

Scaling a Plagiarism Search Service on the BonFIRE Testbed

András Micsik, Péter Pallinger
 Department of Distributed Systems
 MTA SZTAKI
 Budapest, Hungary
 {andras.micsik, peter.pallinger}@sztaki.mta.hu

Dávid Siklósi
 Informatics Laboratory
 MTA SZTAKI
 Budapest, Hungary
 {david.siklosi}@sztaki.mta.hu

Abstract—The KOPI Online Plagiarism Search Portal – a nationwide plagiarism service in Hungary – is a unique, open service for web users that enables them to check for identical or similar contents between their own documents and the files uploaded by other authors. As our recent result, we can also detect cross-language plagiarism, but with a highly increased computational demand. The paper describes our experiment with the BonFIRE testbed to find a suitable scaling mechanism for translational plagiarism detection in a cloud federation.

Keywords—cloud federations; cloud testbeds; scaling; elastic computing; plagiarism search

I. INTRODUCTION

The KOPI Online Plagiarism Search Portal [1] of SZTAKI has been a well-known, operational service since 2004 for English and Hungarian languages. For professors and teachers KOPI gives the opportunity to compare theses and home assignments to all documents uploaded before, or to various document sets openly accessible on the Internet. Students can check their own written works to see if the amount of citation has exceeded the limit set by their home institution. They can also protect their theses by uploading them to the system under their names.

Recently, an innovative feature – cross-language plagiarism detection – was implemented within KOPI, for the first time in the world [2]. For example, with the cross-language feature it is now possible to find paragraphs taken from English Wikipedia and translated into Hungarian. This new technique is costly in terms of processing and data storage; therefore we sought solutions for scaling our service using cloud technologies. The paper describes our experiments for the evaluation of scaling solutions using the outstanding testing and monitoring facilities of BonFIRE.

BonFIRE [3] offers a multi-site testbed with heterogeneous cloud resources, including compute, storage and networking resources, for large-scale testing of applications, services and systems targeting the Internet of Services community.

BonFIRE provides a test platform with an API and a portal both supporting the uniform management of compute nodes, data blocks and network connections in the federated environment of 7 clouds. Among the specific features of BonFIRE one can find bandwidth control for network,

integration with Amazon, and a ubiquitous monitoring framework, which is of prime importance for us.

The next section of the paper describes the architecture and preparations for the KOPFire experiment, section 3 analyzes the possible scaling actions, while section 4 describes the concrete scaling solution and its measurement. Section 5 explains aspects of fault tolerance in the solution, followed by a conclusion.

II. KOPI ARCHITECTURE FOR SCALING

The new cross-language plagiarism search feature of KOPI (introduced in 2011) requires a lot more resources than the original single-language service. We needed to see the possibilities for making the service faster and more adaptive using clouds and their federations. The KOPFire experiment started with the design of the experiment. Before the design is presented we need to explain the data flow of the KOPI service.

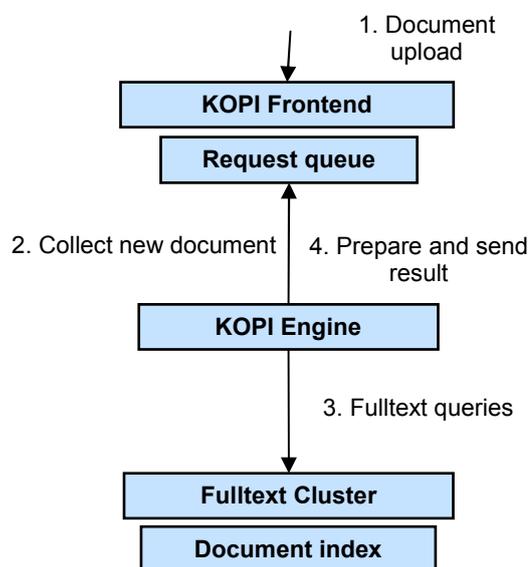


Fig. 1. Overview of request processing in the experiment

The KOPI service works asynchronously: it accepts requests in the form of uploaded documents, which are checked

for copied content over various databases. After this, a report is sent to the user containing the copied parts and their original sources. Processing of incoming user requests is based on a queue, from which processing nodes take out requests and put back results after processing.

