2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems

Extracting Information from Scientific Papers in the Cloud

Petr Škoda, Svatopluk Šperka, Pavel Smrž
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
{iskoda,isperka,smrz}@fit.vutbr.cz

***Abstract*—This paper deals with a system for extracting information from scientific papers. We analyze drawbacks of an existing implementation running on the N1 Grid Engine. Reasons for moving extraction to the Cloud are presented next. The architecture of the Cloud port is discussed and the links to the API and the platform developed within the mOSCAIC project are elaborated.**

Keywords—information extraction; N1GE; Cloud; mOSAIC

I. INTRODUCTION

ReReSearch is an experimental project being developed by our team which aims at building a knowledge base and derived personalized portals about research. The key entities it operates on include researchers, teams and universities, papers, reports and deliverables, books, journals, proceedings and various collections, conferences, workshops and seminars, projects and funding agencies. Information on all of these entities has to be interconnected in order to be useful. To gather the data, the system first identifies relevant sources on the Web and then downloads and processes specific web pages and papers. To transform data from unstructured form into a structured one, information extraction methods are applied.

This paper focuses on a crucial step of the process – information extraction from scientific papers. Papers are collected by special crawlers that search the Web for the pages possibly containing links to papers (e.g., online proceedings or lists of publications linked from homepages of authors that the system already "knows"). Printable formats (PDF, PS, DOC, etc.) are transformed into a semi-structured text (text structuring information such as sections or list items can be recovered). The process of information extraction from the texts may take from a few seconds to several minutes, depending on the complexity and the length of the document.

It is hard or practically impossible to exactly predict the number of scientific papers that the system can acquire in a given time. It is essential to process all the acquired documents as fast as possible in order to maintain the database actual. Currently, we employ the *N1 Grid Engine* (N1GE, formerly developed by Sun, now owned by Oracle) to process large batches of papers. The grid is available to all teams at our faculty so it is often highly utilized. There are also other drawbacks that will be discussed in the paper.

Based on the analysis of the current state, we decided to port the processing components into a Cloud. The platform should provide scalability and elasticity so that the application can deal with large peaks of incoming papers and can process them faster and more reliably than with the current implementation based on N1GE. Moreover, we aim to use the platform developed within the mOSAIC project[1] that offers provider-agnostic APIs allowing migrating between cloud providers and even scaling beyond one, if necessary.

The rest of this paper is organized as follows: Section II describes the process of information extraction from scientific articles in detail. Section III elaborates on the current implementation which employs the N1 Grid Engine and details the analysis of limited performance and other weaknesses of the realized system. Section IV describes the proposed architecture of the Cloud-based implementation and its key components.

II. PROCESSING SCIENTIFIC PAPERS

As mentioned in Introduction, the information extraction from scientific papers takes several steps before the data can be stored in the ReReSearch (RRS) system. We briefly introduce the workflow of the part of RRS responsible for the document processing in this section. We also detail the metadata extraction process and characterize major building blocks of the current implementation.

A. Getting data into the ReReSearch database

There are multiple ways in which information about a publication can be identified and added to the RRS database.

The metadata of documents for which there is no fulltext version yet are typically obtained from digital repositories like *CiteSeerX* or *DBLP*. Bibliographic data can also be obtained by means of information extraction from references contained in texts of publications. Using these data, we can model relationships between papers, authors and other entities in the database.

A simplified schema of the part of ReReSearch concerned with publications is depicted in Figure 1. Only a subset of data gathering modules are represented in the schema. The key elements can be characterized as follows:

[1] http://www.mosaic-cloud.eu

978-0-7695-4687-2/12 $26.00 © 2012 IEEE
DOI 10.1109/CISIS.2012.176

- *RRS DB* — the main database of the ReReSearch system storing structured data as well as indexes for the fulltext search;
- *Document source files store* — storage of original versions of documents (source files);
- *Metadata extractor* — the component performing information extraction on given documents;
- *CiteSeerX/DBLP import* — a module for searching CiteSeerX and DBLP for bibliographic data about new articles and publications;
- *Homepage search* — a module searching author's homepages and lists of their publications and citations; results can be either document source files themselves or just bibliographic data;
- *Document source file search* — a module searching for document source file according to given bibliographic data.

