mapping

Table 1: Type Mapping between Matlab and WPS Data

		Matlab Type	
	Data	**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

4.3. Pooling

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

4.4. License Issues

> *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;*
>
> — The MathWorks, Inc. Software License Agreement

> *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.

— The MathWorks, Inc. Software License Agreement - Deployment Addendum

4.5. Implementation

4.6. Lake-Analyzer WPS

5. Streaming WPS

- the concept of streaming describes the sequential processing of data in contrast to random access processing
- processing takes place on small chunks instead of the complete dataset
- reduced processing resources needed to process smaller chunks
- reduced latency to see the output
- enables processing of indefinite large datasets (e.g. live analysis)
- widely known:
  - media streaming (live/on-demand) video/audio streaming
    - RTP and RTCP (Schulzrinne et al., 2003), RTSP (Schulzrinne et al., 1998), SIP (Rosenberg et al., 2002)
  - inter process communication
    - pipes/sockets (local or network) (Buschmann et al., 1996)

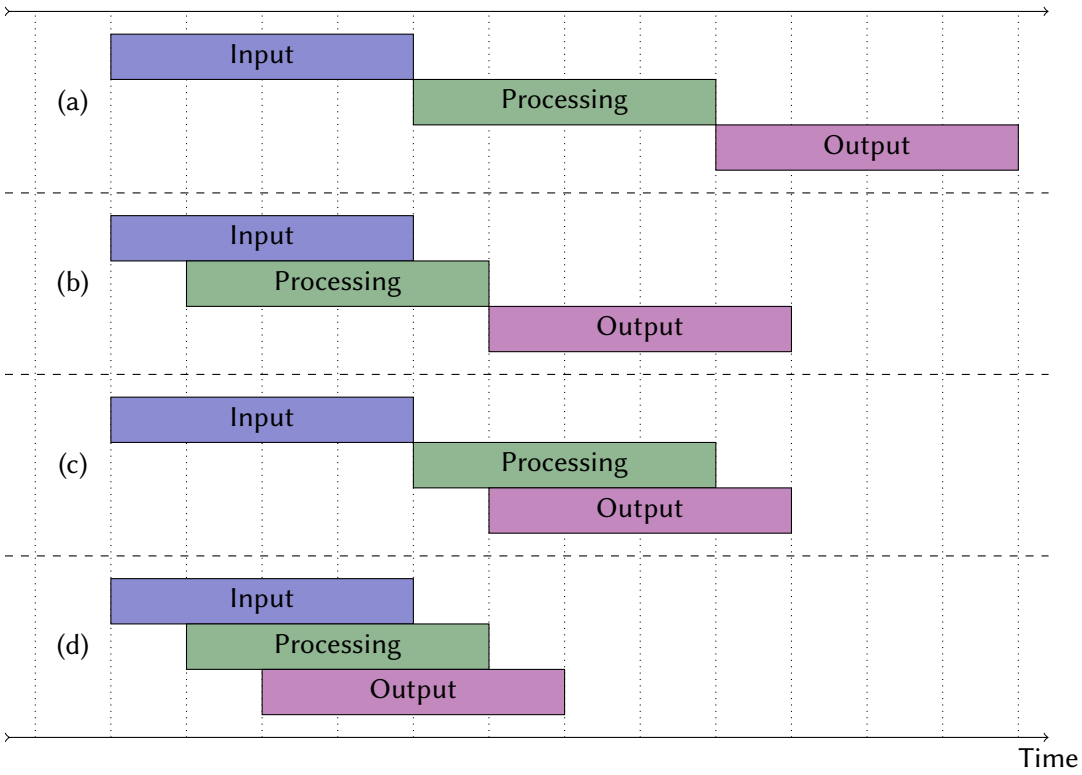

Figure 2: Four different types of processing data: (a) conventional processing, (b) streaming input data (c) streaming output data, (d) full input and output streaming (based on Foerster et al., 2012).