The steps of processing user requests are shown in Fig. 1. First, a user uploads a document, which is stored as a request on the KOPI Frontend, a simplified version of the KOPI portal. The KOPI Engines regularly poll the frontend for new requests to be processed. When a KOPI Engine gets a request, it starts processing the document, and during this, it sends many fulltext queries to the Fulltext Cluster consisting of several virtual machines and index data blocks. With the results of the fulltext queries the KOPI Engine compiles the results of plagiarism search. Finally, the result is sent back to the Frontend, which means it is ready for download by the user. In the results, the user receives a list of potentially copied parts in his document together with their sources and the probability of plagiarism.

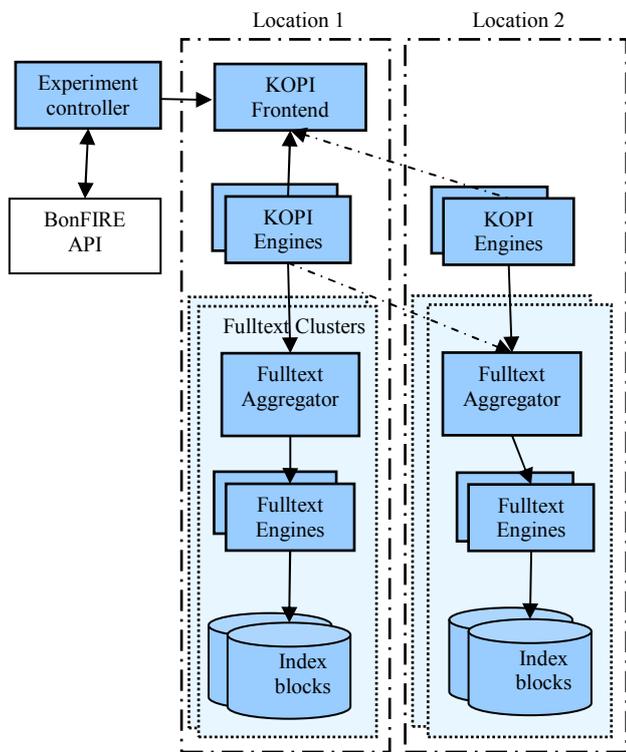


Fig. 2. Basic experiment setup

The service has been modelled in the BonFIRE environment according to Fig. 2., and has been set up and tested at three BonFIRE locations: EPCC, INRIA and HLRS. The Experiment controller is the main entry point for managing and running experiments. This contains several custom scripts developed specifically for the BonFIRE experiments. The KOPI Frontend is a simplified web frontend for KOPI, which imitates receiving requests from users and maintains the queue of documents to be processed. The number of KOPI engines acquiring jobs from the queue can be scaled arbitrarily, also across clouds. Fulltext Clusters can be scaled as a whole, where

each cluster contains an aggregator host and several hosts running the fulltext query engines. The fulltext index is partitioned into index blocks, where each index block is mounted by a single query engine. Thus fulltext queries are run in parallel over all index partitions (i.e. all Fulltext Engines in the cluster), and the results are collected and merged by the Fulltext Aggregator node in the cluster.

The cooperation between nodes in the architecture is based on the frontend and configuration settings at each node. The address of the Frontend and available Fulltext Clusters are configured in each KOPI Engine. The KOPI Engine not only asks for new document, and sends results to the Frontend, but also reports periodically about its progress, so that the Frontend knows the percentage of the document processed so far. Based on this, the Frontend can calculate the amount of work yet to be done on all documents in the queue.