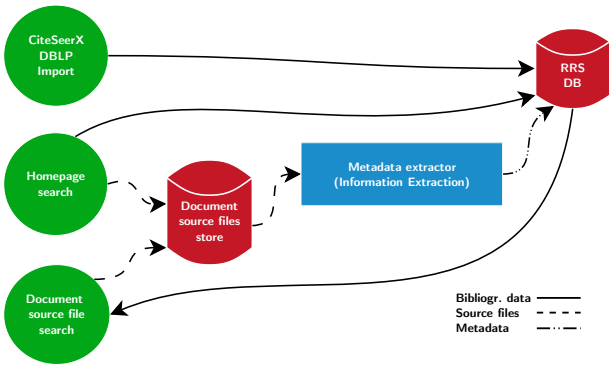

Figure 1. ReReSearch — getting the data into the database.

CiteSeerX/DBLP import and *Homepage search* modules import bibliographic data directly into the database. *Document source file search* and also *Homepage search* are able to find publications in the form of PDFs, PostScripts, etc. that are later processed by the *Metadata extractor* component. Output of this extractor is then fed into the database.

The process can start by giving a name of an author to the *Homepage search* module by the *RRS control system.* It can be a result of an addition of new documents to the RRS DB or it can be triggered by an end user's request to find information on a particular author and his publications. The *Homepage search* module starts a search for a web home page of the author by using various search engines. The HTML code of the home page is analyzed and the module tries to detect links to a page with author's publications or to find publications on the homepage itself. All extracted bibliographical data are loaded into the ReReSearch database and publications are saved to the *Document source files store* in order to be processed by the *Metadata extractor* as soon as possible. The *Homepage search* module then starts again by looking for home pages of previously unknown persons encountered as co-authors and as authors of cited publications.

Bibliographic data of new publications, for which the publication itself was not found, are meanwhile passed to the *Document source file search* module that tries to find it on the Internet. Currently, this module is successful for 10 % of publications. Found files are again processed by the *Metadata extractor* module.

The process triggered by one person passed to the *Homepage search* module can continue for a long time and can find thousands of new persons and documents.

We used our current database that contains bibliographic data on about 10 million documents to estimate how many authors and publications we can obtain by following co-authorship and citations. Because many authors quote famous authors, at some point the amount of new publications and new authors is the same no matter who one we start with. An example graph characterizing the total number of authors and publications collected after several iterations is given in Figure 2. An iteration consists of gathering: publications of authors found in the previous iteration; all co-authors of these publications; and publications and their authors cited in those from the previous iteration. After a few iterations, the number of publications and authors gets easily above 100,000.

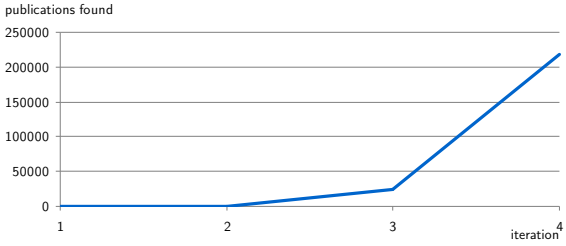

Figure 2. Number of publications obtained by following co-authorship and citation links during four iterations for "Daniel Abadi".

For a practical perspective, Figure 3. captures a number of publications found by the *Homepage Search* module for a starting name. As the process continuously progresses and more connected authors are found, after two hours the number of publications found reaches 1,100.

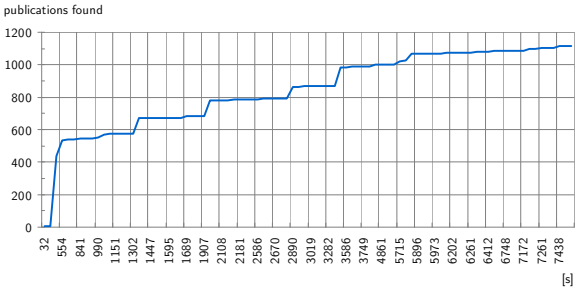

Figure 3. Process of finding publications by the Homepage Search module for starting name "Weatherhead" captured in time.

B. *From printable versions to semi-structured data*

Documents in PDF, PS, DOC and other formats downloaded in the previous phase need to be transformed into a semi-structured form appropriate for the full-text indexing. Additional metadata (such as author's affiliation and e-mail address) are extracted as well. The process comprises a

number of steps. Some of them are tied together so they need to be executed in a given order. Some metadata extractors use large dictionaries to support the analysis of documents. These aspects influence the way the processes can be implemented (see the next subsection). A simplified schema of the *extractor* used in RRS is given in Figure 4.

