J Grid Computing (2011) 9:49–64
DOI 10.1007/s10723-010-9178-4

Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients

Eddy Caron · Frédéric Desprez · Adrian Muresan

Received: 28 February 2010 / Accepted: 15 December 2010 / Published online: 6 January 2011
© Springer Science+Business Media B.V. 2011

Abstract The Cloud phenomenon brings along the cost-saving benefit of dynamic scaling. As a result, the question of efficient resource scaling arises. Prediction is necessary as the virtual resources that Cloud computing uses have a setup time that is not negligible. We propose an approach to the problem of workload prediction based on identifying similar past occurrences of the current short-term workload history. We present in detail the Cloud client resource auto-scaling algorithm that uses the above approach to help when scaling decisions are made, as well as experimental results by using real-world Cloud client application traces. We also present an overall evaluation of this approach, its potential and usefulness for enabling efficient auto-scaling of Cloud user resources.

Keywords Cloud computing · Auto-scaling · Workload prediction

1 Introduction

The evolution of IT software services in the direction of Cloud Computing took a step forward in the efficient use of hardware resources through virtualization. In classical Grids or service oriented platforms, users receive a static amount of hardware resources that they make use of. In contrast to this, the Cloud approach consists in offering on-demand virtualized resources to its users. Because virtual resources can be added or removed at any time during the lifetime of the application hosted on a Cloud, the possibility of dynamic scaling arises. Even more, dynamic application resource scaling can be easily automated either at Cloud provider level or at Cloud client level through the use of the Cloud provider's APIs.

To take full advantage of the benefits of dynamic application resource scaling, a Cloud client (user or middleware) needs to be able to make accurate decisions on when to scale the application resources up and down to achieve good performance. These scaling decisions are influenced by several aspects as for example virtual resource setup time or the migration of existing processes to free resources, but resource usage has the biggest impact on the decision.

The idea of self-similarity in web traffic is not new [4]. Based on this, a new Cloud client application resource auto-scaling strategy can be elaborated. By identifying resource usage patterns that have occurred in the past and have a high similarity to the present resource usage pattern, a decision can be made as to the necessity and/or direction of scaling for the present situation. The

E. Caron · F. Desprez · A. Muresan (✉)
LIP Laboratory,
UMR CNRS—ENS Lyon—INRIA—UCB,
University of Lyon, Lyon 5668, France
e-mail: adrian.muresan@ens-lyon.fr

current work presents an approach to the resource usage prediction problem based on identifying past resource usage patterns that are similar to the present use of the system. We present an efficient algorithm for identifying the patterns by using an approximate matching approach.

In Fig. 1 we have a generic Cloud system usage model to have a top-level view on the role of the prediction model. As part of a Cloud client's resource management module, the prediction module uses the Client's resource usage history to try and make an intelligent guess on short-term resource demands. This alone does not constitute the Client's scaling decision as there are a number of other relevant factors that should be taken into consideration like the migration of currently running tasks from virtual resources that need to be terminated. In our current work, we are focusing only on resource usage prediction. The impact that other factors have on the scaling decisions of a Cloud client is an interesting topic of research, yet it is beyond the scope of our current work. The main contribution of the current work is the elaboration of a resource usage prediction algorithm based on pattern matching with the goal of aiding Cloud clients in making automatic application resource scaling decisions.

The rest of this article is organized as follows. Next section presents in more details the context and motivation of our work. In Section 3 contains an overview of existing approach given in the literature. Then, Section 4 presents our algorithm and its key design principles. Finally, before a conclusion and a description of future work, Section 5 presents our experimental results using actual Cloud traces.

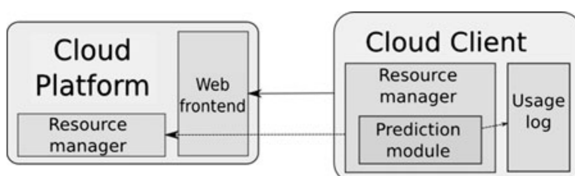


Fig. 1 The role of the prediction component in a generic model of a Cloud system usage scenario

2 Context: the Case for Auto-scaling

In order to understand the motivation of the current work it is important to point out modern approach related to resource provisioning.

In the last few years there has been an ever increasing movement towards dynamic resource provisioning. Recent technical innovations have made this possible by defining abstractions for resources that are disconnected from the physical resources they represent. Resource abstractions are represented by virtualized resources. These virtualized resources have characteristics that are, to some degree, independent from their physical counterparts, the most important of which is their dynamic nature. It is this that opens the door to changing on-the-fly the amount of resources that a platform uses, i.e. dynamic provisioning.

The dynamic provisioning paradigm is inherent in what are currently called Cloud providers under the name of elastic computing. From a client point of view, the elastic computing paradigm can be used by any Cloud client whose programming model exhorts resource usage fluctuations. In this category one can find most types of web applications/services that have a fluctuating number of users over time, like news providers, email providers, multiplayer online games and in general any application that uses a fluctuating number of resources over time. What does not fit in this category are applications that have a minimum user interaction. Consider the execution of a single long-running computation-intensive task. In this situation the number of needed resources is typically known from the beginning of the execution and does not require re-adjustments over the running time of the task.

When a Cloud client achieves a more efficient use of his resources through dynamic provisioning, he saves expenses. It is the cost reduction that drives Cloud clients towards automatic strategies for scaling the number of resources up or down depending on the platform usage and current demands.

The main problem with dynamic provisioning is the fact that new resources are not obtained

instantaneously, they have a start-up time that is not negligible. In essence, the Cloud client has two possibilities of addressing this problem if on-demand provisioning is applied: either delay handling client requests until resources become available or simply drop client requests. In either case, the quality of service of the Cloud client is reduced significantly. To address this problem, current strategies use a predictive approach in hopes of getting an insight into future platform usage and, as a consequence, being able to make scaling decisions ahead of time, compensating for the resource start-up time.

3 Related Work

There are currently two main approaches to facilitate the auto-scaling decisions of Cloud clients as a result of resource usage. The first approach treats the past server usage as a predictable sequence and constructs a mathematical model around it. As a result, the next value of the request sequence is obtained by evaluating the obtained model at the next time point. In other words, a prediction model is built by considering past resource usage. The second approach is a reactive one, based on the current server load and auto-scaling rules that are set up by a human operator (usually a Cloud client). This approach has been often referred to as the "Elasticity rules" approach or the Service Level Agreement (or "SLA") approach [6].

