

# Identifying Failures in Grids through Monitoring and Ranking

Demetrios Zeinalipour-Yazti, Kyriacos Neocleous<sup>‡</sup>, Chryssis Georgiou<sup>‡</sup>, Marios D. Dikaiakos<sup>‡</sup>

Pure and Applied Sciences, Open University of Cyprus, P.O. Box 24801, 1304 Nicosia, Cyprus

<sup>‡</sup> Dept. of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus  
zeinalipour@ouc.ac.cy, {kyriacos,chryssis,mdd}@cs.ucy.ac.cy

## Abstract

*In this paper we present FailRank, a novel framework for integrating and ranking information sources that characterize failures in a grid system. After the failing sites have been ranked, these can be eliminated from the job scheduling resource pool yielding in that way a more predictable, dependable and adaptive infrastructure. We also present the tools we developed towards evaluating the FailRank framework. In particular, we present the FailBase Repository which is a 38GB corpus of state information that characterizes the EGEE Grid for one month in 2007. Such a corpus paves the way for the community to systematically uncover new, previously unknown patterns and rules between the multitudes of parameters that can contribute to failures in a Grid environment. Additionally, we present an experimental evaluation study of the FailRank system over 30 days which shows that our framework identifies failures in 93% of the cases. We believe that our work constitutes another important step towards realizing adaptive Grid computing systems.*

## 1 Introduction

Grids have emerged as wide-scale, distributed infrastructures that comprise heterogeneous computing and storage resources, operating over open standards and distributed administration control [13, 15]. Grids are quickly gaining popularity, especially in the scientific sector, where projects like *EGEE (Enabling Grids for E-science)* [8], *TeraGrid* [25] and *Open Science Grid* [23], provide the infrastructure that accommodates large experiments with thousands of scientists, tens of thousands of computers, trillions of commands per second and petabytes of storage [8, 25, 23]. At the time

of writing, EGEE assembles over 250 sites around the world with more than 30,000 CPUs and 18PB of storage, running over 25,000 concurrent jobs and supporting over 100 Virtual Organizations.

While the aforementioned discussion shows that Grid Computing will play a vital role in many different scientific domains, realizing its full potential will require to make these infrastructures *dependable*. As a measure of dependability of grids we use the ratio of successfully fulfilled job requests over the total number of jobs submitted to the resource brokers of a grid infrastructure. The FlexX and Autodock data challenges of the WISDOM [30] project, conducted in August 2005, have shown that only 32% and 57% of the jobs completed successfully (with an "OK" status). Additionally, our group conducted a nine-month characterization of the South-Eastern-Europe resource broker (`rb101.grid.ucy.ac.cy`) in [6] and showed that only 48% of the submitted jobs completed successfully. Consequently, the dependability of large-scale grids needs to be improved substantially.

Detecting and managing failures is an important step toward the goal of a dependable grid. Currently, this is an extremely complex task that relies on over-provisioning of resources, ad-hoc monitoring and user intervention. Adapting ideas from other contexts such as cluster computing [21], Internet services [19, 20] and software systems [22] seems also difficult due to the intrinsic characteristics of grid environments. Firstly, a grid system is not administered centrally; thus it is hard to access the remote sites in order to monitor failures. Moreover we cannot easily encapsulate failure feedback mechanisms in the application logic of each individual grid software, as the grid is an amalgam of pre-existing software libraries, services and components with no centralized control. Secondly, these systems are extremely large; thus, it is difficult to acquire and analyze fail-

ure feedback at a fine granularity. Lastly, identifying the overall state of the system and excluding the sites with the highest potential for causing failures from the job scheduling process, can be much more efficient than identifying many individual failures. Of course the latter information will be essential to identify the root cause of a failure [20], but this operation can be performed in an offline phase, and thus it is complementary to our framework.

In the FailRank architecture, feedback sources (i.e., websites, representative low-level measurements, data from the Information Index, etc.) are continuously coalesced into a representative array of numeric vectors, the *FailShot Matrix (FSM)*. FSM is then continuously ranked in order to identify the  $K$  sites with the highest potential to feature some failure. This allows the system to automatically exclude the respective sites from the job scheduling process.

The advantages of our approach are summarized as follows: (i) FailRank is a simple yet powerful framework to integrate and quantify the multi-dimensional parameters that affect failures in a grid system; (ii) our system is tunable, allowing system administrators to drive the ranking process through user-defined ranking functions; (iii) we eliminate the need for human intervention, thus our approach gives space for automated exploitation of the extracted failure semantics; (iv) we expect that the FailRank logic will be implemented as a filter outside the Grid job scheduler (i.e., Resource Broker or Workload Management System), thus imposing minimum changes to the Grid infrastructure.