The extraction process starts by passing a source file of a document to the *Document to Text Transformer* where the source file (PDF, PS, DOC) is transformed into the semi-structure form. The resulting text is cleaned from unsuitable characters, special characters, dashes and ligatures by the *Text Cleaner*. The *Document Splitter* then detects main structure of the document (chapters, citations, etc.) and the *Language Identifier* recognizes the language of the document so that it can be handled using language-specific dictionaries and rules. The *Meta Extractor* searches documents for keywords, abstracts and titles. The *Email Extractor* then looks for all e-mail addresses in the document. In the next step, the *Entity Extractor* finds authors of the document and makes relations between authors and their e-mail addresses, if they were identified. In the end, the *Entity Extractor* searches for a publisher of the document. All data gathered during the extraction are finally composed into an XML file and sent back as a result.

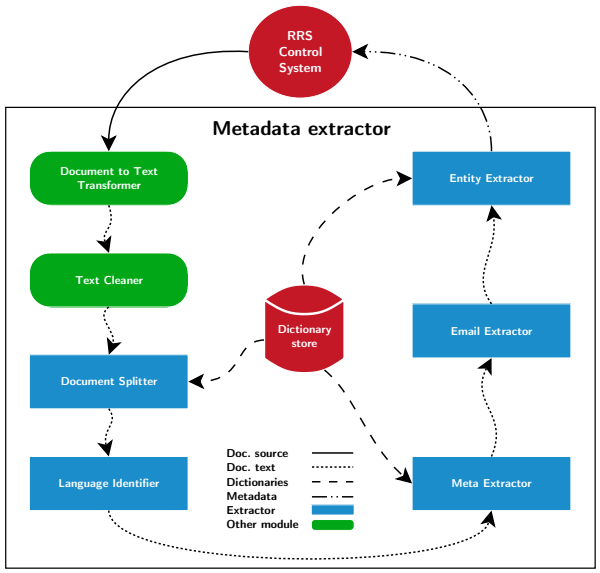

Figure 4. Metadata extractor—from document source to metadata.

The metadata extractor can be extended by other extraction modules, depending on the requirements of each particular case. Currently, the extractors are implemented as Python classes, while *Document to Text Transformers* are legacy libraries written in various languages.

III. Current Implementation

Even before the development of the *Metadata Extractor* had been finished, it was clear that this process would be extremely demanding on a single machine. For example, there are more than 900,000 source files of documents in our store that have not been processed yet (they were downloaded using links gathered from a CiteSeerX traversal). It would take several months to process this amount of documents on a single machine. Obviously, alternative scenarios need to be considered.

A. N1 Grid Engine

There are various ways to use the power of multiple computers. The general concept of a computer grid (consisting of heterogeneous nodes) is the direction our system currently follows. The N1 Grid Engine (N1GE) is able to perform operations on unused computers in a network that are configured as the execution hosts of the N1GE. The computers do not need to be dedicated for the grid so any workstation can be in it's idle times (e.g. during the night) employed in the grid. The N1GE scheduler takes care of monitoring available resources of computers in the grid and plans jobs for them accordingly [1].

The Faculty of Information Technology, Brno University of Technology, runs the N1GE on a number of dedicated high-performance servers and hundreds of workstations from computer labs and offices. The number of available single thread cores in this grid is usually between 700 and 1,200. The grid is available for any research group at the faculty so the load is often very high. Priorities are used for job planning, a higher priority means sooner finalization of the job. A disadvantage of the simple priority approach is that a high priority for a large long-running job can remarkably lower the computational power for all other jobs.

B. Metadata extraction on N1GE

The overall schema of the RRS extraction on the N1GE is showed in Figure 5. As the execution hosts are distributed across the network, we are using a shared file server as a source of all necessary data and executables: source files of documents (PDF, PS, DOC, etc.), dictionaries, executables and libraries. In the other direction, outputs of the extraction modules (in the form of XML with metadata and files with extracted texts) can be stored on the shared file server.

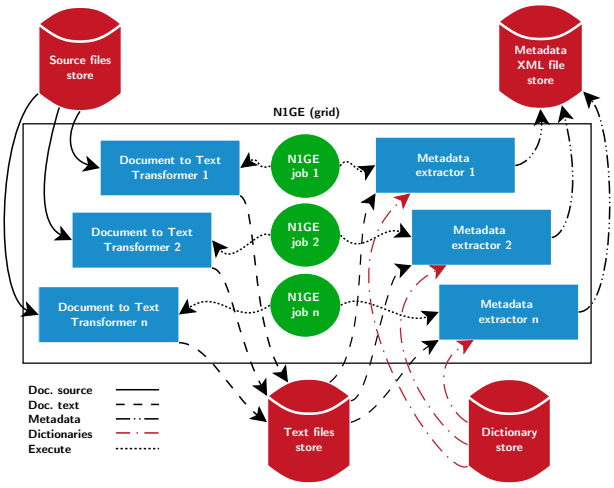

Figure 5. Extracting metadata—current environment with the N1 Grid Engine; extraction in the grid runs on *n* machines.

