Application Integration and Provisioning

Authors
Rainer Schmidt, Matthias Rella (AIT Austrian Institute of Technology)

July 2014

This work was partially supported by the SCAPE Project. The SCAPE project is co-funded by the European Union under FP7 ICT-2009.4.1 (Grant Agreement number 270137).

Table of Contents

1 Introduction 4
2 Tool Specification 4
2.1 Application Shipping 4
2.2 SCAPE Tool Specification Language 5
2.3 Basic Elements of a Tool Specification Document 5
2.4 Examples 6
2.4.1 Single Tool Specification 6
2.4.2 Ad-Hock Script Specification 7
2.4.3 Java-based Invocation (JVM Reuse) 8
3 Command Specification 9
3.1 Text-based Input 9
3.2 Control File Specification 9
3.2.1 Basic command specification 9
3.2.2 File Redirection (Streaming) 9
3.2.3 Joining Multiple Operations with Pipes 10
3.3 Example Control File 10
4 The MapReduce Application 11
4.1 Configuration and Submission 11
4.2 Data Decomposition 11
4.3 Handling Application IO 11
5 Using ToMaR on the Pig Platform 12
5.1 Motivation 12
5.2 Implementation 12
5.3 Application 12
6 Fine-tuning and Optimizations 13
6.1 Chained Operations 13
6.2 Tool-specific Optimizations 13
6.3 Exploiting Data Locality 14
6.4 Application Shipping and VM Reuse 14
7 Links and Further Reading 15
8 Conclusion 15
9 References 15

1 Introduction

The SCAPE project deals with the development of scalable strategies for processing large volumes of content with a focus on long-term preservation. The project is evaluating its results in different testbeds against data originating from different communities. This includes data originating from scientific facilities, computer simulations, audio-visual content, web archives, or large digitization projects. Work in SCAPE focuses on various aspects of digital preservation including decision support [1], monitoring and semantic knowledge extraction [2], as well as the implementation of preservation strategies as reproducible workflows. The SCAPE Preservation Platform [3], developed in this context, provides an infrastructure allowing users to enact these workflows at scale using a data-centric architecture. This includes for example support for executing legacy applications, metadata extraction, automated database building and digital repository integration.

The employment of data-intensive technologies like those provided by the Apache Hadoop[1] eco-system have been designed to process data sets at a very large scale. While this technology shift has become natural for many Big Data domains, it can be challenging to apply for traditional content holders. Scalable environments like Hadoop require the user to conform to the MapReduce programming model, take advantage of a higher-level language (like Pig or Hive), or make use of utilities like the Hadoop streaming API. While these methods provide key abstractions for processing large-scale data sets, they are not directly applicable to the processing of binary content.

Preservation actions like file format identification, format validation, metadata extraction, format migration, or quality assurance require the use of legacy applications. Re-implementing these applications for a particular execution environment like MapReduce is often not feasible. ToMaR efficiently supports the integration of such legacy tools for massively parallel environments such as a MapReduce cluster and has been used with various applications to define scalable (MapReduce-based) workflows in the context of the SCAPE Testbeds. In the following, we describe the core concepts of ToMaR and explain how the application can is used. ToMaR is publicly available through its code repository[2] on Github.

2 Tool Specification

2.1 Application Shipping

ToMaR does not handle the shipping of legacy applications to the computational nodes of a cluster. Instead, it assumes that the required applications are made available via the Hadoop framework or the node's operating system. Hadoop provides built-in support for shipping Java-based applications to the computational nodes via its distributed cache. However, many use-cases found in the content management domain require the employment of applications that rely on software that must be pre-installed on the operating system. With respect to the SCAPE platform, it is assumed that the infrastructure is maintained by a content holding institution and dedicated to preserve content of a certain scope, enabling one to maintain installations of the required preservation tools directly on the cluster nodes. Most platform deployments that have been set-up in the context of the SCAPE

1 http://hadoop.apache.org/

2 https://github.com/openplanets/tomar

project make use of Linux package management systems in order to control the installed software packages.