As an approximation we can assume that the KOPI Engine issues some fulltext queries for each sentence in the document while processing. For this, it selects randomly one Fulltext Cluster from the list. Due to the implementation, the queries cannot be sent in parallel, but the result of each fulltext query must be received before doing any further processing. Typically, finishing a document this way takes 20 to 50 minutes (depending on document size). The migration of a running process to another KOPI Engine is not solved as it would be too complex because of the large number of intermediate results to be moved. Therefore, a document can either be finished or put back into the queue in which case the intermediate processing results are lost.

The Fulltext Aggregator connects to all Fulltext Engines according to its local configuration. Nodes running Fulltext Engines must have the index partition mounted as a separate data storage block, which can be done via local mount or via remote NFS to the shared NFS server of BonFIRE. The number of threads running inside the fulltext query process can also be locally configured.

The overall question is how to scale KOPI Engines and Fulltext Clusters in order to stabilize the end-to-end response time of the service. The response time consists of the waiting time in the queue and the processing time at one of the KOPI Engines. As vertical scaling is infrequently supported by current clouds, we need to limit ourselves to horizontal scaling solutions. Furthermore, replacing component nodes with faster virtual machines (VMs) is not a viable option, as VM setup is slow and running processes should not be interrupted. With horizontal scaling, the processing time depends on the speed of the Fulltext Cluster, but basically it can hardly be influenced in a given setup, as long as the Fulltext Clusters are not overloaded. Therefore, we need to focus on manipulating the waiting time of document requests. For this task, first we need to understand the effects of individual scaling actions and also the time needed to perform these scaling actions.

III. SCALING ACTIONS

The Fulltext Cluster was created in two variants for the experiment based on a smaller and a bigger index. The small index contains 5 index partitions of 2.1 GB size. The big index has 10 partitions of 10.7GB size. Each partition has a dedicated

VM and one aggregator node per cluster, which collects and merges the results from index partitions. Therefore, the small cluster contains 6 VMs and the big cluster contains 11 VMs.

The time needed to create a cluster varied by cloud location in a range of 2.5-7 minutes for the small cluster, and 9-12 minutes for the large cluster. Stopping a cluster requires to issue the delete commands for all VMs which usually took 22-65 seconds. However these time values may further increase in case the cloud is overloaded.

It is worth to mention, that cluster creation requires that all index partitions are uploaded to the cloud and available as a mountable storage block. This is done only once when including a new cloud to the scaling environment, but the full upload time can take 3-12 hours in case of the big index. This is a considerable delay which forbids the quick expansion to new clouds as a remedy for sudden bursts in usage.

Creating a KOPI engine is simpler, yet it can take 50 to 120 seconds depending on the current load in the cloud.

It is also important to see the relations of various configuration settings with the overall speed of processing. We found that the processing time is roughly linear to the document size, although there may be big differences when document language or style differs. We also observed that there is little regularity in response times for fulltext queries issued in the background while processing a document.

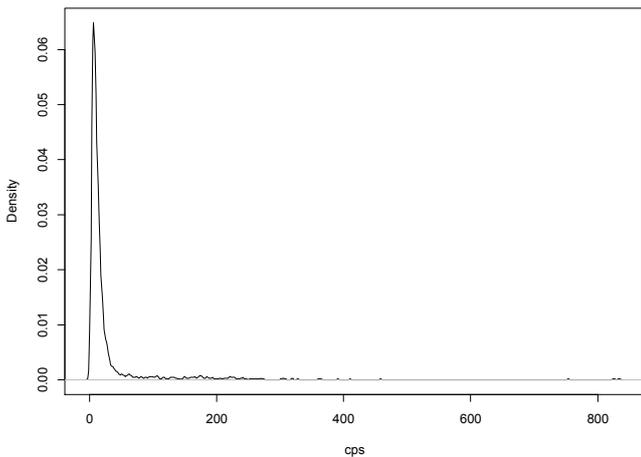


Fig. 3. The normalized distribution of cps values measured in a KOPI Engine

We needed some way to characterize the speed of processing documents; therefore we introduced a new metric: characters per second (cps); which tells us how many characters of a document have been processed in a second. In a fairly stable work context the cps measured at a single KOPI Engine has a clear trend (Fig. 3).