## 2 Monitoring Failures in Grids

In this subsection we overview typical failure feedback sources provided in a grid environment. These sources contain information that is utilized by our system in order to deduct, in an a priori manner, the failing sites. Our discussion is in the context of the EGEE infrastructure, but similar tools and sources exist in other grids [25, 23].

**Meta-information sources:** Several methods for detecting failures have been deployed so far. Examples include (for a detailed description see [27]): (i) *Information Index Queries*: these are performed on the Information Service and enable the extraction of fine-grained information regarding the complete status of a grid site; (ii) *Service Availability Monitoring (SAM)* [31]: a reporting web site that is maintained for publishing periodic test-job results for all sites of the infrastructure; (iii) *Grid statistics*: provided by services such as *GStat* [17];

(iv) *Network Tomography Data*: these can be obtained by actively *pinging* and *tracerouting* other hosts in order to obtain delay, loss and topological structure information. Network tomography enables the extraction of network-related failures; (v) *Global Grid User Support (GGUS)* ticketing system [9]: system administrators use this system to report component failures as well as needed updates for sites. Such tickets are typically opened due to errors appearing in the SAM reports; (vi) *Core Infrastructure Center (CIC)* broadcasts [5]: allow site managers to report site downtime events to all affected parties through a web-based interface; and (vii) *Machine log-files*: administrators can use these files to extract error information that is automatically maintained by each grid node.

**Active benchmarking:** Deploying a number of lower level probes to the remote sites is another direction towards the extraction of meaningful failure semantics. In particular, one can utilize tools such as GridBench [26, 28], the Grid Assessment Probes [4] and DiPerF [7], in order to determine in real time the value of certain low level and application-level failure semantics that can not be furnished by the meta-information sources. For example, the GridBench tool developed by our group provides a corpus of over 20 benchmarks that can be used to evaluate and rank the performance of Grid sites and individual Grid nodes.

Both the Meta-Information Sources and the Active Benchmarking approaches have a major drawback: *their operation relies heavily on human intervention*. As Grid infrastructures become larger, human intervention becomes less feasible and efficient. As we would like Grid Dependability to be scalable, our proposed architecture does not rely on human intervention but instead provides the means for acquiring and analyzing the data from the above resources in an *automated* manner.

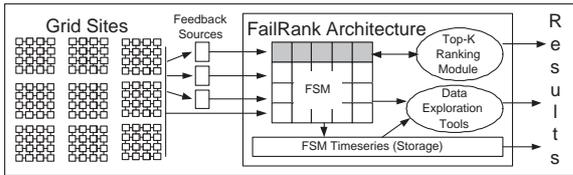
## 3 The FailRank System

In this section we describe the underlying structure that supports the FailRank system. We start out with an architecture overview and then proceed with basic definitions in order to formalize our description. We follow with the description of the failure ranking mechanism deployed in FailRank.

### 3.1 Architecture Overview

The FailRank architecture (see Figure 1), consists of four major components: (i) a *FailShot Matrix (FSM)*, which is a compact representation of the parameters that

contribute to failures, as these are furnished by the feedback sources; (ii) a temporal sequence of FSMs defines an *FSM timeseries* which is stored on local disk; (iii) a *Top-K Ranking Module* which continuously ranks the FSM matrix and identifies the  $K$  sites with the highest potential to run into a failure using a user defined scoring function; and (iv) a set of data exploration tools which allow the extraction of failure trends, similarities, enable learning and prediction. FailRank is tunable because it allows system administrators and domain experts to drive the ranking process through the provisioning of custom scoring functions.



**Figure 1.** The FailRank System Architecture: Feedback sources are continuously coalesced into a representative array of numeric vectors, the *FailShot Matrix (FSM)*. FSM is then continuously ranked in order to identify the  $K$  sites with the highest potential to fail.