2.2 SCAPE Tool Specification Language

ToMaR relies on the SCAPE Tool Specification Language which provides a simple XML schema[3] to formalize the usage of different software packages (tools) by specifying properties like API calls, configuration parameters, or defining how IO is handled. Tool specification documents (called toolspecs) are XML documents that define a set of operations that can be carried out by the *tool* it defines. Tool specification documents can describe general patterns for using a single software package, or define new operations, e.g. for a complex command-line invocation. The operations defined in the toolspec documents specify atomic operations that can be carried out at scale using ToMaR. Tool specification documents also provide a simple and flexible mechanism to define tool dependencies, in particular for workflows. Tool specification documents are intended to be minimalistic so that they can be easily created for individual tools and scripts.

2.3 Basic Elements of a Tool Specification Document

The **<Description>** element is used to provide a short textual description of the tool that is defined by the tool specification document.

The **<License>** element provides information about the software licence under which the tool is distributed.

The **<Installation>** element specifies the operating systems a tool supports and also carries information about the software package and a repository which can be used by a package manager like apt or yum to obtain and install the required software.

The **<Operations>** element defines one or more atomic operations that can be invoked by a generic tool wrapper like ToMaR. Each operation consists of a textual description, a definition of the command to invoke, as well as a definition of how input and output data is handled.

The **<Command>** element defines an executable command that can be called from an automated tool executor. Although we have concentrated on defining commands based on the command-line interface, API calls might also be specified, if supported by the executor. A command may also be formulated as a script comprising of multiple underlying applications. Undefined values like file names can be specified as parameters.

The **<Inputs>** element defines the input data consumed by the defined operation. This includes parameters defining file/directory names as well as a Boolean value defining if data is supplied using the standard input stream (stdin).

The **<Outputs>** element defines the output data produced by the defined operation. This includes file parameters defining file/directory names as well as a Boolean value defining if data is produced using the standard output stream (stdout).

[3] available at: https://github.com/openplanets/scape-toolspecs

2.4 Examples

2.4.1 Single Tool Specification

```
<?xml version="1.0" encoding="utf-8" ?>
<tool xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://scape-project.eu/tool tool-1.0_draft.xsd"
      xmlns="http://scape-project.eu/tool"
      xmlns:xlink="http://www.w3.org/1999/xlink" schemaVersion="1.0" name="ghostscript">
  <operations>
    <operation name="ps2pdf">
      <description>Converts ps to pdfa files</description>
      <command>/usr/bin/gs -dPDFA -dBATCH -dNOPAUSE
                           -sDEVICE=pdfwrite -sOutputFile=${output} ${input}</command>
      <inputs>
        <input name="input" required="true">
          <description>Reference to input file</description>
        </input>
      </inputs>
      <outputs>
        <output name="output" required="true">
          <description>Reference to output file</description>
        </output>
      </outputs>
    </operation>
    <operation name="s-ps2pdf">
      <description>Converts ps to pdf files.IO handled via stdin/stdout</description>
      <command>/usr/bin/ps2pdf - -</command>
      <inputs>
        <stdin>true</stdin>
      </inputs>
      <outputs>
        <stdout>true</stdout>
      </outputs>
    </operation>
  </operations>
</tool>
```

Figure 1: A tool specification document defining two operations for the Ghostscript command-line application. The operation “ps2pdf” defines a command-line to convert PostScript documents to the PDF/a format based on input/output files. The operation “s-ps2pdfs” defines a command-line to convert PostScript documents to PDF using stdin / stdout handle IO streaming.

2.4.2 Ad-Hock Script Specification

```
<?xml version="1.0" encoding="utf-8" ?>
<tool xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://scape-project.eu/tool tool-1.0_draft.xsd"
      xmlns="http://scape-project.eu/tool"
      xmlns:xlink="http://www.w3.org/1999/xlink" schemaVersion="1.0" name="bash">
  <operations>
    <operation name="isHTML">
      <description>Uses file utility to identify HTML files</description>
      <command>if [ "$(file ${input} | awk "{print \$2}" )" == HTML ];
                then echo "HTML" ; fi</command>
      <inputs>
        <input name="input" required="true">
          <description>Reference to input file</description>
        </input>
      </inputs>
    </operation>
    <operation name="sisHTML">
      <description>Uses file utility to identify HTML files.
        Input is read from stdin stream</description>
      <command>if [ "$( file - | awk "{print \$2}" )" == HTML ];
                then echo "HTML" ; fi</command>
      <inputs>
        <stdin>true</stdin>
      </inputs>
    </operation>
  </operations>
</tool>
```

Figure 2: A tool specification document which implements two more complex operations that make use of Linux shell commands. Both operations identify if the mime type of a file equals text/html and write the result to standard out. The operation *"isHTML"* takes a file pointer as input; the operation "sisHTML" reads the input from the standard input stream.