In [11], a description and comparison of three different auto-scaling algorithms is given: Auto-Regression of order 1 (AR1), Linear Regression, and the Rightscale algorithm. The AR1 algorithm belongs to the first category of auto-scaling algorithms. Its approach consists in using a finite history window and identifying appropriate parameters so that a recurring sequence can be obtained and therefore used to calculate the next values. The parameters obtained are adapted as the window slides along the time axis. The Linear Regression algorithm belongs to the first category and calculates a polynomial approximation of the history of requests. The predicted value is then obtained by evaluating the polynomial at a higher point along the time axis. The Rightscale algorithm belongs to the second category, being a version of threshold-based auto-scaling. Its approach consists in using a democratic voting system based on the current server load. Each virtual machine owned by the Cloud client has a vote based on its current load level and two thresholds: low threshold that corresponds to a "scale down" vote (with a default value of 30% system usage) and a high threshold that corresponds to a "scale up" vote (with a default value of 85% system usage). The votes are collected by a central machine and the majority decides the scaling decision for the whole platform. The three algorithms have been put side-by-side and compared by a metric proposed in the same article. Their performance is considerably high.

A more complex form of SLA-based dynamic provisioning can be described by using elasticity rules that dictate what part of the Cloud client needs to scale, in which direction and by how much. In [6], we find such an example with threshold-based rules. This is done by means of an extension to the OVF (Open Virtualization Format), an interoperable, platform and vendor neutral, open format that is used to describe VAs (Virtual Applications). VAs are preconfigured software stacks consisting of one or several Virtual Machines with the purpose of offering self-contained services. The OVF document is actually an XML document containing the description of the OVF package. The elasticity rules come as an extension of this document. They have three components: an associated name, a trigger condition based on the defined key performance indicators and an associated action that represents the implementation of the rule in the form of instantiating new components of the VA or removing existing component instances. Like the Rightscale algorithm, this approach is also a reactive one. Scalability rules have the benefits of combining the high performance of threshold-based algorithms such as Rightscale with tune-ability and therefore have been widely used in practice in commercial Clouds.

The ideas of workload prediction and workload modeling are by no means new, in fact they have been active areas or research in the field of Grid computing.

In [15], a Decentralized Online Clustering model is described and proposed for automatic workload provisioning for enterprise Grids and Clouds and addresses their distributed nature. In this approach a workload prediction algorithm is used and integrated into the system to model the application dynamics. More specifically, a quadratic response surface model is used.

A non-linear model for Grid workload prediction can be found in [5]. The authors propose a prediction model as a series of finite known functional components, usually taken from the sigmoid function class, with unknown coefficients. The coefficients are determined by using using the least square approximation method on a training set. The training set can be split into a training partition and an evaluation partition. This way an early stop strategy can be applied to avoid data over-fitting. Their model has been tested on a 3D image rendering set of tasks based on the Blue Moon Rendering Tool. The error of their prediction is less than 14% with an average of 7.5%.

In [14], we find a real-time resource provisioning system for massive multiplayer online games based on a predictive usage model. The application is dynamically provisioned on a Grid environment. The authors propose a predictive model based on neural networks as this approach has more predictive power than simpler approaches like exponential smoothing, yet is faster in terms of runtime than more complex approaches like auto-regressive models, integrated models or moving average models. The neural network is prepared with two offline phases that include gathering of training samples and using them to train the neural network. As results of experimentation, the neural network approach has prover to have a greater accuracy when compared to the other tested prediction methods: average, moving average, last value and exponential smoothing. The obtained prediction error during the experiments has a maximum value of 33% and a minimum value of 4.94%.

The EMPEROR Grid meta-scheduler [1] uses prediction for estimating a host's load and memory usage and in consequence to have an estimation of the running time of a task. The prediction approach is based on time series analysis techniques. They have used different prediction models including Auto-Regression (AR), Auto-Regressive Integrated Moving Average (ARIMA) and Auto-Regressive Fractionally Integrated Moving Averages (ARFIMA). Their experiments yield that AR and ARFIMA have the best results, depending on the dynamic characteristic of the load traces that are being predicted.

Another prediction approach that uses semi-Markov Process model for estimating resource availability in a Fine-Grained Cycle Sharing system (FGCS) can be found in [16]. Their approach uses resource monitoring on each physical machine and in practice has proved to have more accurate and faster results than time series approaches. Statistical information is used to determine the parameters of the semi-Markov Process model.

The Network Weather Service (NWS) [19] is a distributed, generalized system with the goal of offering short-term predictions for network and computational resources based on historical performance. For forecasting, a time series approach is used with different prediction models. An adaptive approach is used to determine the best prediction model for each measure that is being predicted. Experiments have shown that the predictive performance of the NWS system are at least as good as the predictive performance of the state of the art [18]. The forecast performance of the NWS has been improved when using a support vector machine approach, by using support vector regression [13]. This approach uses a supervised learning technique and its performance are more pronounced when the depth of the prediction set increases.

Prediction has also played a role in generating synthetic workloads of Grid platforms. In [9], we find a fine-detailed study on the topic of Grid performance evaluation by using synthetic workloads obtained from the modeling of Grid workloads. The work describes performance metrics

useful to evaluate Grid environments. These are composed of traditional performance metrics that are time, resource or system related and Grid-specific related to workload completion or failure metrics. The paper continues by describing the specifics of Grid workload modeling. These include user group modeling that underlines the importance of taking into consideration statistics for all jobs on one hand and statistics for each user in particular on the other hand, based on his (or her) past actions. The paper also describes submission patterns that arise in Grid environments and enumerate some of the current approaches of modeling them that include combining Poisson distributions for daily patterns or by using a polynomial function of degree eight. The authors argue that these pattern modeling approaches may not hold as they are indifferent to workload inter-dependency. The authors continue by presenting the GrenchMark synthetic Grid workload generation, execution and analysis framework [7]. They also present extension suggestions to the framework that would make the framework be a better tool for workload generation and analysis.

