Parallel Processing Letters, Vol. 18, No. 3 (2008) 371–390
© World Scientific Publishing Company

METADATA RANKING AND PRUNING FOR FAILURE DETECTION IN GRIDS*

DEMETRIOS ZEINALIPOUR-YAZTI
Pure and Applied Science, Open University of Cyprus, 1304, Nicosia, Cyprus
dzeina@cs.ucy.ac.cy

HARRIS PAPADAKIS
Foundation of Research and Technology - Hellas, Institute of Computer Science
Heraklion, Crete, Greece

CHRYSSIS GEORGIOU, MARIOS D. DIKAIAKOS
Department of Computer Science, University of Cyprus, 1678, Nicosia, Cyprus

Received May 2008
Revised July 2008
Communicated by P. Fragopoulou

ABSTRACT

The objective of Grid computing is to make processing power as accessible and easy to use as electricity and water. The last decade has seen an unprecedented growth in Grid infrastructures which nowadays enables large-scale deployment of applications in the scientific computation domain. One of the main challenges in realizing the full potential of Grids is making these systems *dependable.*

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 and can achieve this by only fetching 65% of the available information sources. We believe that our work constitutes another important step towards realizing adaptive Grid computing systems.

Keywords: Data Ranking Algorithms, Computational Grids, Failures, Scheduling

*This work is supported in part by the European Union under projects CoreGRID (# IST-2002-004265) and EGEE (#IST-2003-508833). A Preliminary version of this paper has appeared in [32] and [33]. The second author was supported by a CoreGRID REP Fellowship during 2008.

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, 14]. 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 failure 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 a 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. Background on Grid Computing

In this section we will describe the anatomy of a Grid system and detail all the components pertinent to the operation of a Grid site. In particular, we will focus on Grid computing in the context of the EGEE project although other architectures feature a similar framework. We also describe the main causes of unsuccessful job executions in a grid system.

2.1. *The Anatomy of a Grid*

A Grid interconnects a number of remote clusters, or *sites*. Each site features heterogeneous resources (hardware and software) and the sites are interconnected over an open network such as the Internet. Figure 1 illustrates the anatomy of a typical grid (rectangles represent hardware while ellipses the services). The figure shows how sites with different capabilities and capacities are contributing their resources to the Grid infrastructure. In particular, each site features one or more *Worker Nodes*, which are usually rack-mounted PCs. The *Computing Element* shown in the same figure runs various services responsible for authenticating users, accepting jobs, performing resource management and job scheduling. Additionally, each site might feature a *Local Storage* site, on which temporary computation results can reside, and local *Software* libraries, that can be utilized by executing processes. The Grid middleware is the component that glues together local resources and services and exposes high-level programming and communication functionalities to application programmers and end-users. For instance EGEE uses the gLite middleware [16], while NSF's TeraGrid is based on the Globus Toolkit [15]. A Grid system also features some global services which are described in the next subsection.

2.2. *Lifecycle of Grid Jobs*

A Grid job, or computation, consists of a set of input files that defines the elements of a given computation (code, custom libraries, input files, etc). Grid jobs can be

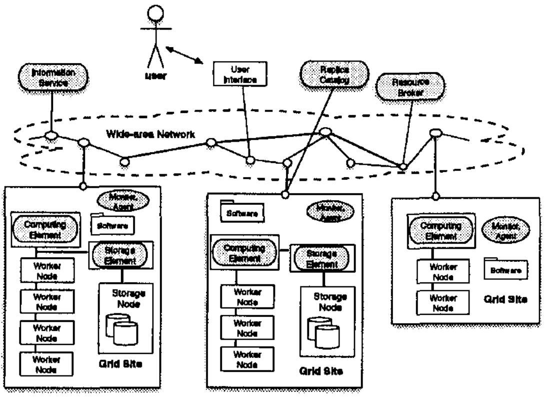

Fig. 1. The Anatomy of a GRID Infrastructure.

classified as *CPU-intensive* and *data-intensive*, depending on the type of work performed. For clarity we divide the lifecycle of a grid job into the following three conceptual phases:

(i) Assignment Phase: Jobs are submitted to a Grid by users through some authenticated remote workstation, denoted as the *User Interface (UI)*. Besides obtaining the output from completed jobs, the UI might also provide supplementary functionality for requesting the status of a job and the status of resources in the system. Jobs submitted to the UI are directed to some *Resource Broker (RB)*, a central global grid service that performs matching between requests and available resources using the *matchmaking* approach [24]. Being able to quickly identify failures, would obviously be very helpful information to the *RB* as it would be able to avoid bottlenecks and resources leading to errors. Although this is not currently possible, our work sets the foundation towards this goal. The matchmaking performed by the *RB* is based on the information provided by another central service, the *Information Index*, which provides information about the state of grid resources. If the matchmaking is successful, the job is sent to the respective computing elements for execution.