2.4.3 Java-based Invocation (JVM Reuse)

```
<?xml version="1.0" encoding="utf-8" ?>
<tool xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://scape-project.eu/tool
    https://raw2.github.com/openplanets/scape-toolspecs/master/toolspec.xsd"
    xmlns="http://scape-project.eu/tool"
    xmlns:xlink="http://www.w3.org/1999/xlink" schemaVersion="1.0" name="fits" version="1.0.1"
    homepage="http://bla.org/">
    <operations>
        <operation name="identify">
            <description>Identifies a file</description>
            <command>/root/fits-0.6.1/fits.sh -i ${input} -o ${output}</command>
            <inputs>
                <input name="input" required="true">
                    <description>Reference to input file or directory</description>
                </input>
            </inputs>
            <outputs>
                <output name="output" required="true">
                    <description>Reference to output file or directory</description>
                </output>
            </outputs>
        </operation>
        <operation name="j-identify">
        <description>Fits</description>
        <command>/usr/bin/java edu.harvard.hul.ois.fits.Fits -i ${input} -o ${output}</command>
        <inputs>
            <input name="input" required="true">
                <description>Reference to input file</description>
            </input>
        </inputs>
        <outputs>
            <output name="output" required="true">
                <description>Reference to zipped output file</description>
            </output>
        </outputs>
    </operation>
    </operations>
</tool>
```

Figure 3: A tool specification for a Java application. The operation *"identify"* calls the application via its command-line interface implemented as a shell script. The operation *"j-identify"* invokes the same interface explicitly through the Java application launcher. This indicator enables ToMaR to significantly reduce execution overheads by employing an optimized invocation method based on direct API calls and reusing the JVM.

3 Command Specification

3.1 Text-based Input

If running as a standalone application, ToMaR takes a plain text file as input (called the control file) specifying how one or multiple joined toolspec operations are applied to the payload data. Each line in the input file (called a *control line*) typically applies the specified command to a particular data item (e.g. a file). Hence, a user would generate a control file with *n* lines in order to process *n* input files with ToMaR. The control file is broken into splits at execution time and processed independently by distributed mapper processes. When applied together with a higher-level platform like Apache Pig, the input for ToMaR can be dynamically generated and it is no longer required to create a control file before execution time. However, the syntax for specifying the commands that will be executed by ToMaR in a cluster is the same, regardless if the control lines are generated statically or on-the-fly. This section describes the command specification presently supported by ToMaR. For more detailed information the reader is referred to the ToMaR documentation on Github[4].

3.2 Control File Specification

3.2.1 Basic command specification

The control line to execute a basic toolspec command is provided in the example below. A control line comprises a number of elements which are separated by whitespace characters. A control line is terminated by a newline character.

Example:

```
fits identify --input="hdfs:///user/bob/1.tiff" --output="hdfs:///user/bob/1.tiff.xml"
```

The first element "*fits*" specifies the name of the corresponding tool specification document (compare section 3). ToMaR makes use of a simple file repository to obtain a list of known tool specification documents. The next element "*identify*" specifies the operation which should be invoked, as defined in the toolspec document. The following elements starting with "--" are parameters (here "*input*" and "*output*") that must be assigned concrete values in the control file.

3.2.2 File Redirection (Streaming)

The tool specification language supports the definition of commands that read from standard input and/or write to standard output. A control line that invokes an operation that streams data from/to a file on HDFS is shown in the example below.

Example:

```
"hdfs:///user/bob/1.ps" > ghostscript s-ps2pdf > "hdfs:///user/bob/1.pdf"
```

File IO using streams is indicated using the ">" character. Compared to file-based IO, this method allows ToMaR to transfer payload data from/to an application (chain) without having to generate temporary files on the local file system.