In [8], we find an integration effort of a Grid application development toolkit named Ibis [17], a Grid co-scheduler named Koala [12] and the GrenchMark [7] synthetic Grid workload generator with the purpose of providing an end-to-end workload generation and testing framework. The authors argue about the benefits that experimental testing of Grid systems has over an analytical or simulated test model. The authors also argue in favor of using synthetic Grid workloads over real Grid workloads or benchmarking approaches. Next the authors describe their integration proposal of building applications with the Ibis toolkit, generating and submitting synthetic workloads with GrenchMark, and then scheduling them with Koala so that the results can be analyzed with GrenchMark again. As result of experimentation they concluded that workloads generated in GrenchMark can cover a wide range of run characteristics.

A shortcoming of existing reactive approaches is the fact that they are blind relative to the trend of the workload they are predicting as they do not consider the recent workload states for their results. In contrast, predictive approaches do consider trend, but the presented approaches that are using mathematical models do not consider arbitrarily-repetitive self-similarities. It is this motivation that led us to the current work.

4 String Matching Based Scaling Algorithm

4.1 The String Matching Concept

The usage of a Cloud client can sometimes have a repetitive behavior. This can be caused by the similarities between tasks that the Cloud client is running or the repetitive nature of human behavior. Given the self-similar nature of web traffic, it follows that current usage patterns of online services have a probability of having already occurred in the past in a very similar form. Therefore we can infer what the system usage will be for a Cloud client by examining its past usage and extracting similar usages.

The pattern strategy has two inputs: a set of past Cloud client usage traces and the present usage pattern that consists of the last usage measures of the Cloud client. Cloud clients working in the same application domain have a higher similarity in resource usages. Due to this similarity it follows that the most relevant historic resource usage data that can be used comes from Cloud clients working in the same application domain. Therefore it would make sense to isolate historical data based on application domains before usage.

The present usage pattern of the Cloud client is used to identify a number of patterns in the historical set that are close to the present pattern itself. Identified patterns should not be dependent on their scale, just on the relation between the elements of the identified pattern and the pattern we are looking for. The resulting closest patterns will be interpolated by using a weighted interpolation (the found pattern that is closest to the present pattern will have a greater weight) and will have as result an approximation of the values that will follow after the present pattern. In essence, the

usage of the Cloud client is predicted by finding similar usage patterns in the past or in other usage traces.

The problem of finding a pattern inside an array of data that is very similar to a given pattern is close to the problem of string matching. The approximate string matching problem has been widely studied especially in its relation to bio-informatics problems, yet it is considerably different from the problem we are addressing.

One definition for the approximate string matching problem is the following. Given a text string $T = t_0t_1...t_n$ and a pattern $P = p_0p_1...p_m$ find a substring of consecutive characters from T call it $T_{i,j}$ that has the smallest edit (or Levenshtein) distance as possible [2].

The edit distance is defined as the number of simple string operations (insert, delete, replace and sometimes exchange) that need to be performed on the identified text substring to have equality to the pattern. The operations can have the same or different weights, depending on the problem needs. The identified match can have any length because of the possible insert and delete operations.

For the problem that we are addressing, the edit distance cannot be applied as we are not comparing string character values, but floating point values. We are interested in identifying sub-arrays of the same, or very close, length and whose floating point absolute value difference is as close as possible to zero. An insertion into or deletion from the identified sub-array would have a great impact on the floating-point difference.

We shall now describe the problem of string matching and its relation to the problem that the current work addresses, as well as our proposal for the approximate variant that is relevant to our problem.

4.2 The String Matching Algorithm

There are several solutions to the string matching problem. We have chosen the Knuth-Morris-Pratt (abbreviated KMP) algorithm as its performance are good (as described in [3]).

Despite the great similarities, our own pattern matching problem has some particularities of its own:

1. an approximate matching is needed since the odds of finding an identical pattern to the one we are looking for are considerably low;
2. matches which are too dissimilar either on small intervals or as a whole need to be discarded;
3. when comparing the pattern to the matching data, scale also needs to be taken into consideration. To be more exact, the scale of the pattern and the scale of the possible match should not affect the comparison, therefore it needs to be scale-independent.
4. The resulting matches are interpolated having different weights on the final result, based on their similarity to the identified pattern.

In order to do an approximate matching, the original KMP algorithm needs to be changed in the content of both functions, therefore they need to be modified accordingly.

Two types of approximation errors are used for the matching:

1. an instant error which dictates the amount by which the current match is allowed to differ from the pattern by comparing in smallest possible units. In our pseudo-code, this is returned by the *Distance*() function.
2. a cumulative error that characterizes the amount by which the current match is allowed to differ from the pattern as a whole. This is basically a sum of the instant errors of the whole matching and is returned by the *CumulativeDistance*() function in our pseudo-code.

The distance between the pattern we are trying to match and a candidate pattern should be computed in a scale-independent manner by first normalizing the two pattern values to a common scale. To decrease floating point approximation errors, one can choose a distance computation that does not use divisions and therefore calculating only on integer values.

As an example consider that the pattern is an array containing the values: 20 , 38 , 21 and the candidate match contains the values: 42 , 81 , 39 . In this form we cannot compare the two patterns. A first idea would be to normalize both arrays to a floating point [0..1] interval and then compare. Working with floating point numbers can be avoided by working with big integer numbers. To reach a common scale we simply multiply each array by the scale of the other. For the scale of each array we can simply consider the first element. As a result, the pattern array is multiplied by the scale of the candidate (this is 42) and the candidate is multiplied by the scale of the pattern (which is 20). In this new situation, comparing two components of each array is done simply by subtraction. The instant error is used here to assure that there are no two components that differ two much (in percentage) from the two arrays.

Once the comparison is done, the identified candidate is stored along with its total distance from the pattern. This facilitates the significance of the result, as the candidate that is closest to the pattern has a higher weight in final result.

The pseudo-code for computing the instant error is illustrated Algorithm 1 in the *Distance*() function.

Algorithm 1 Distance(PatternElement, PatternScale, DataElement, DataScale)

```
return
    PatternElement × DataScale
    - DataElement × PatternScale
```

The cumulative error is obtained by summing up the instant errors from all the elements of the pattern and candidate. This is illustrated in the *CumulativeDistance*() function.

4.3 KMP Modification

The prefix calculation function is changed as described in Algorithm 3. The scales of the two components compared are represented by the first value of each component. This is arguable, but in practice we have achieved good results with this approach. In the function, *scaleK* represents the scale of the prefix and *scaleQ* represents the scale of the post-fix of the pattern. The *Distance*() function returns an appreciation of the distance between two different pattern instances, each having a different scale which is passed as parameter. The comparisons on lines 9 and 14 guarantees that the current instant distance does not differ by more then the acceptable error (in percentage) from the actual pattern that we are matching.