- the system should extends the traditional processing paradigm (see Figure 2 (a))…
  - …to enable input only streaming (see Figure 2 (b))
    - input should be supplied subsequently
  - …to enable output only streaming (see Figure 2 (c))
    - intermediate outputs should be published as they come available
  - …to enable full input and output streaming (see Figure 2 (c))
    - input should be supplied subsequently
    - intermediate outputs should be published as they come available
- Many processes accept inputs that are aggregates of smaller inputs (such as rasters and tiles, feature collections and features, etc.). Often these inputs are processed separately
- it should…
  - …not rely on inefficient polling techniques
  - …be deployable in a web browser environment
  - …should rely on open and widely used standards
  - …be as inter operable as possible with the existing WPS standard
  - …allow not only sequential analysis but should also take dependencies between spatio-temporal features into account
  - …be not dependent on the data format
  - …should allow live analysis of data
  - …should allow analysis of great data sets
  - …should allow chaining
  - …should allow to easily transform existing WPS processes into streaming processes
  - …should process data chunks in parallel if possible while maintaining provenance
- previous approaches (Foerster et al., 2012)
  - in strong correlation to media streaming (Pantos and May, 2013)
  - publishing data chunks in playlists
  - client/wps polling playlist and fetches data chunks when they become available
  - big overhead of continuous fetching (in what frequency?)
  - asynchronous WPS Execute
  - output playlist is transported by wps:ProcessStarted: "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."

  - WPS standard highly constraining
  - approach still stick to it for the sake of interoperability
- previous approach is highly limited
  - implementation only supports output streaming (2) (c))
  - WPS/algorithm is splitting outputs ⇒ highly format specific
  - splitting of complex data is often a complex procedure that can not be automated
  - each data items context important
  - dependencies between data chunks can not be considered
  - automatic splitting of e.g. features in a Feature Collection is highly format dependent
  - browser based clients can not use streaming inputs
  - they can not offer a file under a URL
  - multiple outputs to stream?
  - how to correlate/connect/coordinate multiple streamed inputs?
- this approach…
  - will fulfill all above mentioned requirements
  - break out of the constraints imposed by the WPS standard
  - while reusing terminology and technology of the WPS standard
  - use modern web browser compatible technologies
- create a messaging based architecture
- use WebSockets to accomplish true full-duplex streaming of data
- WPS is highly XML based: use widely known SOAP+WSA on top of WebSockets

5.1. Protocol

As the the Web Processing Service (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 spatio-temporal 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 pro-

cess, 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.

The detailed interaction protocol is depicted in Figure 3. 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[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 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 the are free capacities to process the data and all other requirements are met, the delegate will 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 (step 15). After this it will destruct the delegate (steps

[1]Even though sender and receiver are 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).