[4] https://github.com/openplanets/tomar

3.2.3 Joining Multiple Operations with Pipes

Instead of streaming a command's output to a file, the output stream can also be redirected to another toolspec command, similar to using pipes in the UNIX shell. Compared to performing these tasks separately, the mechanism allows one to create chained operations which are handled by ToMaR within a single map invocation step. IO redirection between toolspec operations is indicated with the "|" character.

Example:

```
"hdfs:///user/bob/1.ps" > ghostscript s-ps2pdf | unix s-file > "hdfs:///user/bob/mime_type.txt"
```

In a control line, pipes may be used to join an arbitrary number of operations which must support streaming. Input data may be read from a file reference or via a stream. If a control line produces standard output and there is no final redirection to an output file, then the output is written to Hadoop default output file (such as part-r-00000).

3.3 Example Control File

The example snippet below shows a control file that makes use of two chained toolspec operations. The File Information Tool Set (FITS)[5] is applied on a set of files on the HDFS in order to generate an XML metadata document. The used operation *"j-identify"* makes use of Java shipping so that it can be invoked directly by ToMaR within the same JVM (as compared to using the command-line interface). Data is read from the file system and the output stream is redirected to the next operation *"j-xpath-stdin"*, which evaluates an XPath expression and returns the result to stdout.

```
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100001" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100002" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100004" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100005" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100007" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100008" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
fits j-identify-stdout --input="hdfs:///user/bob/html-files/extracted.100010" | xpath j-xpath-stdin --
xpath="//identity[1]/@mimetype"
```

[5] http://code.google.com/p/fits/

4 The MapReduce Application

4.1 Configuration and Submission

ToMaR is implemented as a MapReduce application that can be easily configured and run on a Hadoop cluster. To run a job, a user submits ToMaR as a .jar-file together with a set of parameters. These typically include a path to the toolspec directory on HDFS *(-r option*), the input/control file *(-i option*), and the desired output location *(-o option*). The *–n* option specifies the number of lines per input split, which is important to control the number of initialized map tasks. Other relevant Hadoop options include the options "*-libjars*/*-archives"* to specify resources that are shipped with the job as well as the option "*-Dmapreduce.inputformat.class"* to specify alternative input file formats. A typical command-line example to start a ToMaR job is provided by the example below.

Example:

```
hadoop jar tomar-1.5.2.jar -i controlfile-file.txt -r toolspecs -n 50
```

4.2 Data Decomposition

The control file provides the input data of the MapReduce application, which is split and distributed across the cluster nodes. Hadoop creates an input split for each line by default (causing the creation of a map task for each line of the control file). In most cases it is therefore advisable to configure the number of lines per split (-n option) with respect to the cluster size as well as the nature of the application. The total number of splits for an input file might be selected to be a fraction of the maximal available map tasks. There is however a trade-off between large splits (less efficient with respect to fail-over/load balancing) and small splits which increase the overhead on application. Each line of a split (called a record) is processed iteratively on the worker node the split has been assigned to by the framework. The workload that is imposed on a node by a single record depends on the formulation of the control lines, which can significantly vary between different applications.

4.3 Handling Application IO

Tool specification documents define invocation patterns for individual applications. ToMaR's control file is used to apply these patterns and to map them to concrete data sets. The control file provides the input file for ToMaR which is launched as a MapReduce job.

ToMaR is capable of handling data in the form of file references and streams. The present implementation supports the HDFS file system as well as the Azure blob storage (WASB) as a data source / data sink. Additional file systems and APIs (like for example based on *file://* or *http://* references) may however be added using ToMaR's *Filer* interface. A *Filer* used by ToMaR basically implements methods to read/write data from/to a file system based on input/output files as well as input/output streams.

Depending on the IO capabilities of a tool and/or its application in the control file, data is supplied accordingly by ToMaR. ToMaR copies data from the source file system (e.g. HDFS) to the local file

system for tools that are specified to read data from local file references prior to the execution. File based output data is copied back to the source file system once the record has been processed. ToMaR may also handle data buckets based on directories.
For tools that are specified to support stream-based IO, data is written/read directly from/to the source file system using output/input streams. ToMaR is also capable of redirecting standard input/output streams allowing one to create tool chains. Hence, the output of a toolspec operation can be specified to be directly piped to a subsequent toolspec operation using streams.