N1GE operations are handled as *jobs*. A scheduler decides when and on which hosts will be jobs executed. From the scheduler's point of view, an ideal execution time is rather minutes than seconds as there is a provisioning overhead. That is why we run 10 extractions per job. The overall average time needed to run all extractions is 83 seconds (see in Figures 6 and 7). The RRS control system prepares jobs and sends them to the N1GE queue.

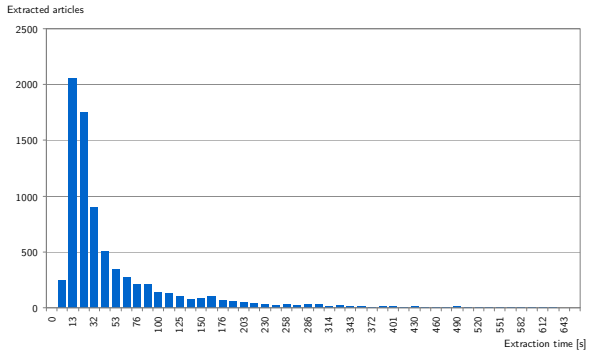

Figure 6. Histogram of extraction times (data from 7854 extractions)

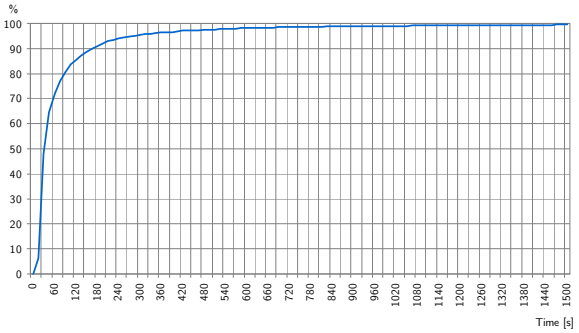

Figure 7. Percentage of documents that can be processed in a given time (data from 7854 extractions)

C. *Shortcommings of the current implementation*

ReReSearch allows users to target the system to new topics, authors, publications and other relevant entities reflecting their specific needs. Unfortunately, N1GE cannot scale up behind some limits. Also, the actual performance cannot be estimated in particular cases as the availability of computational power depends on other activities in the system [2]. Though there are concepts supporting market-like scheduling policies in N1GE [3], they are not easy to apply in heterogeneous environments. The current system cannot deal with peaks in the number of documents to be processed.

On the other hand, even if a job contains only one or a few documents, the job will be very probably just queued and executed much later than would correspond to an average processing time. The length of the queue grows either because of high utilization of the grid by other users or because of hardware limits (e.g., the network bandwidth) preventing the system from running more jobs at the same time. We were able to run only 110 or less jobs at a time during tests on a heavy loaded N1GE.

Another issue consists in sequential processing of documents in each job. Unfortunately, it is not always possible to predict the processing time from the metadata or the length of the document (given by the number of pages or its size in bytes—Figure 8). Our testing on 7854 documents showed that the average of the processing times is about 2.5 higher than the median. Further analysis revealed that about 0.5 % of documents take a time much longer than expected. Documents in a job after a "slow" one have to wait, there is currently no way to re-schedule the processing.

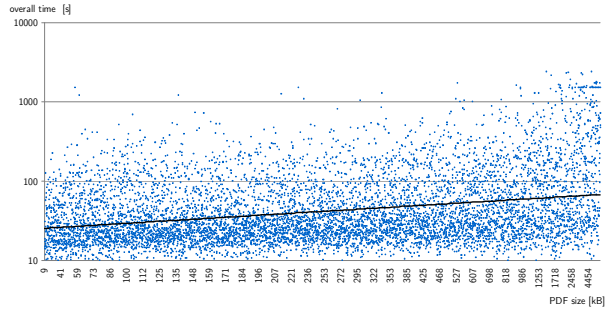

Figure 8. Extraction time in relation to size of an input PDF file.

Finally, the faculty grid is primarily intended for new research; it should not be occupied by continuous computation of long-term applications in their deployment phase.