(ii) Execution Phase: During job execution, if any *input files* are necessary, these have to be pushed to a remote grid site at runtime. Alternatively these files could have been pushed to the grid site during the assignment phase. In both occasions, a service called the *Replica Catalog* maintains the location of various replicas of a file held in remote Storage Elements.

(iii) Completion Phase: When the job completes successfully, the user is informed through the User Interface with a set of output files that are a superset of the command line outputs, had the job run on a standalone computer. Although the user will be notified in the event of a failure, there is no indication about the possible cause.

2.3. *Causes of Failures*

In this section we identify the main causes of failures in Grid infrastructures. These observations are extrapolated from the experiences we acquired by operating an EGEE grid site that consists of: (i) a Regional Resource Broker (3.6GHz/1GB RAM), (ii) a Regional Information Service which features the same aforementioned characteristics, (iii) a 72 CPU cluster of Worker Nodes which utilizes a blend of 2.6GHz AMD Opteron and 2.8GHz Xeon CPUs, and (iv) a Storage Element which features 4x250GB disk space in RAID 5. Our analysis takes into account 37,860 job submissions ($\approx$19K normalized CPU hours), between March 2005 and June 2006. We combine our observations with others obtained by fellow-researchers [30, 18] to conclude the following:
Grid component failures: One or more of the components involved in the Grid infrastructure could malfunction due to *hardware failures* (e.g., hard drive burns, RAM or motherboard failures, power supply failures and overheating) and *software faults* (e.g., O/S mis-configurations and middleware bugs). Such problems may result to a total collapse of a component (crash failure) or to a component becoming partially unresponsive or extremely slow.
Network failures: Network links could cause permanent or transient network disconnections leading to a loss, corruption or delay of messages and data transfers. Network disconnections may result to total inaccessibility of a Grid component, a condition that is equivalent to a crash failure of that component. Network access mis-configuration (firewall changes or updates) lead to the same effect.
Information faults: The information provided by the Grid *Information Service*, which provides state information about the distributed grid sites, may be erroneous or obsolete due to administrator errors, software faults, and network delays. As a result, the *Resource Broker*, a central service that performs matching between resources and requests based on this information, may take sub-optimal decisions that result to excessive delays in job processing or even to failures in job execution.
Excessive delays: In the large, shared and dynamic Grid infrastructure, unusual workload conditions, like those triggered by flash crowds and denial of service attacks, may lead to long queuing delays in Computing or Storage Elements, to reduced Grid service throughput, and to long network delays in data transfers. Such conditions may result to job turnaround times that are substantially longer than those expected by Grid users. A similar effect may arise also because of the heterogeneity of the Grid: jobs may end-up being executed on very slow resources, resulting to unacceptably slow execution times. Because of the resource virtualization imposed by many Grids, end-users have limited control over the performance characteristics of resources allocated to their jobs.

3. Monitoring Failures in a Grid Environment

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.

4. 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.

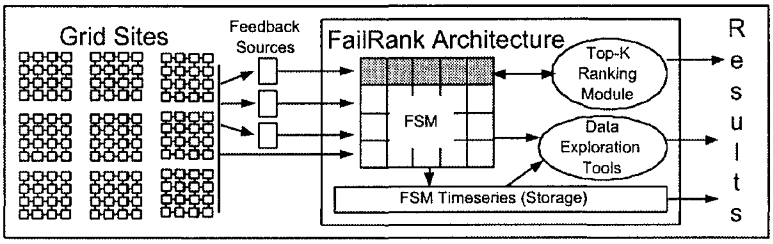

Fig. 2. The FailRank System Architecture: Feedback sources are continuously coalesced into a representative array of numeric vectors, the *FailShot Matrix (FSM)*. FSM is then continuously ranked in order to identify the K sites with the highest potential to feature some failure.

4.1. *Architecture Overview*

The FailRank architecture (see Figure 2), 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.

4.2. *Definitions and System Model*

In this section we will provide some definitions and our system model upon which we will structure our presentation in the subsequent sections.