### 3.2 Definitions and System Model

**Definition (FailShot Matrix (FSM)):** Let  $S$  denote a set of  $n$  grid sites (i.e.  $S = \{s_1, s_2, \dots, s_n\}$ ). Also assume that each element in  $S$  is characterized by a set of  $m$  attributes (i.e.  $A = \{a_1, a_2, \dots, a_m\}$ ). These attributes are obtained by the feedback sources described in Section 2. The rows in Table 1 represent the sites while the columns represent the respective attributes. The  $j^{th}$  attribute of the  $i^{th}$  site is denoted as  $s_{ij}$ . The  $j$ -th attribute specifies a *rating* (or *score*) which characterizes some grid site  $s_i$  ( $i \leq n$ ) at a given time moment. These ratings are extracted by custom-implemented parsers, which map the respective information to real numerical values in the range  $[0..1]$  (1 denotes a higher possibility towards failure). The  $m \times n$  table of scores defines the *FailShot Matrix (FSM)*, while a *Site Vector* is any of the  $n$  rows of FSM.

A graphical illustration for some synthetic example is given in Table 1. The figure shows five sites  $\{s_1, \dots, s_5\}$  where each site is characterized by five attributes: CPU (% of CPU units utilized), DISK (% of storage occupied), QUEUE (% of job queue occupied), NET (% of dropped network packets) and FAIL (% of jobs that don't complete with an "OK" status).

**Definition (FSM Timeseries):** A temporal sequence of  $l$  FailShot Matrices.

Keeping a history of the failure state for various prior time instances is important as it enables the automatic post-analysis of the dimensions that contributed to a given failure, enables the prediction of failures and others (Section 6 provides an overview). It is important to notice that the FSM timeseries can be stored incrementally in order to reduce the amount of storage required to keep the matrix on disk. Nevertheless, even the most naive storage plan of storing each FSM in its entirety, is still much more storage efficient than keeping the raw html/text sources provided by the feedback sources. In constructing FailBase, described in Section 4, we found that the FSM representation saves us approximately 350GB of storage per month.

### 3.3 The Ranking Module

Although the snapshot of site vectors in FSM greatly simplifies the representation of information coming from different sources, observing individually hundreds of parameters in real time in order to identify the sites that are running into trouble is still a difficult task. For example a typical LDAP query to the Grid Information Service returns around 200 attributes. Monitoring these parameters in separation is a cumbersome process that is very expensive in terms of human resources, can rarely lead to any sort of a priori decision-making and is extremely prone to mistakes and human omissions. Instead, automatically deducting the sites with the highest potential to suffer from failures is much more practical and useful. Since this information will be manipulated in high frequencies, we focus on computing the  $K$  sites with the highest potential to suffer from failures rather than finding all of them ( $K$  is a user-defined parameter). Therefore we don't have to manipulate the whole universe of answers but only the  $K$  most important answers, quickly and efficiently. The answer will allow the Resource Broker to automatically and dynamically divert job submissions away from sites running into problems as well as notify administrators in advance (compared to SAM & tickets) to take preventive measures for the sites more prone to failures.

**Scoring Function:** In order to rank sites we utilize some aggregate scoring function which is provided by the user (or system administrator). For ease of exposition we use, similarly to [2], the function:

$$Score(s_i) = \sum_{j=1}^m w_j * s_{ij} \quad (1)$$

Site	CPU	DISK	QUEUE	NET	FAIL
$s_1$ =USC-LCG2	0.63	0.61	0.01	0.28	0.35
$s_2$ =TAU-LCG2	0.66	0.91	0.92	0.56	0.58
$s_3$ =ELTE	0.48	0.01	0.16	0.56	0.54
$s_4$ =UCL-CCC	0.99	0.90	0.75	0.74	0.67
$s_5$ =KIMON	0.44	0.07	0.70	0.19	0.67

**Table 1.** The *FailShot Matrix (FSM)* coalesces the failure information, available in a variety of formats and sources, into a representative array of numeric vectors.

where  $s_{ij}$  denotes the score for the  $j^{th}$  attribute of the  $i^{th}$  site and  $w_j$  ( $w_j > 0$ ) a weight factor which calibrates the significance of each attribute according to the user preferences. For example if the CPU load is more significant than the DISK load, then the former parameter is given a higher weight. Should we need to capture more complex interactions between different dimensions of FSM we could construct, with the help of a domain expert, a custom scoring function or we could train such a function automatically using historic information (Section 5.3 conducts an evaluation of this parameter). It is expected that the scoring function will be much more complex in a real setting (e.g. a linear combination of averages over  $n'$  correlated attributes, where  $n' \ll n$ ).