We ran several experiments to measure the overall speed of possible setups. First, the number of threads within each fulltext query engine can be configured. Compute nodes with 2 cores and 2 GB RAM have been used for Fulltext Engines, and we found that the query performance is optimal with 4 threads in this case.

The overall processing speed of KOPI Engines is maximal in case the number of KOPI Engines using the cluster equals to the threads running in the query engine process (Fig. 4), although a small deviation from this causes no serious degradation of performance. Fig. 4 also shows that there is not much difference in speed between locally and NFS mounted index partitions.

Regarding the KOPI Engine nodes the number of document checking processes depends on the memory available, so a small VM instance with 1GB RAM can tolerate up to 4 processes, but running just 1 or 2 processes is the safest.

Further measurements revealed that the round-trip time of network communication between different clouds is about 30ms, which means that effectively it is almost the same to have component in a local or in a remote cloud. Furthermore we did not need controlled network bandwidth between clouds as the data transferred between nodes is typically small (some hundred kilobytes at most). However there is significant difference in the speed of I/O and VM management operations among the clouds in the federation.

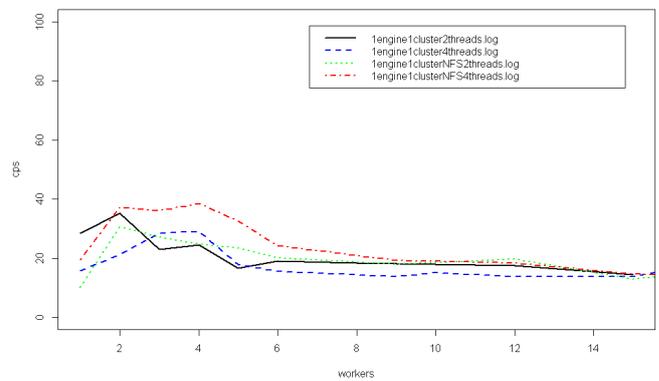


Fig. 4. Processing speed for several workers using one fulltext cluster

IV. SCALING ALGORITHMS

The scaling solution was implemented using the Ruby Restfully add-on for BonFIRE which provides easy manipulation of resources via the BonFIRE API as well as easy access to monitoring data from the BonFIRE Aggregator running Zabbix. The script configures and runs controlled experiments while collecting monitoring data at the same time.

A scaling experiment is initiated by setting the desired limits for the number of clusters, number of KOPI Engines, measurement intervals, etc. and selecting a queue sample (i.e. a sample usage pattern recorded) and a scaling algorithm. Then the KOPI Frontend and other initial components are configured and document processing is started. During the experiment, the important metrics are collected frequently from the BonFIRE Aggregator. The scaling algorithm is run in a loop:

- A decision is made about necessary scaling of KOPI Engines.
- KOPI Engines are scaled.

- Fulltext Clusters are scaled according to the current number of KOPI Engines. The goal is to have a cluster for every 4-5 KOPI Engines.
- The status of document processing is checked: if all documents are done then the experiment is finished. If there are documents with no progress for longer time, the environment is checked for errors.
- Waiting a configurable amount of time (typically two minutes) before next scaling decision.

In the Frontend additional measures were implemented, some of them were collected by Zabbix, some of them were fetched on-demand by the scaling script. The former group of measures include the queue size, the current cps, the number of characters altogether in the queue, etc. The latter group contains data helping decision making and fault tolerance: list of engines that seem to stopped working, time of latest report from any engine, etc.