Algorithm 2 CumulativeDistance(P, T, DataOffset)

```
1: patternScale ← P[0]
2: dataScale ← T[DataOffset]
3: length ← length(P)
4: distance ← 0
5: for index ← 0 to length do
6:     distance ← distance + | dataScale × P[index] −
       patternScale × T[index + DataOffset] |
7: end for
8: return distance
```

Algorithm 3 Calculate-prefix-approx(P, ACCEPT_INST_ERR)

```
1: m ← length(P)
2: π[0] ← −1
3: k ← -1
4: scaleK = P[0]
5: scaleQ = P[1]
6: for q ← 1 to m − 1 do
7:     dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
8:     maxDistance ← ACCEPT_INST_ERR ×
       scaleQ × P[k+1]
9:     while k > −1 and dist > maxDistance do
10:        k ← π[k]
11:        dist ← Distance(P[k+1], scaleK, P[q],
           scaleQ)
12:        scaleQ = P[q − (k+1)]
13:    end while
14:    if dist ≤ ACCEPT_INST_ERR × scaleQ ×
       P[k+1] then
15:        k ← k+1
16:    end if
17:    π[q] ← k
18: end for
19: return π
```

In the comparison on line 14, the *scaleQ* term represents the scale of the data. It is needed in order to bring the pattern to the same scale as the data.

The matching algorithm is changed as described in Algorithm 4. Compared to the original KMP algorithm, the main difference is the use of the instant and cumulative distances as a mean of filtering out potential matches that are too different either on small time intervals or as a whole.

On lines 10 and 16 we guarantee that the instant distance between the identified candidate and the pattern is no more than what the acceptable error permits. In order to guarantee a correct comparison, the pattern term needs to be scaled to the same size as the data, hence the *scaleT* term is used in the comparison. Filtering by cumulative distance is done in lines 20 to 24. The *CumulativeDistance*() function returns a sum of instant distances for every instant of the two compared arrays. The running time of this function is $\Theta(m)$ where m is the length of the arrays, which in our case is always equal to the length of P. Line 22 of the algorithm assures that the cumulative distance of the candidate does not differ more than is accepted by the cumulative error from the pattern itself. The pattern itself is represented by the *patternSum* term in the comparison. This is a sum of all the terms in the pattern and should be calculated only once, at the beginning of the algorithm. The pattern sum needs to be brought to the same scale as the candidate sequence and therefore the *scaleT* term is used. Filtering by an acceptable cumulative error that is smaller or equal to the acceptable instant error is useless. This conclusion is trivial when taking into consideration that the cumulative error is a sum of all the instant errors.

The use of the cumulative error changes the running time of the matching algorithm to $\Theta(n \times m)$ in the worst case, where n is the length of the string to match against and m is the length of the input pattern.

Algorithm 4 KMP-approx(T, P, ACCEPT_INST_ERR, ACCEPT_CUMUL_ERR)

```
1: n ← length(T)
2: m ← length(P)
3: π ← Calculate-prefix(P)
4: q ← −1
5: scaleP = P[0]
6: scaleT = T[0]
7: for i ← 0 to n − 1 do
8:    dist ← Distance(P[q+1], scaleP, T[i], scaleT)
9:    maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
10:   while q > −1 and dist > maxDist do
11:      dist ← Distance(P[q+1], scaleP, T[i], scaleT)
12:      q ← π[q]
13:      scaleT = T[i − (q+1)]
14:      maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
15:   end while
16:   if dist ≤ maxDist then
17:      q ← q+1
18:   end if
19:   if q = m−1 then
20:      dist ← CumulativeDistance(P, T, i − m + 1)
21:      maxDist ← ACCEPT_CUMUL_ERR × patternSum × scaleT
22:      if dist ≤ maxDist then
23:         StoreSolution(dist / scaleT, i − m + 1)
24:      end if
25:      q ← π[q]
26:      scaleP = P[q+1]
27:      scaleT = T[i − (q+1)]
28:   end if
29: end for
```

4.4 Interpolating the Found Values

Once approximate matches have been found, the problem of obtaining a relevant result from those matches is raised. Each match should have a contribution to the final result that is proportional to its relative distance to the pattern with respect to the other identified patterns. This corresponds to a weighted sum of the identified matches, where weights are calculated by considering the distance of the current match to the pattern and to the rest of the matches.

Once the weights are calculated, the interpolation is performed between the following L elements after each approximate match. The result is a predicted sequence of length L.

4.5 Algorithm Parameters

The algorithm accepts a number of parameters used for fine-tuning in accordance to each use-case. These parameters are:

- The maximum number of matches (called closest neighbors) to take into consideration (denoted K).
- The length of the predicted sequence (denoted L).
- The acceptable instant error representing the amount by which the identified sequence is allowed to differ on the smallest possible interval lengths from the pattern we are looking for.
- The acceptable cumulative error which represents the amount by which the identified sequence is allowed to differ as a whole from the pattern we are looking for.
- The input set of data representing the database of past requests.
- The input pattern representing a sequence with the last period of requests received.

The first parameter is not independent of the others. It is actually influenced considerably by the acceptable errors. The correlation is strong and can be expressed very easy: the larger the acceptable error, the more matches the algorithm identifies, but the more irrelevant they will be.

Calculating the Acceptable Errors The value of the acceptable errors can be calculated based on the maximum number of neighbors that we wish to find. The approach for this is to use a binary search to zone in on the appropriate values for the acceptable errors.

By using the binary search approach, we have obtained values that have proved to be good in practice. We have used a lower bound of 20% of K for a minimum of identified neighbors and 90% of K as the upper bound for maximum number of identified neighbors.

Calculating the Appropriate Pattern Length The length of the pattern that represents the last traces of server usage has a great impact on the results of the algorithm. Finding the appropriate length is a problem on its own as we have a trade-off between patterns of big lengths that yield a small number of similar candidates, that might be to small in order to be usable, and patterns of small lengths, that find more candidates but they tend to be more irrelevant to our current situation.

We have taken two approaches to this problem. The first approach is to find the most lengths of the most frequent repetitive patterns and use the same length as input to the prediction algorithm.