**Example:** In order to stimulate our description, consider the example of Table 1. In order to infer the overall rank for two site vectors, such as  $s_2 = \{0.66, 0.91, 0.92, 0.56, 0.58\}$  and  $s_4 = \{0.99, 0.90, 0.75, 0.74, 0.67\}$ , we apply the scoring function with  $w_j = 1$  (i.e. all dimensions are of equal importance), and find that  $s_2 = 3.63$  and  $s_4 = 4.05$ .

In order to minimize the computation of the scoring function, which potentially has to join hundreds of columns in each run, we will utilize the *Threshold Algorithm (TA)* [12]. *TA* is one of the most widely recognized algorithms for finding the  $K$  highest rank answers in database and middleware scenarios. Suppose that we are interested in finding the  $K = 1$  objects with the highest score. *TA* starts out by performing a parallel access to the  $n$  lists of the Sorted-FSM in which each attribute is sorted in descending order. While an object  $s_i$  is seen, *TA* performs a random access to the other lists to find the exact score for  $s_i$  using the given scoring function. In our working example the exact score would be computed for the two objects in the first row (i.e.  $s_4 = 4.05$  and  $s_2 = 3.63$ ) since sorted access is executed on a row-at-a-time basis. It then computes a *threshold* value  $\tau$  as the sum of all scores in the first row (i.e.  $\tau = .99 + .91 + .92 + .74 + .67 = 4.23$ ).

Since  $\tau$  is larger than both scores of  $s_4$  and  $s_2$ , the *TA* algorithm performs another iteration in which the threshold  $\tau$  is refined as the sum of scores across the second row (i.e.  $\tau = 3.54$ ). It also computes the exact score for  $s_5 = 2.07$  (the only unresolved object in the second row). Now the algorithm finds at least  $K=1$  objects above the threshold (i.e.  $s_4 \geq \tau$  and  $s_2 \geq \tau$ ) and therefore terminates. It is easy to prove that no other object can have a score above  $s_4$  thus the score function calculation can be omitted for these objects.

## 4 The EGEE FailBase Repository

In the previous section we outlined the main components of the FailRank architecture. In this section we present the tools we developed in order to evaluate the proposed architecture. In particular, we present the *FailBase Repository* which is a 38GB corpus of state information that we constructed and which characterizes the EGEE Grid for one month in 2007. Such a corpus paves the way for the community to systematically uncover new, previously unknown patterns and rules between the multitudes of parameters that can contribute to failures in a grid environment.

### 4.1 Overview

FailBase currently contains 32 days of monitoring data obtained from tests executed on the EGEE Grid Infrastructure between 16/3/2007 and 17/4/2007. The trace was collected at the High Performance Computing systems Lab (HPCL) at the University of Cyprus. We utilized a dual Xeon 2.4GHz CPU machine with 1GB of RAM connected to the European Academic Network (GEANT) at 155Mbps.

The trace maintains information for 2,565 Computing Element (CE) queues. It is important to note that resource brokers perform the *matchmaking* between the requests and the available and appropriate queues at the CE-queue granularity rather than on individual nodes. Thus, we focus on characterizing failures at the same granularity as well. Each CE-queue is stored in an individual folder that currently contains 72 attributes (i.e., files) and each file characterizes the CE-queue it is stored in. For example, `ce101.grid.ucy.ac.cy_jobmanager-lcgpbs-atlas` is the directory that contains measurements specific to the ATLAS experiment job queue that is maintained on the Computing Element `ce101.grid.ucy.ac.cy`.

Each of the files in the CE-queue folders can be thought of as a timeseries (i.e., a sequence of [times-

tamp,value] pairs) for the given attribute using a time step of approximately 1 to 10 minutes (varies according to the type of source). We currently share the Failbase repository with the researchers of our group using the UNIX filesystem interface which maintains openness and portability. In the future we have plans to store the information in a relational database on the EGEE grid in order to allow researchers from other institutes to access and manipulate the stored information using the expressive power of the Structured Query Language (SQL).

## 4.2 Meta-information Sources

We shall next describe the adopted methodology for acquiring the 72 failure-related attributes from the respective meta-information sources:

(i) *Service Availability Monitoring (SAM)*: We obtained approximately 260MB of data in raw html form (one html file for each CE) using the UNIX system utility *curl*. We then processed these pages using a set of perl scripts and generated 18 attributes. These attributes contain information such as the version number of the middleware running on the CE, results of various replica manager tests and results from test job submissions.

(ii) *Information Index Queries (BDII)*: We used the *ldapsearch* system utility tool to perform approximately 2 million LDAP queries on the Information Index hosted on *bdi101.grid.ucy.ac.cy*. We then performed a projection in order to extract another 15 failure-related attributes. This yielded attributes such as the number of free CPUs and the maximum number of running and waiting jobs for each respective CE-queue.