IV. Towards Implementation in Cloud

Limitations described in the previous section—mainly the lack of guarantees concerning availability of computational resources and the inability to scale the system according to actual needs make us to choose a new infrastructure that would better meet the requirements of applications based on ReReSearch.

Cloud can be defined as a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers [4]. Thus, one of the essential characteristics of the Cloud environment is *elasticity*—the ability to add or remove resources at a fine grain and with a lead time of minutes rather than weeks that allows matching resources to workload much more closely [5]. Elasticity is exactly the feature that fulfills the key requirement of ReReSearch – the ability to process incoming papers as fast as possible by scaling up computational resources whenever they are needed.

There are two service models relevant for a developer who wants to run an application at a Cloud provider. The first model is *Platform as a Service* (*PaaS*) where the consumer does not have control over the underlying infrastructure (servers, storage, OS or network) but only over deployed applications created using programming languages and tools supported by the provider. The second model—*Infrastructure as a Service* (*IaaS*)—provides capability to provision fundamental computing resources where the consumer is able to deploy and run arbitrary software [6]. The

developer then usually configures or provides his environment in the form of a *Virtual Machine* (*VM*).

It would be tedious and error-prone to rewrite the codebase of ReReSearch from Python to another language. The existing code uses many libraries; it requires a specific version of Python and a specific setting represented by environment variables. It is necessary to reuse as much as possible when porting the system to the Cloud. This fact made us to aim at the *IaaS* concept and target Cloud infrastructures that allow setting up all necessary features in custom VMs.

It is also essential to avoid the provider lock-in – ReReSearch is a representative of systems that can benefit from portability across clouds and cloud providers. *mOSAIC*—a language- and platform-agnostic API for usage of multi-Cloud resources, and at the same time a portable platform for utilization of Cloud services based on the API and Cloud usage patterns [7] provides such flexibility. It allows one version of an application to be deployed to any supported provider. Moreover, it aims at auto-scalability of applications [8], i.e., elasticity.

The *mOSAIC*'s API has several layers: *Connector API* abstracts types of resources commonly offered by cloud providers such as message queues, key-value stores or distributed file systems. *Driver API* is at the bottom of the mOSAIC API hierarchy; it sits on top of a native API for a particular resource and enables a uniform access protocol. This layer offers plugins where each plugin enables access to one implementation of a resource type. Intermediate *Interoperability API* ensures language independency of the API and thus it mediates between the Connector API implementation and a compatible driver implementation. Connector API and Cloudlet API—the APIs offered by the platform to developers—are currently only available in the Java programming language.

User components are called *cloudlets*. They use the *Cloudlet API* that handles life-cycle of cloudlets, enables initialization, configuration, migration and obtaining bindings to used resources [8]. In mOSAC terms, a *cloud component* is a *cloudlet* and there are one or several *containers* within which one or more *cloudlet instances* execute. The number of instances is under control of the container and is managed in order to grant scalability [9], i.e. the mOSAIC platform controls scaling up and scaling down of an application. In order to ensure scalability, components should communicate indirectly—usually by using message queues as intermediary so more cloudlet instances can dispatch messages from the same queue.

A. *Application Architecture*

The architecture of the scientific paper processing application expressed in terms of *cloudlets* and *Cloud resources* can be seen in Figure 9. The purpose of individual components and application's overall workflow is described in the following paragraphs.

Communication between the ReReSearch control system and the application in the Cloud is realized by means of HTTP gateways. The process starts when the RRS control system submits an HTTP POST request containing a URL of the document to be processed by the Cloud. Delivery of extractors' output is handled by the same method with roles of the control system and Cloud app switched (i.e. Cloud app submits an HTTP POST request to the RRS control system).

The *Saver* cloudlet receives URLs through a queue from the HTTP gateway (cloudlets always communicate indirectly through a queue in the designed system). It downloads documents from given locations and passes them to the *Source File Processor* along with respective URLs. The *Source File Processor* combines several modules from the architecture depicted in Figure 4 (Section II.B)—the *Document to Text Transformer* that extracts a textual content from the source file; the *Text Cleaner*; the *Document Splitter* and the *Language Identifier*. After the source file is processed by the pipeline consisting of these modules, the *Source File Processor* adds the output into the key-value store under the key computed as a hash of the URL from which the file was downloaded by the *Saver*.