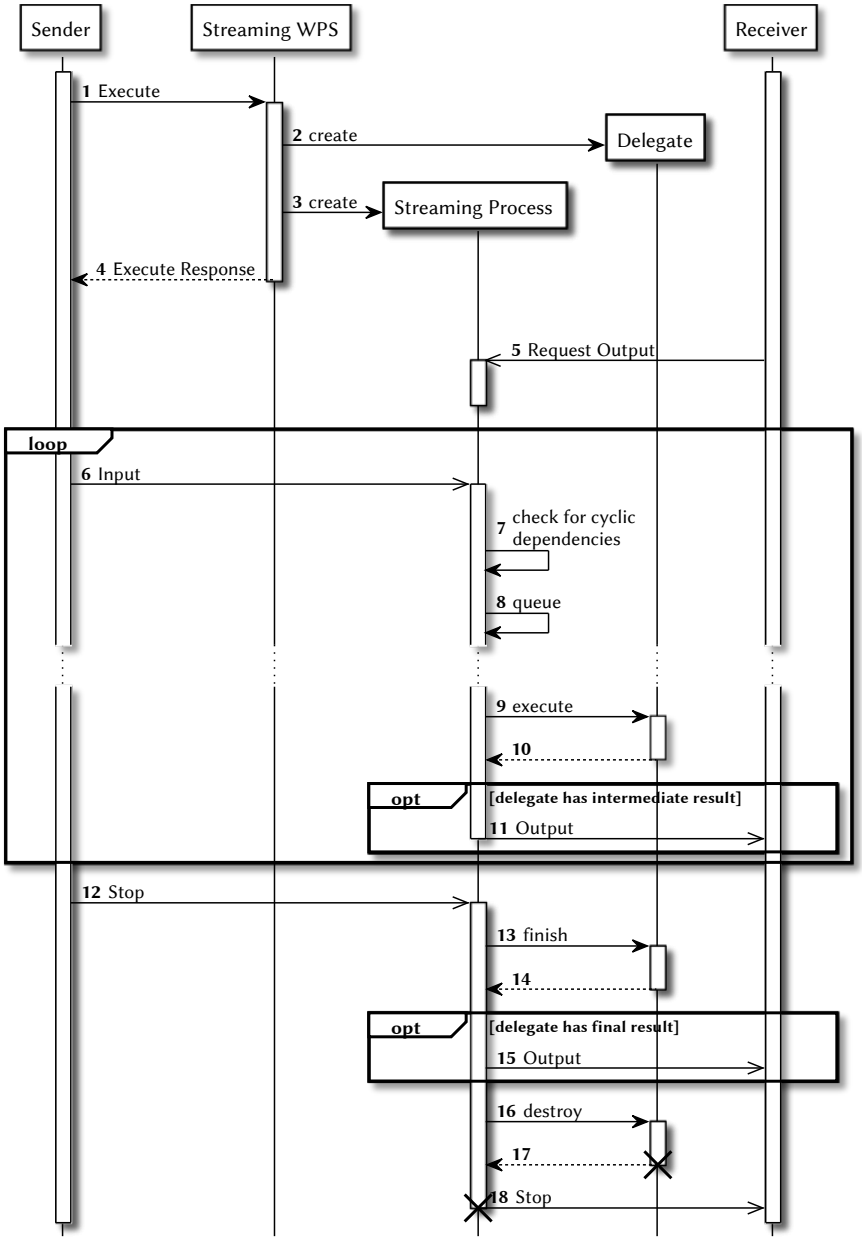

Figure 3: Sequence diagram of the Streaming WPS.

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 2 (d)), a delegate that produces only a final output results in a input only streaming process (see Figure 2 (b)). By suppling a single input message and repeating step 11, a suitable delegate may create a output streaming process (see Figure 2 (c)) and, although not reasonable, even the traditional processing approach depicted in Figure 2 (a) 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 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.

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 4). 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,