We have the following constructive approach to identifying the length of the most frequent repetitive patterns:

1. find all similar patterns of length 2 in the historic data
2. take all similar patterns of length 2 and try to match the next element too. This yields all similar patterns of length 3.
   ...
3. take all patterns of length n and try to match the next element too. This yields all similar patterns of length $n + 1$.

The result is that the number of identified similar patterns decreases as the length of the patterns increases:

$$count[n+1] \leq count[n] \leq \ldots \leq count[3] \leq count[2]$$

The conclusion is that the most frequent patterns are of the ones with length 2. In practice, using a pattern length of 2 would prove to be too short to be considered a valid pattern. It follows that the predicted values would be "polluted" by too many identified patterns that have no connection to the current usage pattern. This means a bad prediction capability.

We need to have a better way for choosing the pattern length, that would give more relevant results and avoid pollution as much as possible.

The length of the pattern should be influenced by the time it takes to service a request on the server. We then have the following possibilities:

- Median/average
  - Representative of most of the requests

- Minimum
  - A large pattern cannot match against a smaller pattern that's half different
  - A small pattern can match against a large pattern that's half different
  - The minimum is very probably close to 0 (testing experiments)
  - A close minimum can be selected (ex. 5–10% from the bottom)

Our experimentation shows that a pattern length that is a minimum or even median of the time it takes to service a request is unusable. In practice we have used the average of the request service time and have obtained good results.

5 Experimental Results

To validate our model we have used real-world traces from one Cloud client platform. In all our experiments, we have used a time unit of 100 s as a discrete step. The predicted traces represent the total number of CPUs used by different jobs running in parallel in the time unit of 100 s. We have focused only on CPU usage as the information of memory usage was not available. Nevertheless, should the information of memory usage be available our approach can also be applied for its prediction.

5.1 Data Sources

We have tested our auto-scaling approach with traces from several Cloud client applications: seven applications deployed on an IBM Cloud platform and another application deployed on Amazon EC2. Each of the applications have its own usage particularities, with main differences in the frequency and amplitudes of changes in their overall usages.

Animoto[1] is a Cloud client application that specializes in automatically-orchestrated videos starting from user-generated content. Their platform usage represents oscillations as per user activity. A plot of number of virtual CPUs over a time period of about 4 months, with time slices of 100 s can be found in Fig. 3d.

IBM Cloud Application Traces that we have used represent seven different applications. They feature a high volatility in use and a relatively low number of concurrent running jobs. Plots of the platform usage traces for all seven Cloud application can be found in Figs. 2 and 3.

5.2 Analyzing the Data Sources

We need to have a better way to choose the pattern length, that would give more relevant results and avoid pollution as much as possible. The length of the pattern should be influenced by the time it takes to service a request on the server.

By analyzing the data sources from Animoto and the IBM platform we have obtained the running time in seconds of each job with the results given in Table 1. The conclusions here are that, for all practical purposes, a pattern length that is a minimum or even a median of the time it takes for a job to be run, is unusable when dealing with servers that have a similar usage to the Cloud applications described above. In practice we have used the average of the request service time and have obtained good results.

5.3 Experiment Setup

All the experiments use the server traces with the same form of input data as described above with time units of 100 s, and resource usage value consisting of the total number of CPUs used across the 100 s A pattern length of 100 time units has been used for all the experiments (this is 100 × 100 s—approximately 2.7 h of server time) and predictions are made for one time unit, this is 100 s, which is a little over 1 min 30 s.

The results are displayed under the form of a set of standard metrics that include minimum, maximum, median, and average percentage and value difference between the prediction and the actual value.

A second set of metrics has also been used that allows the comparison to other existing auto-

[1] http://animoto.com

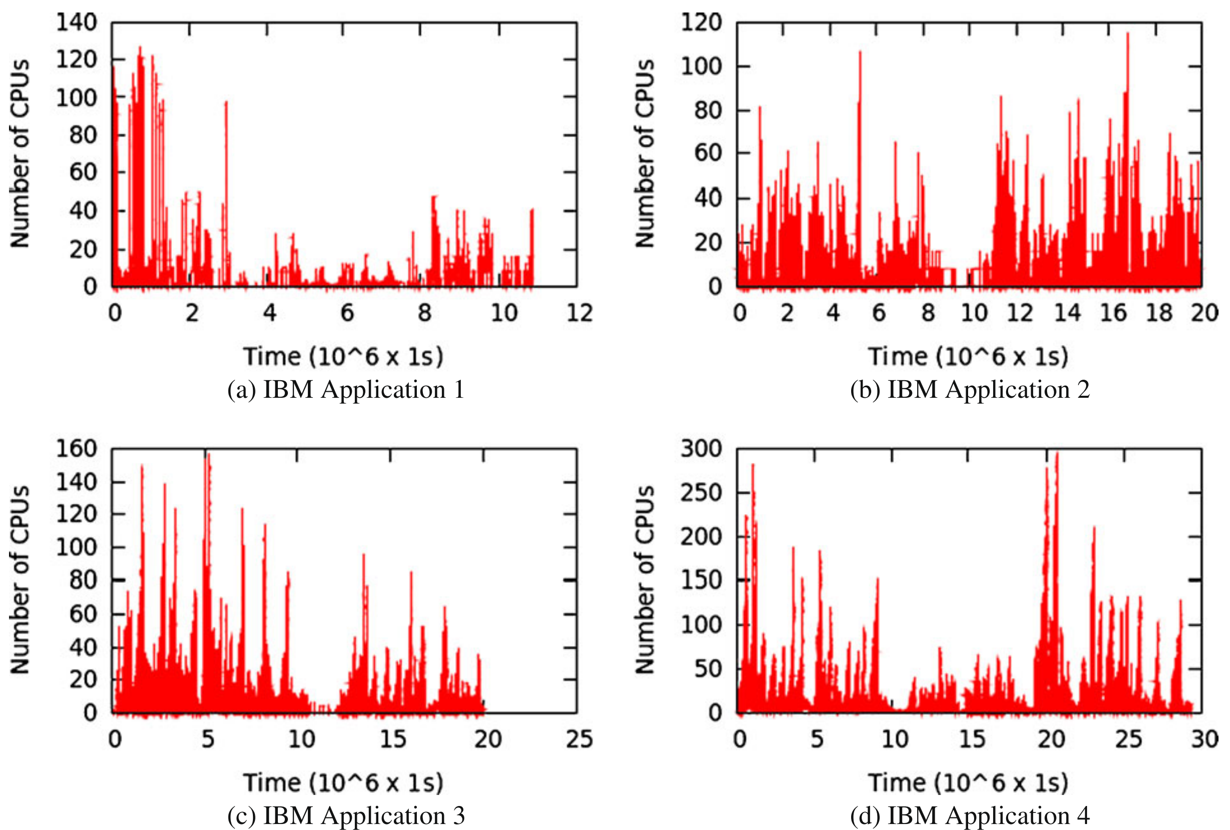


Fig. 2 Plot of number of virtual CPUs used versus time for the first four Cloud application traces

scaling algorithms. This metric was proposed and used by UCSB to compare the performance of three existing auto-scaling algorithms [11]: AR1, Linear Regression and the Rightscale democratic voting algorithm.