Definition 1 (***FailShot Matrix (FSM)***): Let S denote a set of n grid *sites* (i.e., $S = \{s_1, s_2, ..., s_n\}$). Also assume that each element in S is characterized by a set of m attributes (i.e., $A = \{a_1, a_2, ..., a_m\}$). These attributes are obtained by the feedback sources described in Section 3. 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 $\{s_1, ..., s_5\}$ 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 2 (***FSM Timeseries***): A temporal sequence of l FailShot Matrices defines an *FSM Timeseries of order l*.

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 7 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 5, we found that the FSM representation saves us approximately 350GB of storage per month.

4.3. *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. Finally, we developed a mechanism for selective extraction of monitoring information for selecting those K sites, which we describe later on. This mechanism is capable of reducing the information we need to fetch and process by approximately one third.

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:

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

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 6.3 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 $n' << n$).

Table 1: The *FailShot Matrix (FSM)*.

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=CY01-KIMON	0.44	0.07	0.70	0.19	0.67

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 can 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 table, which is similar to Table 1 with the exception that each column is sorted in descending order of the value. 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 τ as the sum of all scores in the first row (i.e., $\tau = .99 + .91 + .92 + .74 + .67 = 4.23$). Since τ is larger than both scores of s_4 and s_2, the *TA* algorithm performs another iteration in which the threshold τ 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.

5. 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.

5.1. *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 [timestamp,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).

5.2. *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

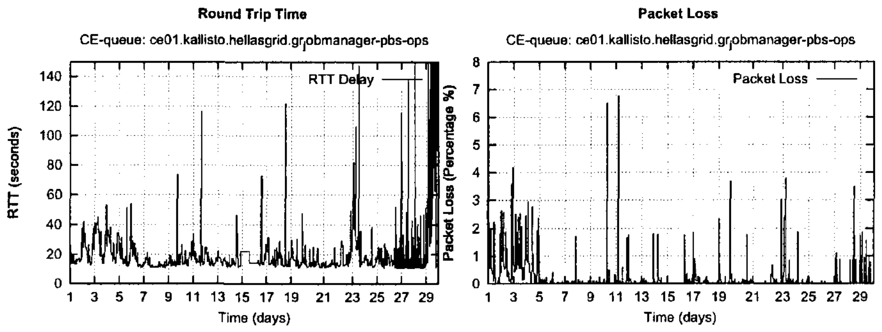

Fig. 3. Round-Trip-Time (left) and Packet Loss (right) for the CE-queue `ce01.kallisto.hellasgrid.gr_jobmanager-pbs-ops`. These attributes are two of the 72 attributes maintained for the 2,565 CE-queues in the Failbase Repository.

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 *bdii101.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 the filesystem.

(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 (see Figure 3) 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.

5.3. *Pruning the Meta-Information Retrieval Space*

Although the Failbase repository is an invaluable tool for offline data exploration and analysis it is quite expensive (with regards to network I/O, processing and storage) to construct and maintain such a repository in an online manner. Additionally, a huge meta-information repository could also impose a limitation on how often the ranking function can be executed, consequently limiting the failure detection capability of our system. Therefore, we seek to prune the space of possible FSM values and only focus on those values that will determine the final top-k result.

In this subsection we will sketch a greedy algorithm to prune the meta-information space in an online manner without compromising the accuracy of the FailRank framework. In particular, we devise an iterative algorithm which consists of the following steps: We first sort the m attributes of $A = \{a_1, a_2, \ldots, a_m\}$ in descending weight order (i.e., $w_1 \geq w_2 \geq \ldots \geq w_m$). Next, we fetch the information from the meta-information source with the highest weight (i.e., w_1). Let this column be the j^{th} attribute of the FSM table (i.e., $a_j = (s_1, s_2, \ldots, s_n)$), where $j \leq m$. For each value in the a_j vector we construct an upper bound $high(s_i)$ $(i \leq n)$ by substituting the value of the missing $m - j$ attributes by their maximum possible value (i.e., $high(s_i) = s_i + (m - j) * \alpha$, where α is the maximum possible value for each attribute). Obviously, the final score for each site s_i $(i \leq n)$ lies somewhere in the range $[s_i \ldots high(s_i)]$. The problem that we are now challenged to solve is that of identifying the K sites with the highest overall value (i.e., even for the attributes that have not been fetched yet). To achieve this without fetching all respective attributes we process the $[s_i \ldots high(s_i)]$ ranges in descending $high(s_i)$ order discarding any range with an upper bound lower than the K^{th} highest-ranked lower bound s_i. The latter one defines a threshold τ below which all tuples can safely be eliminated. In particular, it can be proven that any pruned-away tuple s_x can not be in the final top-K result-set, thus s_x can safely be excluded from further consideration. The same procedure is iteratively repeated until K sites have been identified.