5 Using ToMaR on the Pig Platform

5.1 Motivation

The Apache Pig platform supports a high-level data-flow language for large-scale data analytics on top of Hadoop. Pig applications are implemented as Pig Latin scripts which are automatically translated into several MapReduce jobs by the underlying platform. Pig provides a convenient abstraction allowing users to efficiently create parallel applications without having to deal with the complexity of the MapReduce environment. Pig Latin is a procedural language allowing one to perform data transformations based on (database) relations.

Pig provides support for custom operation by supporting the concept of user-defined functions (UDFs). UDFs are Java applications that can be registered with a Pig Latin script and used as part of a statement for example to load/store, filter, or aggregate data. ToMaR implements the necessary hooks to be registered and used within a Pig Latin script. This is motivated by the fact that ToMaR is typically used as part of a larger workflow that needs to analyse the data produced by the wrapped tools. By utilizing ToMaR within a Pig script, one can directly pick up results generated by legacy applications and process and analyse them on Hadoop. Here, data is directly loaded from/into Pig relations and there is no need to generate a control file prior to the execution. The generation of Map and Reduce tasks for performing data manipulations are entirely delegated to the Pig platform. Experimental tests have shown no significant impact on the performance if ToMaR is used through Pig via its UDF, as compared to being run as a standalone MapReduce application.

5.2 Implementation

The class *ControlLineUDF* in the package *eu.scape_project.pt.udf* implements a UDF for handling ToMaR ControlLines in Pig (by extending the Pig *EvalFunc<Tuple>*). As input, the function takes a Tuple with two elements containing a reference to the toolspec repository (which is typically a folder on the HDFS) and a single control line to process. The ControlLine UDF invokes ToMaR as a library (without utilizing any Hadoop-specific code) in order to parse and execute the control line. The creation and handling of MapReduce job is left up to the Pig platform.

5.3 Application

The Pig Latin snippet below provides an example for using ToMaR's control line UDF to generate XML documents using the File Information Tool Set. The documents are subsequently parsed using an UDF for evaluating XPath expressions.

In order to make use of the UDF from a Pig script, ToMaR must be registered with the framework by providing the path to the .jar file. The DEFINE statement is used to assign an alias to the used UDF functions. The script uses the LOAD statement to load a relation containing references to a set of image files residing on the HDFS. FOREACH/GENERATE statements are used to transform data based on expression and columns in a relation. The example script makes use of *ToMaRService* to generate a FITS XML document for each reference in the *image_paths* relation. The result is a new relation *fits* which is subsequently parsed using another UDF named *XPathService*.

Example:

```
REGISTER tomar-1.4.2-SNAPSHOT.jar;

DEFINE ToMarService eu.scape_project.pt.udf.ControlLineUDF();
DEFINE XPathService eu.scape_project.pt.udf.XPathFunction();

image_paths = LOAD '$image_paths' USING PigStorage() AS (image_path: chararray);

fits = FOREACH image_paths GENERATE image_path as image_path, ToMarService('$toolspecs_path',
CONCAT(CONCAT('fits stdxml --input="hdfs://', image_path), '")) as xml_text;

fits_validation_list = FOREACH fits GENERATE image_path, XPathService('$xpath_exp1', xml_text)
AS node_list1;
```

6 Fine-tuning and Optimizations

6.1 Chained Operations

A method to enhance the performance of a ToMaR based workflow is to combine multiple tool invocations per record using IO pipes, which is supported by ToMaR. Chained operations (section 1.1.3) allow a user to reduce the MapReduce overhead which occurs when multiple jobs have to be started in order to execute a workflow.

6.2 Tool-specific Optimizations

Exploiting tool-specific optimization strategies provides an important aspect that can dramatically improve performance of the overall workflow. An example is the recursive processing of multiple files, which is often implemented by specific tools. Recursive processing is typically faster compared to invoking the same tool separately for each payload file. ToMaR supports file references that point to directories. Generating control files that partition payload data into data buckets based on directory, and using recursive tool processing, often helps to improve the performance of tools that suffer from significant invocation latency.