In practice, of particular interest is availability when it comes to service providers. A bad scaling up decision results in wasted money for the service provider. A bad scaling down decision results in dropping client requests and as a result a poor quality of service. There is a clear trade-off between the two and service providers usually prefer to favor the latter and as a result have a higher availability, but more wasted money.

We have also measured the average running time necessary for calculating one prediction. This has an impact on the practical usefulness of the prediction since it needs to be subtracted from the prediction time—which is 100 s—to calculate the effective prediction time.

The metric proposed by UCSB is influenced by platform availability and cost by the following formula:

$$\frac{(A_{\log})^{\alpha}}{C} - \frac{\gamma C}{A_{\log}} + \beta \tag{1}$$

where: A represents the availability of the platform and is defined as follows:

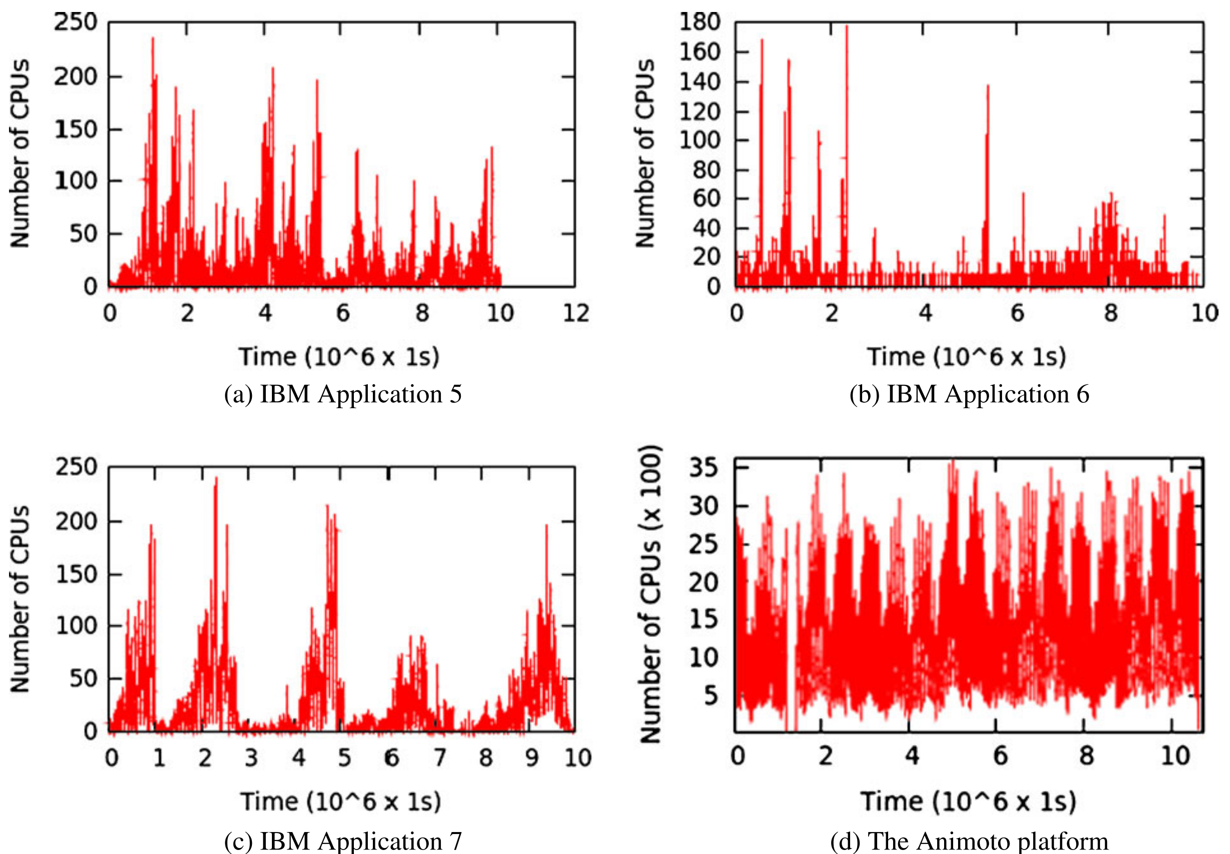
$$A = \frac{\#serviced_requests}{\#of_requests} \tag{2}$$

$A_{\log}$ represents the availability in logarithmic scale and is defined as:

$$A_{\log} = -\log\left(1 + \delta_a - A\right),\ \delta_a < 1 \tag{3}$$

and C represents the cost and is defined as:

$$C = \frac{\#CPU}{hours \times 0.10} \tag{4}$$



(a) IBM Application 5
(b) IBM Application 6
(c) IBM Application 7
(d) The Animoto platform

Fig. 3 Plot of number of virtual CPUs used versus time for the last four Cloud application traces

The constants α, β, γ and δ_a have been chosen through experimentation.

We have used two versions of the metric proposed by the UCSB team: an instant score where we considered resource cost as being charged per fraction of an hour, although this is not the case in current Cloud providers and a second score where we take the maximum prediction over the course of an hour and use that as static provisioning for the whole hour.

In our experimentation we have considered as reference for time and cost comparisons, the `m1.small` virtual machine instance type available in on the Amazon EC2 platform.