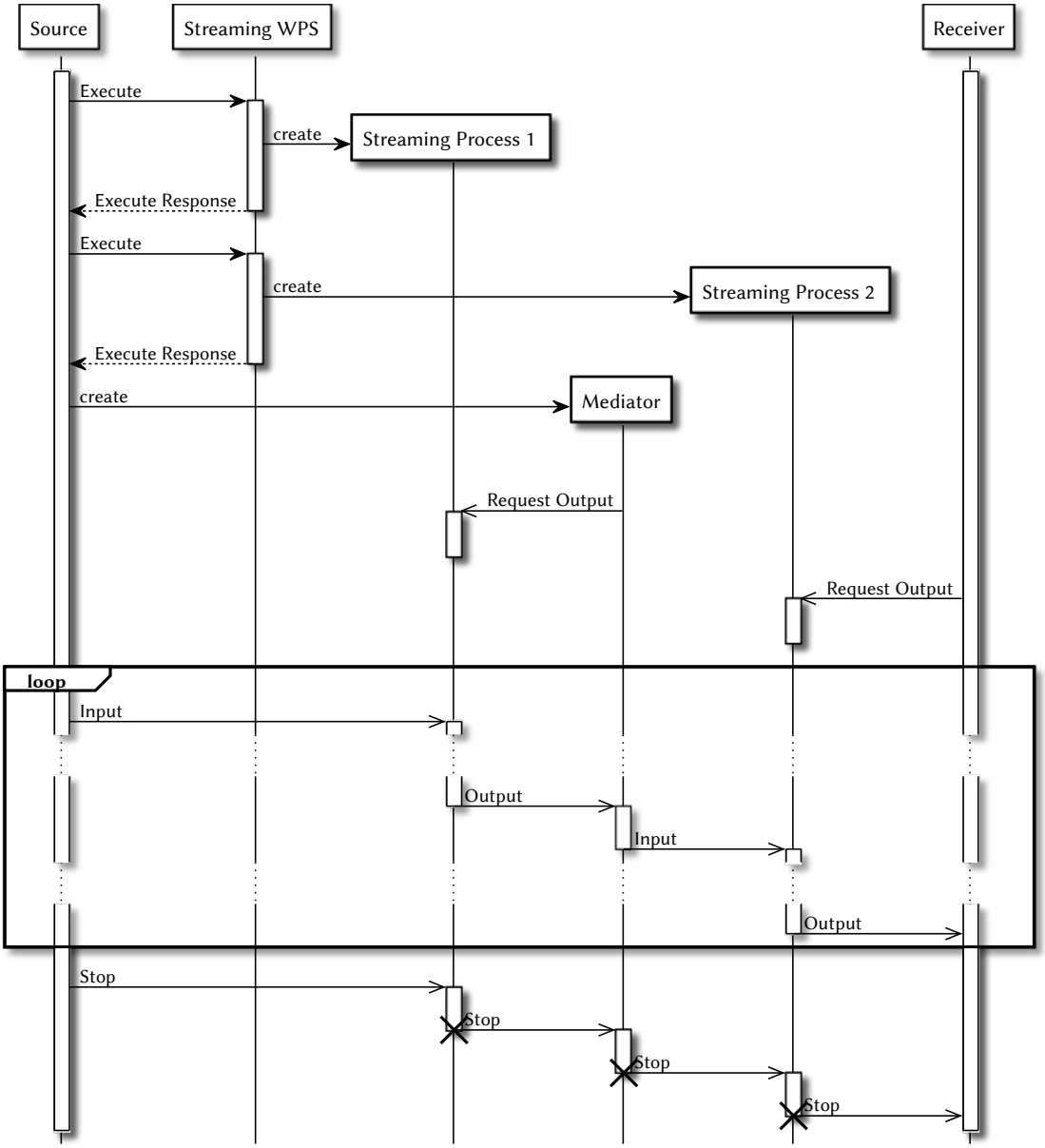

Figure 4: Sequence diagram of chaining two different streaming processes.

another layer is needed. As the WPS is mostly based on Extensible Markup Language (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 combina-

tion 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 4, possible inputs can be seen in section 5.3.

Listing 4: 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>
    <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>
```

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 5.

Listing 5: Example for a Streaming WPS output message (see Appendix B 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>
```

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 mes-

saging 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 6.

Listing 6: Example for a Streaming WPS output request message (see Appendix B 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>
```

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 7) 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 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.

Error Message Errors are transported, as in the WPS specification, using OGC Web Services Common (OWS) exception reports (Open Geospatial Consortium, 2007b). 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

Listing 7: 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>
  </soap:Header>
  <soap:Body>
    <stream:StopMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</stream:ProcessID>
    </stream:StopMessage>
  </soap:Body>
</soap:Envelope>
```

message of a failure may obtained from the reply relation encoded using WSA. An example of an error message can be found in Listing 8.

Listing 8: 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>
        <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>
```

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 9). 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 10).

Listing 9: 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>
  </soap:Header>
  <soap:Body>
    <stream:DescribeMessage>
      <stream:ProcessID>uuid:c99e6f21-f0a0-4770-9615-db3501490f0a</stream:ProcessID>
    </stream:DescribeMessage>
  </soap:Body>
</soap:Envelope>
```

5.3. Input Types

The before mentioned 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:

Complex Input Complex 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 type of data. This data can be supplied inline or as reference to an external HTTP resource.

Literal Input Data that can be represented by a single string value and can be described by data type and a unit of measurement.

Bounding Box Input Data that represents a multi dimensional bounding box with a associated coordinate reference system.

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 and are the core of a streaming

Listing 10: Example for a Streaming WPS description message (see Appendix B 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"
          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>
```

enabled process (see Listing 11).

A traditional algorithm to compute the histogram of a raster (e.g. a satellite image) would need 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 in a single input message to

Listing 11: Example for a Streaming WPS streaming inputs (see Appendix B 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>
```

the streaming process. The process can process each tile on it's own and update the global histogram. Besides that the process never has to store the complete raster, it is also able to output intermediate histograms to the client.

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 has to be introduced. Static inputs are parameters that are supplied when a streaming process is

created and apply to every streaming iteration (see Listing 12). While the streaming process processes 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.

Listing 12: Example for a Streaming WPS static inputs (see Appendix B 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 coordinate reference system (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.