(iii) *Grid Statistics (GStat)*: We downloaded, again using *curl*, and parsed data files from the monitoring website of Academia Sinica. From these files we generated 19 attributes for each given center and then replicated these attributes to all the respective queues. The 19 attributes contain information such as the geographical region of a Resource Center, the available storage space on the Storage Element used by a particular CE, and results from various tests concerning BDII hosts.

(iv) *Host sensor data (GridICE)*: We performed over 500,000 LDAP queries on every EGEE Computing Element host that published GridICE [10] sensor data (i.e., on  $\approx 184$  computing element hosts). The interval between consecutive probes was 10 minutes. We were able to extract 18 attributes of interest that includes information such as the total and available sizes of RAM, virtual memory and filesystem-specific information.

(v) *Network Tomography Data (SmokePing)*: We obtained a 313MB snapshot of the *gPing* database from ICS-FORTH (Greece) for the studied period. The database contains network monitoring data for all the EGEE sites. From this collection we measured the average round-trip-time (RTT) and the packet loss rate relevant to each South East Europe CE which therefore yielded 2 additional attributes. In order to make the information consistent with the FailBase repository schema, we replicated files from the CE-level to CE-queue-level using a one-to-one mapping function.

## 5 Experimental Evaluation

In this section we describe an experimental study of the FailRank framework as well as our methodology.

### 5.1 Methodology

We have implemented a trace-driven tool in GNU C++ which processes the Failbase repository and then simulates the execution of the FailRank framework. In particular, we replay the trace in our simulator and at each timestamp we identify the  $K$  sites that might fail to respond. We will denote these  $(timestamp, siteID)$  tuples as the *Identified Set* ( $I_{set}$ ). The  $I_{set}$  is constructed by selecting the  $K$  highest-ranked answers from the execution of the scoring function described in Section 3.3 with equal weights on the FSM table.

Note the resource broker can compute the  $I_{set}$  directly from the FSM matrix, before the timestamp at which the actual error happens, thus such an approach provides an a priori failure detection mechanism. In order to assess this claim and validate that the  $I_{set}$  corresponds to the actual sites that have failed to respond, we need a set of  $(timestamp, siteID)$  tuples at which real site failures happened. We shall denote such a set as the *Real Set* ( $R_{set}$ ) and we construct it by combining the 18 attributes provided by the SAM service (described in 4.2) using the scoring function described in Section 3.3. That yields an average score per site for every timestamp. For each timestamp, we then again choose the  $K$  sites which have the highest score. We define the *penalty*, for not finding the correct sites at timestamp  $i$ , using a set-theoretic notation as follows:

$$Penalty_i = |R_{set} - I_{set}| \quad (2)$$

where  $|R_{set}| = |I_{set}| = K$  and the penalty at each timestamp  $i$  is defined as the cardinality of the set difference  $R_{set} - I_{set}$ . In our experimentation, we shall also use the *Aggregate Penalty* (i.e.,  $\mathcal{A} =$

$\sum_{i=1}^{timestamps} Penalty_i$ ), which provides a measure of overall efficiency for the  $I_{set}$  in all timestamps.

Having identified the correct  $I_{set}$  sites, our objective is to blacklist these sites and exclude them from the job scheduling process, decreasing in that way the number of failures.

## 5.2 Evaluating FailRank

In this subsection we evaluate the efficiency of the FailRank framework in identifying the sites that will fail. In particular, we obtain the  $I_{set}$  using two alternative strategies: i) *FailRank Selection*, which utilizes the FSM matrix and selects the  $K = 20$  sites ( $\approx 10\%$  of all sites) that maximize the scoring function of Section 3.3 with equal weights; and ii) *Random Selection*, which does not utilize the FSM matrix and simply selects the  $K = 20$  sites at random.

We then measure the respective penalty using our provided definition. Note that for this experiment we utilize a subset of the Failbase repository (i.e., 197 OPS queues monitored for 32 days) for which we had the largest number of available attributes. We also apply a spline interpolation smoothing between consecutive time points in our graph in order to facilitate presentation.