5.4 Results

We have done self-prediction test in which a resource trace is used as historic data to predict

Table 1 Job length statistics for the data sources

	App1	App2	App3	App4	App5	App6	App7	Animoto
Avg	13,045	3,746	4,437	4,556	4,734	5,971	7,623	1,296
Min	6	3	4	4	4	6	6	4
Median	10,221	2,865	3,491	3,703	3,844	4,228	6,083	283
Max	129,725	128,999	450,247	534,525	35,185	431,125	46,931	22,452

Values represent time in seconds based on the running time of the recorded jobs

Table 2 Results of self-prediction experiments with traces from the eight data sources: the seven IBM Cloud applications and Animoto

Metric	App1	App2	App3	App4	App5	App6	App7	Animoto
Min prediction error (%)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max prediction error (%)	1.04E8	1.05E8	8.78E7	5.47E7	9.3E7	1.18E8	7.74E7	100.0
Med prediction error (%)	100.0	100.0	100.0	41.58	74.7	100.0	16.8	2.69
Avg prediction error (%)	65.8E3	76.5E3	84.7E3	42E3	90.5E3	20.3E4	44.4E4	5.42
UCSB metric (max per hour)	49.6	49.06	48.91	49.19	47.83	48.81	48.46	−1.39
UCSB metric (instant)	49.5	48.92	48.69	48.86	47.52	48.64	47.81	−18.38
Runtime per instance (ms)	94.19	147.5	40.51	66.51	147.82	30.2	44.2	186.625

The experiments have been done across time slices of 100 s. The last three columns represent comparative results for other algorithms—RightScale, auto-regression of order 1 and linear regression

itself, ignoring exact matches. The results can be seen in Table 2.

For the sake of comparison we can consider results of evaluating three other algorithms obtained in [11]. These values are taken from experimentation against a randomly generated usage pattern and feature the variants of all three algorithms that implement the Smartkill strategy, thus making them more effective. Only measures of the UCSB metric (instant) were available with the following values:

– RightScale: 11.11
– Autoregression of order 1: 17.3
– Linear Regression: 10.8

It is worth noting that there is a considerable difference between the results obtained by using the first seven Cloud applications and the Animoto Cloud application. The error of the prediction results of the former are a lot higher, with median values ranging from 16.8 to 100.0%, when compared to the latter that has average median errors of 2.69%. It is also worth noting that the results obtained by using the UCSB metric are exactly the opposite, with good results for the first seven applications and weaker results when using the Animoto traces. The large difference between the obtained results is caused by the large difference between the number of concurrent users that the applications have: the first seven application have a relatively low number of concurrent users whereas the Animoto platform has a larger number.

The fact that percentage prediction error for the first seven applications is high is caused by the relatively low number of concurrent users which translates into a low number of CPUs used per application. In combination with the high volatility of the platform, this yields small value differences between the actual and predicted usage and high percentage differences between the two.

When compared to other approaches, we have obtained both better and worst results, depending on the volatility of the platform usage and on the algorithm parameters. The first seven Cloud applications from the IBM platform have a higher volatility than the Animoto platform, at the granularity level on which the algorithm works. This leads us to believe that the algorithm can yield better results in practice if its parameters are calibrated accordingly. Of most importance are the pattern length and the statistical relevance of the domain of the historic data with respect to the domain of the platform whose usage is being predicted.

The time necessary for computing one prediction instance has proved in practice to be low

Table 3 The prediction error obtained for various lengths of historic data and pattern lengths for the Animoto platform

Pattern length	Data length (%)			
	100.0	50.0	25.0	12.5
1,000	7.2	8.7	7.5	9.3
500	6.9	7.9	8.6	10.0
100	2.6	2.7	2.8	2.7
50	2.4	2.5	2.4	2.4
25	1.9	2.1	2.1	2.2
12	1.2	1.4	1.5	1.6
2	>100	>100	>100	>100

Table 4 Total virtual machine allocation wait time by using on-demand provisioning and our predictive approach for the eight Cloud applications

Metric (s)	On-demand	App1	App2	App3	App4	App5	App6	App7	Animoto
Min	69	0	0	0	0	0	0	0	0
Avg	82	0	0	0	0	0	0	0	0
Max	126	26.094	26.147	26.04	26.066	26.147	26.03	26.044	26.186

Experiments consist of predicting one platform's usage with the platform's own traces as historic data, across time slices of 100 s

relative to the prediction time, between 30 and 147 ms.

We have experimented with various lengths of the historic data set and of the pattern that is considered for input. The results with the prediction error in each case can be seen in Table 3. Although this does not show that the algorithm yields the best possible results, it does show that there is a clear tendency for the accuracy of the prediction to improve as we increase the size of the historic data and as we find the best pattern length to take into consideration when predicting. The results table illustrates results when varying the pattern length and the length of the historic data used for prediction. We have varied the historic data from 100%—the full set, to 50, 25 and 12.5% of the set. The pattern length has also been varied from 1,000 time units to 500, 100, 50, 25, 12 and 2 time units.

To have a better view of the qualitative performance of an allocation system that uses the current predictive approach, consider the results in Table 4. The table contains resource allocation wait times (in seconds) for instances of type `m1.small` from Amazon EC2. We have used this type of virtual machine as it was also the basis for price calculation in the metric proposed by UCSB, discussed in the previous paragraphs. The on-demand times for provisioning represent the total time needed to obtain a working virtual machine (the sum of the `install` and `boot` time) for single-instance instantiation and are taken from [10]. The allocation time delays for the predictive approach have been obtained from the on-demand times, by subtracting the prediction depth and adding the running time of the algorithm for predicting one time instance. If the obtained value is less than 0, then 0 is presented. For our experimentation we have used time slices of 100 s. It is clear that a predictive approach has a better performance when it comes to resource allocation wait time. To increase the performance, a bigger prediction depth can be used, at the cost of prediction accuracy. In our current experiment we have used a prediction depth of 1.

To evaluate the approach from a practical point of view, we have computed monetary differences between ideal provisioning (on-demand, yet with zero wait time) and the predictive approach we are proposing. As base price we have considered the cost of 1 h on the Amazon EC2 platform for an instance of type `m1.small`. This is 0.095 $ per hour in most availability regions. Results can be seen in Table 5. The actual cost values are influenced by the length of the trace interval and as a result it does not make sense to compare cost values from one platform with cost values from another. On the other hand, the percentage value for false positives can be compared for all platforms. The results of the prediction are influenced

Table 5 Total cost estimation for an ideal resource provisioning (on-demand with zero wait time) and the proposed approach

Metric	App1	App2	App3	App4	App5	App6	App7	A w/A
Ideal ($)	1783.01	500.07	1600.40	3892.04	3365.71	616.25	4030.14	392307.06
Scale up false positives ($)	32.85	6.9	25.51	61.68	48.5	15.63	51.3	9765.34
Scale up false positives (%)	1.84	1.38	1.59	1.58	1.44	2.53	1.27	2.48
Scale down false positives ($)	677.27	252.82	461.11	651.28	688.61	281.74	342.5	3733.95
Scale down false positives (%)	37.98	50.55	28.81	16.73	20.45	45.71	8.49	0.95

by the correlation between the historic resource usage data and the current resource usage. This can be seen in all test cases.