Listing 13: 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>
```

- see Listing 13
- references the output of a previous or upcoming streaming iteration as an input for this iteration
- used to model dependencies between iterations/features/etc.
- breaks out of the classical non-random access paradigm of streaming
- example: analyzing a river system where each processing of a river depends on results of rivers flowing into it
  - conventional: the complete river system is a single input
  - streaming: each river is pushed separately referencing the output of the rivers it depends on

5.3.4. Polling inputs

- Not implemented inside the streaming WPS.
- what to do if multiple polling inputs are defined?
- how to combine them?

- how to define polling frequency?
- how to define notifications?
- better handled on client side (see Figure 5) and transformed to streaming inputs

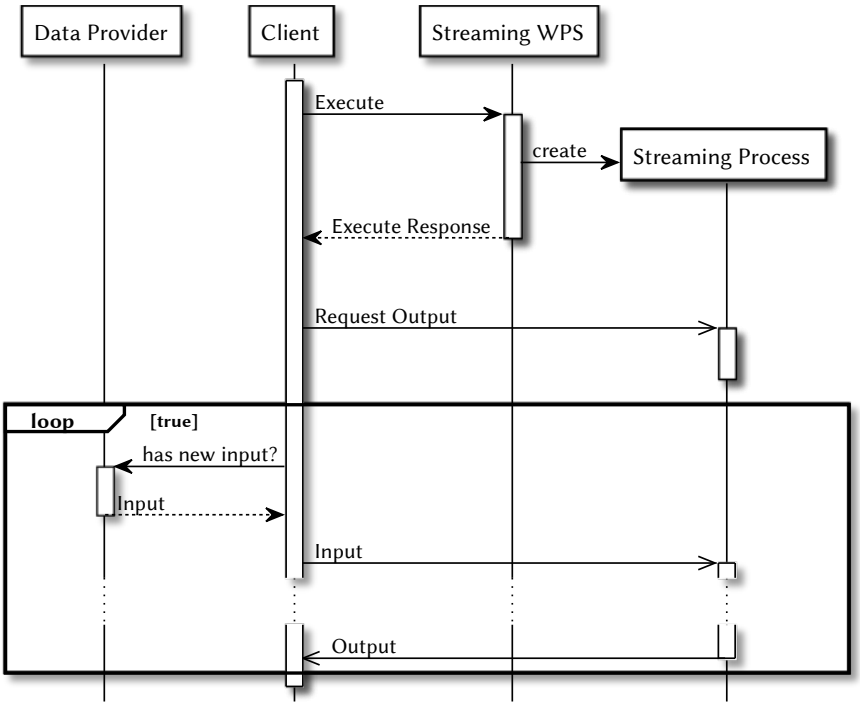

Figure 5: Sequence diagram of polling inputs of the Streaming WPS.

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.

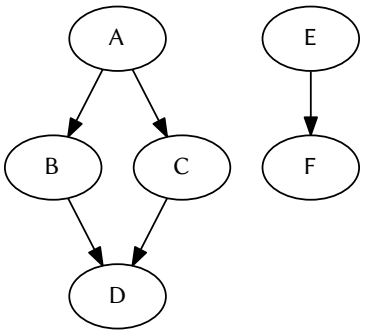

Figure 6: Example for a dependency graph consisting of two independent subgraphs.

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 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 $x \rightarrow y \in E$" (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 7 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

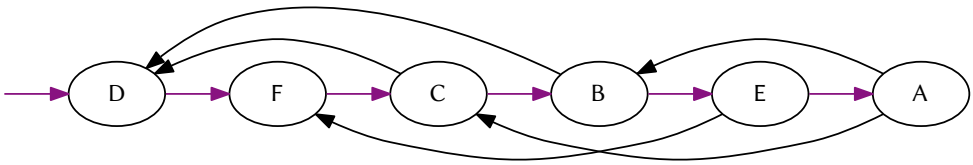

Figure 7: Possible execution/topological order of the dependency graph in Figure 6. Black arrows represent dependence to another vertex, colored arrows the execution order.

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.

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).

The actual implementation uses a DAG only for a cyclic dependency check. Execution ordering is listener based to allow a better parallelization of streaming iterations.

- missing inputs at process stop -> failure
- execution failed -> process stop (dependent iterations may be affected)

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, 2007b).

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.

5.6. Stateful vs. Stateless

- stateful: iterative convexhull
- stateless: every delegating

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

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.

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.

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.

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. OGC Web Services Common Specification. Implementation Specification, OGC, February 2007a. URL http://portal.opengeospatial.org/files/?artifact_id=20040.

Open Geospatial Consortium. OpenGIS Web Processing Service. Implementation Specification, OGC, June 2007b. 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.

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.

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.

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.

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 _____