6. Experimental Evaluation

In this section we describe our experimental methodology and the results of our evaluation.

6.1. *Methodology*

We have implemented a trace-driven tool in GNU C++ and JAVA 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 evaluate a variety of evaluation metrics, as these are described next, in order to assess the efficiency of our framework:

i. **Prediction Accuracy:** this metric quantifies how accurately FailRank can identify the failing sites. 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 4.3 with equal weights on FSM.

   Note the system 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 have 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 5.2) using the scoring function described in Section 4.3. These attributes provide an accurate view of the failure state for each CE-queue [a]. That yields an average score per site for every timestamp. For each timestamp, we then again choose the K sites with the highest score. We define the *penalty*, for not finding the correct sites at timestamp i, using a set-theoretic notation as follows:

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

   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.

ii. **Pruning Efficiency:** this metric quantifies the efficiency of our pruning algorithms which eliminates the values of the FSM table that can not contribute to the final top-k result. Practically, that means that the FailRank system will need to acquire less information in order to derive the K highest ranked answers all this without compromising the top-k retrieval accuracy. In particular, we replay the trace in our simulator and identify at each timestamp i all the FSM values that are below the threshold τ and that can be excluded. We will denote the remaining (*timestamp*, *value*) tuples, those that will be downloaded from the meta-information sources, as the *Fetched Set ($Fetched_{set}(i)$)*. Note that the FailRank system computes $Fetched_{set}(i)$ incrementally as the data gradually streams from the distributed meta-information sources. The upper bound on the number of all possible values that are available to the FailRank system on time instance i is denoted as the *All Values Set ($AV_{set}(i)$)*. $AV_{set}(i)$ has a known cardinality of $m \times n$, where m is the number of attributes available to

[a]Note that the SAM attributes unveil a posteriori the failure state of each individual grid site, thus these can not be taking into account for the derivation of the I_{set}.

the system and n the number of CE-queues that the FailRank system monitors.

We investigate the achieved pruning of our system using two different criterions. The first criterion measures the amount of pruning (denoted as $Pruning_i$) that is achieved at each timestamp i of the trace. In particular, this metric is defined as the cardinality ratio of the $Fetched_{set}(i)$ over the $AV_{set}(i)$, formally:

$$Pruning_i = \frac{|Fetched_{set}(i)|}{|AV_{set}(i)|} \tag{3}$$

The second criterion measures the number of iterations our pruning algorithm requires in order to derive the $Fetched_{set}$ and consequently determine the K highest-ranked answers. In particular, since the pruning algorithm is an iterative algorithm in each iteration it fetches the next attribute of the FSM table with the highest weight and we are interested in finding how many iterations it takes until our algorithm converges. For this reason we define the *Level-Wise Pruning* metric (denoted as $\mathcal{L}_j$) which defines the number of FSM rows pruned-away in each algorithm iteration j. In particular, for each iteration we calculate the average for all time instances using the following summation:

$$\mathcal{L}_j = \left(\frac{1}{timestamps}\right) \sum_{i=1}^{timestamps} Rows\_Pruned\_Away_i \tag{4}$$

6.2. *Evaluating the Prediction Accuracy*

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 4.3 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 4 illustrates that FailRank selection always has an extremely low penalty (i.e., on average 2.14 ± 1.41 with $\mathcal{A} = 92,596$) while Random selection is always very close to 20 (i.e., on average 18.19 ± 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

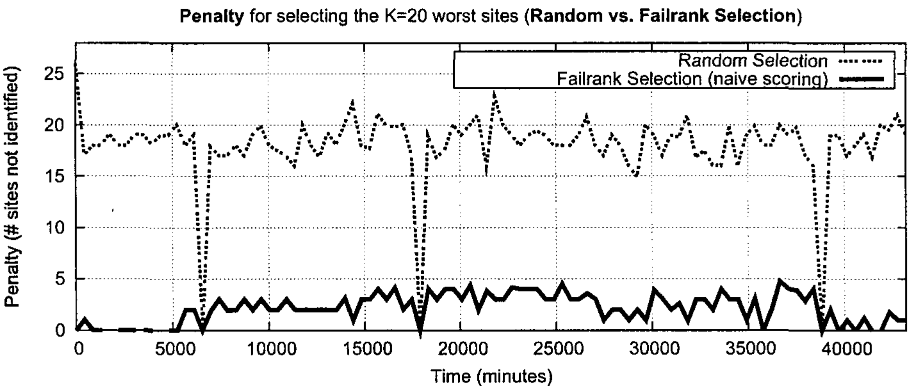

Fig. 4. **FailRank selection vs.Random selection:** FailRank identifies the site that have failed as opposed to Random which always identifies very few of the K=20 sites.