The reader will note that in our experiments we have considered only CPU usage as measure and prediction target. In a Cloud environment, a virtual resource usually has more characteristics associated to it than just CPU power. In particular, memory usage is one of the most notable characteristics. Our approach can also be used to have a prediction of the memory usage if the server traces also contain information about past memory usage. With predictions for both memory and CPU usages, the scaling component of the Cloud client should be able to more accurately decide the characteristics of the virtual resources that are to be instantiated or released. The topic of making a good scaling decision both in direction and in virtual machine characteristics is an interesting topic of research, yet it is beyond the scope of the current work.

6 Conclusions and Future Work

One of the most important benefits of Cloud Computing is the ability for Cloud clients to adapt the number of resources used based on their actual use. This has great implications on cost saving as resources are not paid for when they are not used. Dynamic scalability is achieved through virtualization. The downside of virtualization is that it has a non-zero setup time that has to be taken into consideration for an efficient use of the platform. It follows that an accurate prediction method would greatly aid a Cloud client in making its auto-scaling decisions.

In this work, a new resource usage prediction algorithm is presented. It uses a set of historic data to identify similar usage patterns to a current window of records that occurred in the past. The algorithm then predicts the system usage by interpolating what follows after the identified patterns from the historical data. Experiments have shown that the algorithm has good results when presented with historic data that has a high relevance to the current platform and the quality of results can improve by choosing an appropriate value for the prediction time frame and by increasing the historic data size. The running time of the algorithm has proved negligible in our experiments, which makes it a good candidate for practical use.

As future work directions, we will be looking into ways that a relevant set of historic data can be composed for a particular application domain as well as implementing our approach for an actual Cloud client.

Acknowledgements This work was developed with financial support from the ANR (Agence Nationale de la Recherche) through the SPADES project referenced 08-ANR-SEGI-025. The authors would like to thank Animoto and The Grid Workload Archive for the high-quality workload traces that they provided.

References

1. Adzigogov, L., Soldatos, J., Polymenakos, L.: Emperor: an ogsa Grid meta-scheduler based on dynamic resource predictions. J. Grid Computing **3**, 19–37 (2005). doi:10.1007/s10723-005-9001-9
2. Chang, W.I., Marr, T.G.: Approximate string matching and local similarity. In: CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, pp. 259–273. Springer-Verlag, London (1994)
3. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, chapter 32. String Matching, McGraw-Hill Higher Education (2001)
4. Crovella, M., Bestavros, A.: Explaining World Wide Web Traffic Self-Similarity. Technical Report TR-95-015. Boston, MA (1995)
5. Doulamis, N., Doulamis, A., Litke, A., Panagakis, A., Varvarigou, T., Varvarigos, E.: Adjusted fair scheduling and non-linear workload prediction for QoS guarantees in Grid computing. Comput. Commun. **30**(3), 499–515 (2007) Special Issue: Emerging Middleware for Next Generation Networks
6. Galán, F., Sampaio, A., Rodero-Merino, L., Loy, I., Gil, V., Vaquero, L.M.: Service specification in Cloud environments based on extensions to open standards. In: COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System Software and Middleware, pp. 1–12. ACM, New York, NY (2009)
7. GrenchMark.: http://grenchmark.st.ewi.tudelft.nl (2010). Accessed 10 May 2010
8. Iosup, A., Iosup, R., Epema, D.H.J., Maassen, J., Van Nieuwpoort, R.: Synthetic Grid workloads with ibis, koala, and grenchmark. In: Proceedigs of the CoreGRID Integrated Research in Grid Computing (2005)
9. Iosup, A., Epema, D.H.J., Franke, C., Papaspyrou, A., Schley, L., Song, B., Yahyapour, R.: On Grid performance evaluation using synthetic workloads.

In: JSSPP'06: Proceedings of the 12th International Conference on Job Scheduling Strategies for Parallel Processing, pp. 232–255. Springer-Verlag, Berlin (2007)
10. Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of Cloud computing services for many-tasks scientific computing. IEEE TPDS (accepted in print, 2010)
11. Kupferman, J., Silverman, J., Jara, P., Browne, J.: Scaling Into The Cloud. http://cs.ucsb.edu/~jkupferman/docs/ScalingIntoTheClouds.pdf (2009). Accessed 19 Oct 2009
12. Mohamed, H., Epema, D.: Koala: a co-allocating Grid scheduler. Concurr. Comput.: Pract. Exper. **20**(16), 1851–1876 (2008)
13. Prem, H., Raghavan, N.: A support vector machine based approach for forecasting of network weather services. J. Grid Computing **4**, 89–114 (2006). doi:10.1007/s10723-005-9017-1
14. Prodan, R., Nae, V.: Prediction-based real-time resource provisioning for massively multiplayer online games. Future Gener. Comput. Syst. **25**(7), 785–793 (2009)
15. Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., Sharma, N.: Towards autonomic workload provisioning for enterprise Grids and Clouds. In: Proceedings of the 10 th IEEE/ACM International Conference on Grid Computing (Grid 2009), pp. 50–57 (2009)
16. Ren, X., Lee, S., Eigenmann, R., Bagchi, S.: Prediction of resource availability in fine-grained cycle sharing systems empirical evaluation. J. Grid Computing **5**, 173–195 (2007). doi:10.1007/s10723-007-9077-5
17. van Nieuwpoort, R.V., Maassen, J., Wrzesińska, G., Hofman, R.F.H., Jacobs, C.J.H., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient java-based Grid programming environment: Research articles. Concurr. Comput.: Pract. Exper. **17**(7–8), 1079–1107 (2005)
18. Wolski, R.: Dynamically forecasting network performance using the network weather service. Cluster Comput. **1**, 119–132 (1998). doi:10.1023/A:1019025230054
19. Wolski, R., Spring, N.T., Hayes, J.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Gener. Comput. Syst. **15**(5–6), 757–768 (1999)