Figure 2 illustrates that FailRank selection always has an extremely low penalty (i.e. on average  $2.14 \pm 1.41$  with  $\mathcal{A} = 92,596$ ) while Random selection is always very close to 20 (i.e. on average  $18.19 \pm 3.5$  with  $\mathcal{A} = 786,148$ ). We can conclude that FailRank misses the correct sites in only 9% of the cases while Random misses the correct results in 91% of the cases. Another observation is at time instances 6000, 16000 and 39000, both selection curves drop to zero. This is attributed to the fact that our meta-information trace contained missing values at the given points (i.e.,  $I_{set} = R_{set} = \emptyset$ ). One final observation is that the Random selection curve is in some cases above 20. This is attributed to the fact that the cardinality of the  $R_{set}$  might be bigger than  $K$ , instead of equal to  $K$ , in certain cases. This is explained as follows: to construct the  $R_{set}$  we identified the  $K$  highest ranked tuples for each timestamp. In some cases the  $K^{th}$  tuple has an equal score to the  $K^{th} + 1$  tuple (or maybe even the  $K^{th} + 2$  tuples, etc.). As a result,  $|R_{set}|$  might be bigger than  $|I_{set}|$  which consequently might yield a penalty larger than  $K$  (e.g. consider the case where  $R_{set} \cap I_{set} = \emptyset$ ).

## 5.3 Scoring Function Evaluation

In the second experimental series we study whether we can further decrease the penalty of the Failrank approach by tuning the scoring function. Since some of the 75 attributes might be more important in defining the failure, we asked our administrators to manually provide weights to the 75 attributes given in the trace. Of course this assignment might not be optimal but it provides us with an understanding of how much the penalty can be improved. We will denote this edition of the FailRank algorithm as the *Expert Scoring* approach while the former approach, that assigned equal weights to all attributes, as the *Naive Scoring* approach.

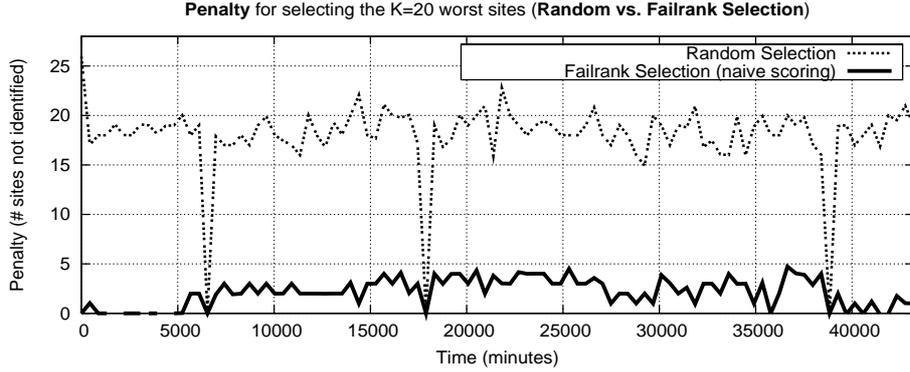
Figure 3 illustrates that by fine-tuning the weights using the expert scoring method we can achieve a significant reduction in the penalty. In particular, the penalty is now on average  $1.48 \pm 1.04$  (with  $\mathcal{A} = 64,008$ ) which presents a 31% improvement from the naive scoring approach. The Failrank method with expert scoring misses failures in only 7.4% of the cases which is clear improvement to the Random method presented in the previous subsection. Finally note that the current penalty is not a lower bound on the penalty achieved by the optimal assignment of weights but identifying such a bound efficiently and in real-time will be a subject of future work.

## 6 FailRank Extensions

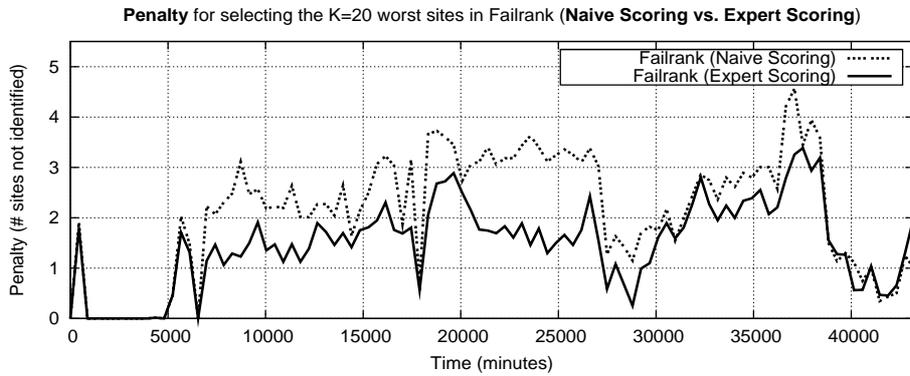
In this section we review some exploratory data analysis, learning and prediction applications that can be built on top of the FailRank architecture.