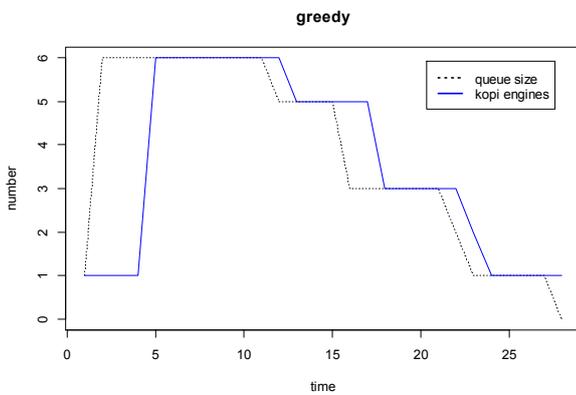


Fig. 5. Processing documents with ‘greedy’ scaling

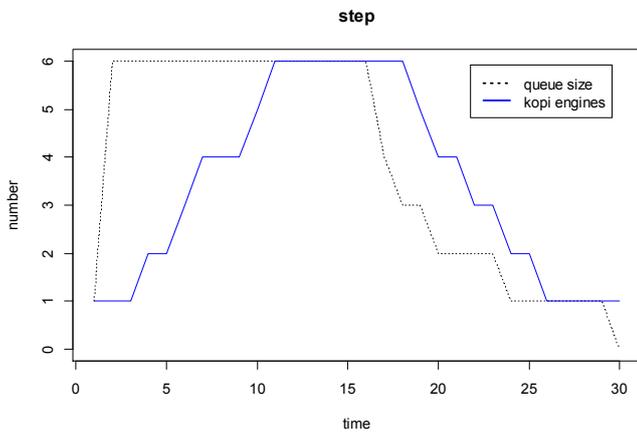


Fig. 6. Processing documents with ‘stepper’ scaling

This general framework can serve for various scaling solutions. We have chosen some threshold-based techniques as characterized in [4]. The natural option is to scale based on the queue size which means the number of documents for which processing has not finished yet. Our ‘greedy’ algorithm tries to start the processing of all documents in the queue, within a

given maximum allowed for compute nodes. A sample run is presented in Fig. 5, showing the change in queue size and number of running KOPI engines over time (on a minute scale) until all documents are processed. A more traditional ‘stepper’ algorithm scales up or down with a single KOPI Engine in one cycle (Fig. 6), also based on the queue size. We also thought about scaling based on the size of all documents in the queue, but this won’t help as we cannot speed up the processing of a single large document by scaling up.

Another approach for scaling can be based on the processing speed. In case we want to achieve a certain processing speed (cps) for all documents in the queue, we can set the following goal:

$$desired_cps * queue_size = measured_cps * engine_size$$

where *engine_size* is the suggested number of KOPI Engines currently. In order to smooth the oscillation of cps, *measured_cps* should be an average of measured values in a longer time window. A sample run of our ‘speed’ algorithm is shown in Fig. 7. For this algorithm, the desired cps value has to be given as a parameter.

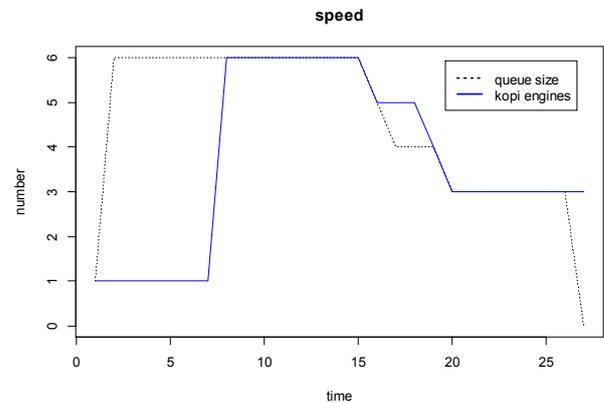


Fig. 7. Processing documents using a processing speed based scaling

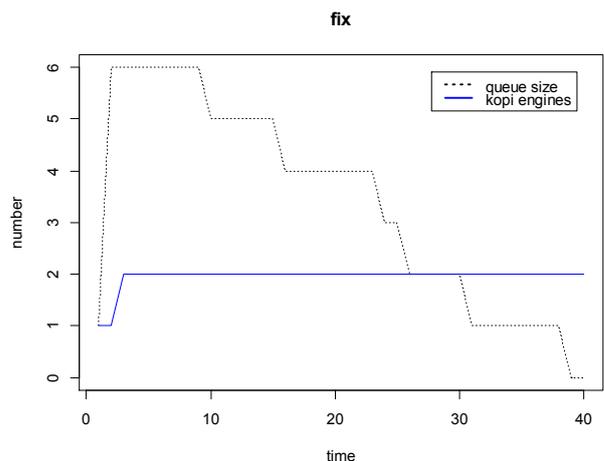


Fig. 8. Processing documents with a fix number of KOPI Engines

For comparison, we also provide a method without any scaling, which runs a given number of KOPI Engines (Fig. 8).