Textual content is fed into extractors by means of the key identifying it in the key-value store. Extractors can run parallel in order to speed up the process. Serialization is only enforced when there is dependence between modules (see Figure 9). That is the case of the *Entity Extractor* that assigns persons to email addresses extracted by the *Email Extractor*. We refer to the rest of the extractors as *Extractor 1* to *Extractor N* to stress the fact that the number of extractors will increase in time as more components are being developed.

Each extractor processes the content and passes a result of the extraction either to the next extractor or to the queue from which the *Sender* cloudlet reads messages. The *Sender* simply submits a POST request to the specified URL with the result of a particular extraction. Outputs of extractors from one source file are combined by a ReReSearch specialized module. This task is not handled in the Cloud.

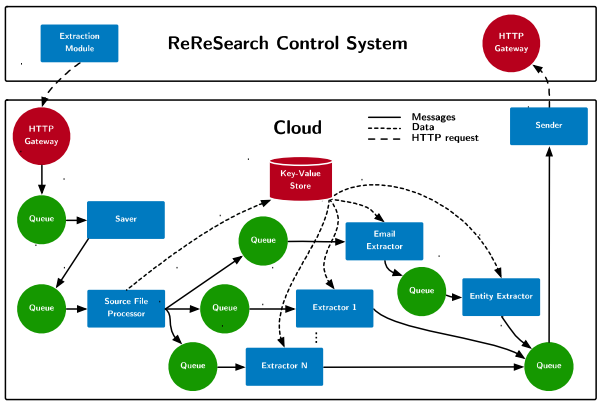

Figure 9. Architecture of information extraction application in the cloud

As already mentioned, it is undesirable to rewrite our codebase from Python to another language—e.g. *Java* (or other JVM compatible language such as *Scala*)—in order to run extractions on the mOSAIC platform. The extractor cloudlets and the *Source File Processor* cloudlet thus simply wrap existing Python and other legacy code. The rest of cloudlets implement their functionality directly in Java. We are preparing a Virtual Machine containing all required li-

braries, scripts and software so that it could be used from the mOSAIC cloudlets.

V. CONCLUSION

We presented a system for information extraction from scientific papers and an existing infrastructure based on N1GE the software currently runs on. As we do not have an exclusive access to the infrastructure, the availability of computational capacity cannot be guaranteed. This limits an ability to handle peaks of requests (a number of scientific papers waiting to be processed). We argue that the Cloud environment and its elasticity can cope with the discussed issues. In order to guarantee portability of our application to new Cloud providers and, in case of a need, to be able to scale beyond one provider, we aim to use the *mOSAIC* platform and its API. We describe the architecture of the application that is being implemented.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 256910 (mOSAIC Cloud) and by the IT4Innovations Centre of Excellence project, Registration number CZ.1.05/1.1.00/02.0070, supported by Operational Programme "Research and Development for Innovations" funded by Structural Funds of the European Union and the state budget of the Czech Republic.

REFERENCES

[1] Sun Microsystems, Inc. *Sun N1 Grid Engine 6 User's Guide*. Santa Clara, CA, USA. http://docs.sun.com/app/docs/doc/817-6117/, 2004.

[2] C. Chaubal. Scheduler *Policies for Job Prioritization in the Sun N1 Grid Engine 6 System*. Technical report, Sun BluePrints Online, Sun Microsystems, Inc., Santa Clara, CA, USA. 2005.

[3] Stößer J., Bodenbenner P., See S., Neumann D. *A Discriminatory Pay-as-Bid Mechanism for Efficient Scheduling in the Sun N1 Grid Engine.* Institute of Information Systems and Management Universität Karlsruhe. 2008.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility", Future Generation Computer System, vol. 25, issue 6, June 2009, pp. 599–616, doi:10.1016/j.future.2008.12.001

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "A View of Cloud Computing", Communications of the ACM, vol. 53, issue 4, April 2010, pp. 50–58:10.1145/1721654.1721672

[6] P. Mell and T. Grance, "NIST Definition of Cloud Computing", National Institute of Standards and Technology, 2009

[7] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, M. Loichate, "Building a Mosaic of Clouds", Euro-Par 2010 Parallel Processing Workshops, LNCS, vol. 6586, pp. 571–578, Springer, 2011

[8] D.Petcu, C. Crăcium, M. Neagul, M. Rak, I. L. Larrarte, "Building an Interoperability API for Sky Computing", Proc. 2011 International Conference on High Performance Computing and Simulation, Workshop on Cloud Computing Interoperability and Services (InterCloud 2011), IEEE CS, pp. 405–412, 2011

[9] G. Macariu, "mOSAIC API: Java Programming Guide", eAustria Research Institute, 2011