**(i) Finding State-related Sites:** An interesting question is whether any pair of sites features a similar *site vector*. This is an indication that two or more sites are in a similar failure state, with regards to the attributes of FSM. In order to answer this question we need a method that compares two vectors ( $\vec{q}, \vec{s}_i$ ), and finds if these are similar. An efficient technique, widely used in the information retrieval domain, is the *cosine similarity* [11]. The cosine similarity finds the cosine of the angle between two vectors. If two vectors are identical then the cosine similarity is 1 (because the angle between them is 0). On the contrary if two vectors are different then the similarity is closer to 0. The cosine similarity is calculated as following:

$$sim(q, s_i) = \frac{\sum(\vec{q} * \vec{s}_i)}{\sqrt{\sum(\vec{q})^2} * \sqrt{\sum(\vec{s}_i)^2}} \quad (3)$$



**Figure 2. FailRank selection vs. Random selection: FailRank identifies the site that have failed as opposed to Random which always identifies very few of the  $K=20$  sites.**



**Figure 3. Failrank Scoring Function Evaluation: When the Failrank scoring function is tuned by an expert (Expert scoring) it yields more accurate results than the alternative of setting the scores uniformly (Naive Scoring).**

By executing the cosine similarity for the sites in Table 1, we find that the highest similarity is  $\text{sim}(s_2, s_4) = 0.97$  while the smallest is  $\text{sim}(s_1, s_5) = 0.57$ . This means that  $s_2$  and  $s_4$  have a close relation across the different dimensions of the Failshot Matrix while  $s_1$  and  $s_5$  have a very distant relation.

**(ii) Timeseries Similarity Search:** Identifying which attribute timeseries are similar allows us to find the correlated attributes in FSM. For instance we can find that the QUEUE timeseries is correlated to the CPU timeseries for some site. To formalize our description, let  $P = (p_1, p_2, \dots, p_l)$  and  $Q = (q_1, q_2, \dots, q_l)$  denote two 1-dimensional timeseries of length  $l$  (each point denotes some item  $s_{i,j}$  in FSM).

The most straightforward way to compute the similarity between  $P$  and  $Q$  is to apply the *Euclidean distance* ( $L_2$ ) which is given by  $d = |P - Q| =$

$\sqrt{\sum_{i=1}^l |p_i - q_i|^2}$ . Since data points are only matched at identical time positions, the running time of this approach is  $O(l)$ . However this distance is not able to handle *out-of-phase matches*. To understand this consider two identical timeseries  $P$  and  $Q$ , where  $Q$  is shifted in time by some offset  $t$  (i.e.  $p_i = q_{i+t}, \forall i \leq l$ ). Using  $L_2$  would obviously not yield any similarity between  $P$  and  $Q$ . The Dynamic Time Warping (DTW) [1], Longest Common Sub-Sequence (LCSS) [3] and the Upper *LCSS* method [29] allow local stretching by matching each point of  $P$  with other points of  $Q$  within some window  $\delta$  (i.e.  $p_i$  is matched with  $q_{i \pm \delta}, \forall i \leq l$ ). This allows us to correlate noisy failure timeseries with out-of-phase matches again in an effective manner (i.e., in  $O(l)$  time).

**(iii) Decision Tree Learning:** Given a site vector  $s_i = \{a_1, \dots, a_m\}$ , we want to predict if  $s_i$  will fail (with

some statistical confidence). To answer this question, we train a Decision tree  $T$  [11] in an offline phase using a corpus of annotated failures. We then extract the classification rules that are utilized by the FailRank system. For instance if we learn that a site vector of the form  $\{\text{CPU} \geq 0.70, \text{DISK} \geq 0.90, \text{QUEUE} \geq 0.85, \text{any}\}$  fails in 95% of the cases, then sites satisfying this rule are excluded from the job scheduling process. An interesting problem is to provide a decision tree which continues its learning behavior even after the initiation of the system and which gracefully adapts to changes.

## 7 Conclusions & Future Work

In this paper we introduce FailRank, a novel framework for integrating and ranking information sources that characterize failures in a grid system. This perspective is to our knowledge new and fits well the computation model of grid infrastructures. Another advantage is that FailRank streamlines the very complex task of monitoring large-scale distributed resources in an automated manner. In the future we plan to provide more elaborate ranking algorithms and perform an in-depth assessment of our prototype system under development.