Samples taken from the usage statistics of the real service were used to test and tune the scaling algorithms. Fig. 9 shows a comparison of the implemented scaling algorithms on a sample of 14 documents with a processing time of roughly 1.5 hours. The time label means the total completion time of the whole sample queue in seconds. The cost is calculated as the sum of the cores used per minute. The maximum difference between the completion times is around 10%, while the maximum difference in costs is around 11%. Further experiments showed that the winner varies greatly based on the usage pattern.

For example, Fig. 10 presents another comparison of the scaling algorithms on a different 30 minutes long test run. On this shorter period there is a clear winner in speed: the greedy algorithm, which is 33% faster than the slowest solution running a single fixed KOPI Engine. It should be noted, that winner in cost category is the “fix 3” algorithm, although the greedy algorithm has just slightly more cost. The second sample proves that in some cases there can be large differences in the performance of the scaling algorithms. It requires further research to approximate the optimal scaling algorithm depending on the current queue characteristics. On one side this depends on the future of the queue, which is quite unpredictable as it does not match any of the usual workload conditions (on-off etc.). On the other hand, the samples suggest that on the long run the time or cost differences become quite small.

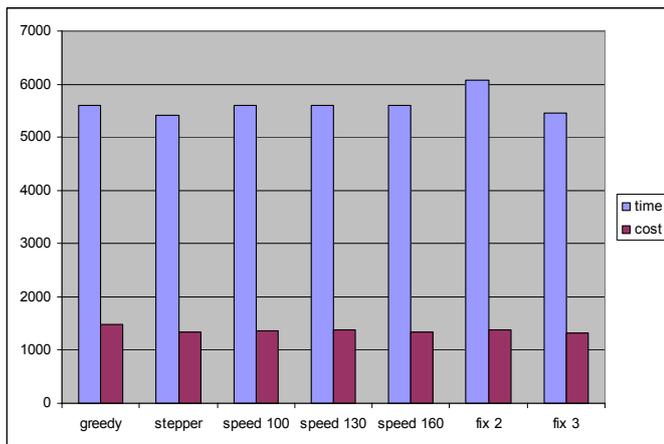


Fig. 9. Comparison of various scaling algorithms on a 1.5 hours sample

The technique used for scaling is closest to threshold-based techniques such as RightScale [5], where the thresholds are not fixed, but calculated from queue statistics. The increment and decrement steps are not fixed either. The waiting period after a scaling operation is naturally ensured by waiting for the scaling operation to finish. Our technique has similarities with control theory using a feedback controller [6]. The stepper, greedy and speed algorithms implement typical fixed gain controllers, where the current state (queue size, processing speed, number of engines working, etc.) is used to generate control actions to the system with the goal of maintaining selected QoS parameters of the system. Control theory also gives us future

directions for improvement such as adjusting controller tuning parameters on-the-fly or switching between scaling algorithms dynamically (reconfiguring control). We did not use any predictive or learning approaches [7] such as time-series analysis or reinforcement learning, because we think the usage of our service is largely unpredictable, and it lacks fix patterns as well. Furthermore, if we expand our service to many new customers, then previously collected patterns or prediction models become obsolete with great probability.