6.3 Exploiting Data Locality

The input file to a ToMar job is provided by the control file which holds references to the actual payload data. While Hadoop is capable of exploiting data locality by allocating CPUs for processing the splits of a large input file which reside closely to the data, this mechanism fails for input files containing only references to the payload data. Although the control file holds information on data locality implicitly it is not exploited per se. ToMaR implements a specific input format class called *ControlLineInputFormat* which enables Hadoop to exploit data **for** the distributed processing of control files. This is achieved by sorting and splitting the control file with respect to the referenced data blocks. ToMaR can be configured to use the *ControlLineInputFormat* by using a generic option for setting the input format class of a job, as shown in the example below.

Example:

```
hadoop jar tomar-1.5.2.jar –i controlfile-file.txt -r toolspecs -n 50
-Dmapreduce.inputformat.class=eu.scape_project.pt.mapred.input.ControlFileInputFormat
```

6.4 Application Shipping and VM Reuse

ToMaR assumes that command-line tools are pre-installed on a cluster at execution time. The invocation pattern that should be applied is specified in a control file and corresponding tool specification documents. ToMaR, however, supports application shipping and direct invocation for Java command-line applications (i.e. Java programs that can be invoked via their main method). This mechanism relieves ToMaR from creating expensive OS-level processes to perform the command-line invocation. Making use of direct Java invocations can significantly improve the application performance for executing Java based third party applications.

In order to specify a Java based invocation mechanism a reference to the Java application launcher and the fully qualified Java class name has to be provided through the *<command>* element of the corresponding toolspec operation.

Example:

```
<command>/usr/bin/java edu.harvard.hul.ois.fits.Fits -i ${input} -o ${output}</command>
```

Using the Hadoop options *-libjars* and *-archives* third party archives and Java libraries can be made available to a MapReduce application through the cluster's distributed cache. These options should also be used when using Java application shipping with ToMaR.

Example:

```
hadoop jar tomar-1.5.2.jar -libjars $APPCLASSPATH -archives ./fits-0.6.2/fits-xmls.tar#xml,./fits-
0.6.2/fits-tools.tar# –i controlfile-file.txt -r toolspecs -n 50
```

7 Links and Further Reading

ToMaR project on Github: https://github.com/openplanets/tomar

ToMaR paper at CCGrid2014:
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6846550&contentType=Conference+Publications

Blog Post on Using ToMaR for ARC to WARC Migration:
http://www.openplanetsfoundation.org/blogs/2014-03-24-arc-warc-migration-how-deal-de-duplicated-records

Blog Post on Using ToMaR with Apache Pig:
http://www.openplanetsfoundation.org/blogs/2014-06-24-will-real-lazy-pig-please-scale-quality-assured-large-scale-image-migration

8 Conclusion

The development of ToMaR was primarily motivated by the strong need for executing SCAPE tools and workflows in a scalable fashion, dealing with data volumes beyond test examples. In the project, experimental application scenarios have been developed using local workflows, legacy applications, and novel applications developed in a variety of languages. Testbed practitioners, however, needed a generic means to formulate scalable preservation workflows. As these workflows are typically IO bound, the incorporation of data intensive computing technology was desired.
However, the fact that the developed preservation workflows highly depend on legacy applications and content-specific libraries (like ffmpeg, OpenCV, scientific libraries) was a major obstacle to migrating them to a platform like Hadoop. Due to the diversity of the developed workflows, re-engineering these individual applications to fit a particular programming model / execution environment was not feasible. Consequently, ToMaR was developed in order to provide a flexible application wrapper for executing legacy application efficiently on MapReduce clusters. ToMaR can be used both as a standalone MapReduce application as well as in combination with the Apache Pig platform to implement complex data transformation workflows.

9 References

[1] C. Becker and A. Rauber. Decision criteria in digital preservation: What to measure and how. J. Am. Soc. Inf. Sci. Technol., 62(6):1009–1028, June 2011.

[2] Luis Faria et al. Automatic preservation watch using information extraction on the web. In Proc. of the Tenth Int. Conf. on Preservation of Digital Objects (iPres13), Lisbon, Portugal, September 2013.

[3] R. Schmidt. An architectural overview of the scape preservation platform. In Proc. of the Ninth Int. Conf. on Preservation of Digital Objects (iPres12), Toronto, Canada, October 2012.