**Acknowledgements:** This work was supported in part by the European Union under projects CoreGRID (#IST-2002-004265) and EGEE (#IST-2003-508833).

## References

- [1] Berndt D., Clifford J., “Using Dynamic Time Warping to Find Patterns in Time Series”, In *KDD* 1994.
- [2] Bruno N., Gravano L. and Marian A., “Evaluating Top-K Queries Over Web Accessible Databases”, In *ICDE* 2002.
- [3] Das G., Gunopulos D., Mannila H., “Finding Similar Time Series”, In *PKDD*, 1997.
- [4] Chun G., Dail H., Casanova H., and Snavely A., “Benchmark probes for grid assessment”, In *IEEE IPDPS* 2004.
- [5] “CIC”, <http://cic.gridops.org/>
- [6] Da Costa G., Orlando S., Dikaiakos M.D., “Nine months in the life of EGEE: a look from the South”, In *IEEE MASCOTS* 2007.
- [7] Dumitrescu C., Raicu I., Ripeanu M., Foster I., “DiPerF: An automated DIstributed PERformance testing Framework”, In *IEEE/ACM Grid* 2004.
- [8] “EGEE”, <http://www.eu-egee.org/>.
- [9] “Global Grid User Support (GGUS) ticketing”, <https://gus.fzk.de/pages/home.php>
- [10] “GridICE”, <http://grid.infn.it/gridice/>
- [11] Han J. Kamber M., “Data Mining: Concepts and Techniques”, 2nd edition, Elsevier, 2006.
- [12] Fagin R., Lotem A. and Naor M., “Optimal Aggregation Algorithms For Middleware”, In *PODS* 2001.
- [13] Foster I. and Kesselman C., “The Grid: Blueprint for a New Computing Infrastructure”, Elsevier, 2004.
- [14] Foster I., Kesselman C., and Tuecke S., “The Anatomy of the Grid: Enabling Scalable Virtual Organizations”, In *Intl. J. Supercomputer Applications*, 15(3):200–222, 2001.
- [15] Foster I., “Globus Toolkit Version 4: Software for Service-Oriented Systems”, In *IFIP ICNP* 2005.
- [16] Glite middleware <http://glite.org/>
- [17] GStat: <http://goc.grid.sinica.edu.tw/gstat/>
- [18] Junqueira, F. P., and Marzullo, K., “The virtue of dependent failures in multi-site systems”, In *HotDep* 2005.
- [19] Kiciman E. and Fox A., “Detecting Application-Level Failures in Component-based Internet Services”, In *IEEE Transactions on Neural Networks*, 2004.
- [20] Kiciman E. and Subramanian L., “Root Cause Localization in Large Scale Systems”, In *HotDep* 2005.
- [21] Krishnamurthy S., Sanders W.H., Cukier M.: “A Dynamic Replica Selection Algorithm for Tolerating Timing Faults”, In *DSN* 2001.
- [22] Locasto M.E., Sidiroglou S., and Keromytis A.D., “Application Communities: Using Monoculture for Dependability”, In *HotDep* 2005.
- [23] “OSG”, <http://www.opensciencegrid.org>.
- [24] Raman R., Livny M., Solomon M.H., “Matchmaking: An extensible framework for distributed resource management”, In *Cluster Computing*, Vol 2, 1999.
- [25] “TeraGrid”, <http://www.teragrid.org/>
- [26] Tsouloupas G., Dikaiakos M.D., “GridBench: A Tool for the Interactive Performance Exploration of Grid Infrastructures”, In *Journal of Parallel and Distributed Computing*, Vol 67, pp 1029-1045, 2007.
- [27] Neokleous K., Dikaiakos M.D., Fragopoulou P., Markatos E.P., “Failure Management in Grids: The Case of the EGEE Infrastructure”, In *Parallel Processing Letters* (in press, Dec. 2007).
- [28] Tsouloupas G. and Dikaiakos M.D., “Grid Resource Ranking using Low-level Performance Measurements.”, In *Euro-Par* 2007.
- [29] Vlachos M., Hadjieleftheriou M., Gunopulos D., Keogh E., “Indexing multi-dimensional time-series with support for multiple distance measures” In *KDD* 2003.
- [30] “WISDOM”, <http://wisdom.eu-egee.fr/>
- [31] “Service Availability Monitoring (SAM)”, <http://goc.grid.sinica.edu.tw/gocwiki/SAM>