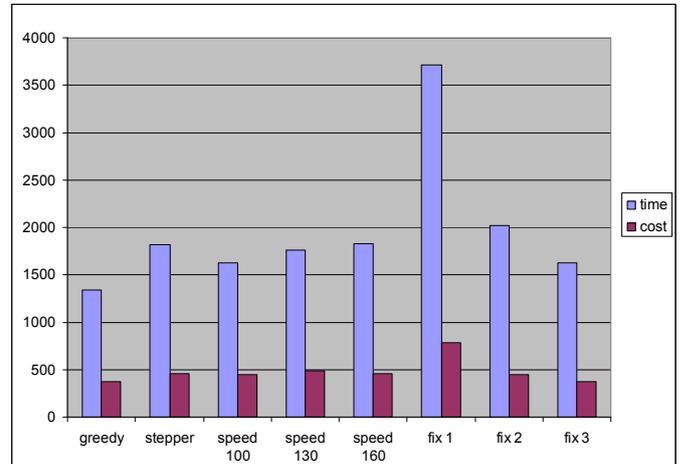


Fig. 10. Comparison of various scaling algorithms on a 0.5 hours sample

V. FAULT TOLERANCE

The engines processing documents periodically report their status to the document queue. However, this report may come quite infrequently as the engine may wait for 3-6 minutes to get reply from an overloaded fulltext cluster. Overall, it can take 10-15 minutes until we can surely state that there is a problem with the processing of a certain document. If the problem is corrected earlier and the engine is stopped, we loose all previous processing done on the document, so even 30-40 minutes of processing work can be lost.

After detecting the problem, we still don't know the root of the problem, which can be quite many. Therefore a continuous state checking was added to the experiment controller, which goes over each running component and checks its correct operation. The checking process can detect and correct the following errors:

- The KOPI engine failed or it is not accessible,
- The document checking process died on the KOPI Engine,
- The addresses of Fulltext Clusters are not properly set in the KOPI Engine,
- The Fulltext Aggregator failed or it is not accessible,
- The aggregator process on the Fulltext Aggregator node is not running,
- The addresses of fulltext engines are not properly set in the aggregator,

- A Fulltext Engine failed or it is not accessible,
- The query engine process is not running on a Fulltext Engine,
- The query engine process is running, but provides incorrect answers on a Fulltext Engine.

The checking period can be freely set, and the check may take 0.5 to 7 minutes depending on the number of components and the load of used cloud environments.

VI. CONCLUSIONS

We presented the solutions and measurements of automatic scaling experiments of a real service in a cloud federation testbed. We will be able to exploit the know-how and also concrete code parts in the real service.

The result of the scaling experiments with KOPI helps us to provide better response times for the service, so that we can avoid situations when days are needed to process waiting documents. Furthermore, we can estimate the time needed to finish documents, and thus we can also influence the cost and quality of our service by automatic scaling.

The availability of the service can also be increased with the new fault tolerance mechanism which automatically checks and replaces or restarts failed components.

.All these results help us in further growing our user base and establish new business relationships.

ACKNOWLEDGMENT

This work has been supported by BonFIRE, an EC supported 7th Framework Programme ICT project (FP7-257386).

REFERENCES

- [1] KOPI Online Plagiarism Search Portal, <http://kopi.sztaki.hu>
- [2] M. Pataki, "A new approach for searching translated plagiarism", 5th International Plagiarism Conference, 16-18 July 2012, Newcastle, UK
- [3] EC FP7-ICT BonFIRE Project, <http://www.bonfire-project.eu/>
- [4] T. Lorigo-Bostrán, J. Miguel-Alonso, J. A. Lozano, "Auto-scaling Techniques for Elastic Applications in Cloud Environments", UPV/EHU Technical Report: EHU-KAT-IK, 2012.
- [5] RightScale Cloud Management, <http://www.rightscale.com/>, 2012
- [6] T. Patikirikoralala, A. Colman, "Feedback controllers in the cloud", 17th Cloud workshop at Asia Pacific Software Engineering Conference (APSEC 2010), 2010.
- [7] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, L. Kovacs and R. M. Badia, "Semantic Resource Allocation with Historical Data Based Predictions", in proceedings of the IARIA First International Conference on Cloud Computing, GRIDs, and Virtualization, 2010.