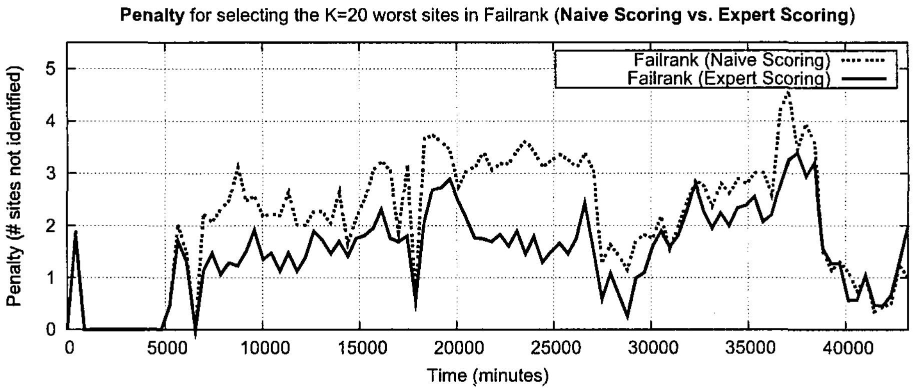

Fig. 5. **FailRank Scoring Function Evaluation:** When the FailRank scoring function is tuned by an expert (Expert scoring) it yields more accurate results than the alternative of setting the scores uniformly (Naive Scoring).

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 $R_{set} \cap I_{set} = \emptyset$).

6.3. *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 a lower bound

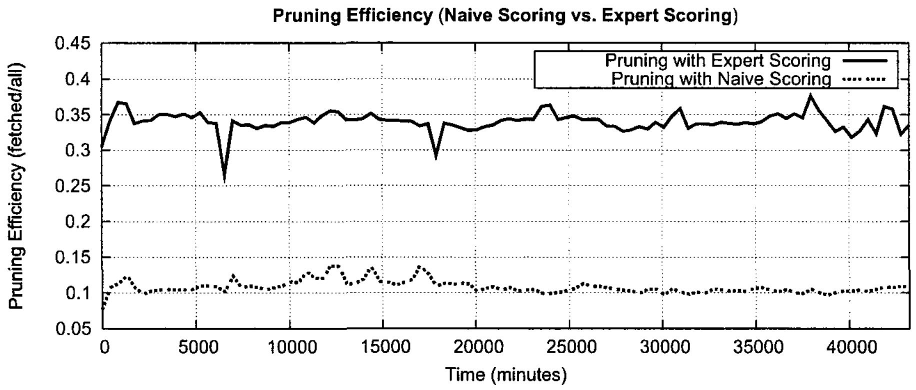

Fig. 6. **Pruning Efficiency:** The ratio of attribute values not fetched in each time instance.

on the feasible improvement of the penalty metric. 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 5 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 ± 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.

6.4. *Pruning Efficiency Evaluation*

In the last experimental series we assess the two pruning evaluation metrics we defined earlier.

Figure 6, presents the $Pruning_i$ evaluation metric for the 43,200 timestamps by utilizing the Naive Scoring scheme and the Expert Scoring scheme. The figure shows that by utilizing the naive scoring scheme we can still retrieve the K highest-ranked answers by spending 11% less on retrieving data from the meta-information sources. Notice that the FailRank system will conduct the meta-information gathering very frequently, thus even a seemingly small increase in the pruning magnitude has a significant benefit on the performance of the system. The result is even more encouraging for the Expert Scoring approach in which we achieve a 34% pruning magnitude. That means that the system will require to fetch only the 2/3 of the available metadata in order to derive the correct answer.

Figure 7 (top-bottom), presents the level-wise pruning efficiency $\mathcal{L}_j$, where j is in the range 1-25. From the two figures we can draw the following conclusions: i) The Expert Scoring approach convergences much faster than the Naive Scoring approach. In particular, the bottom figure shows the Expert scheme will complete in 21 iterations while naive scoring in 25 iterations. This observation can be explained

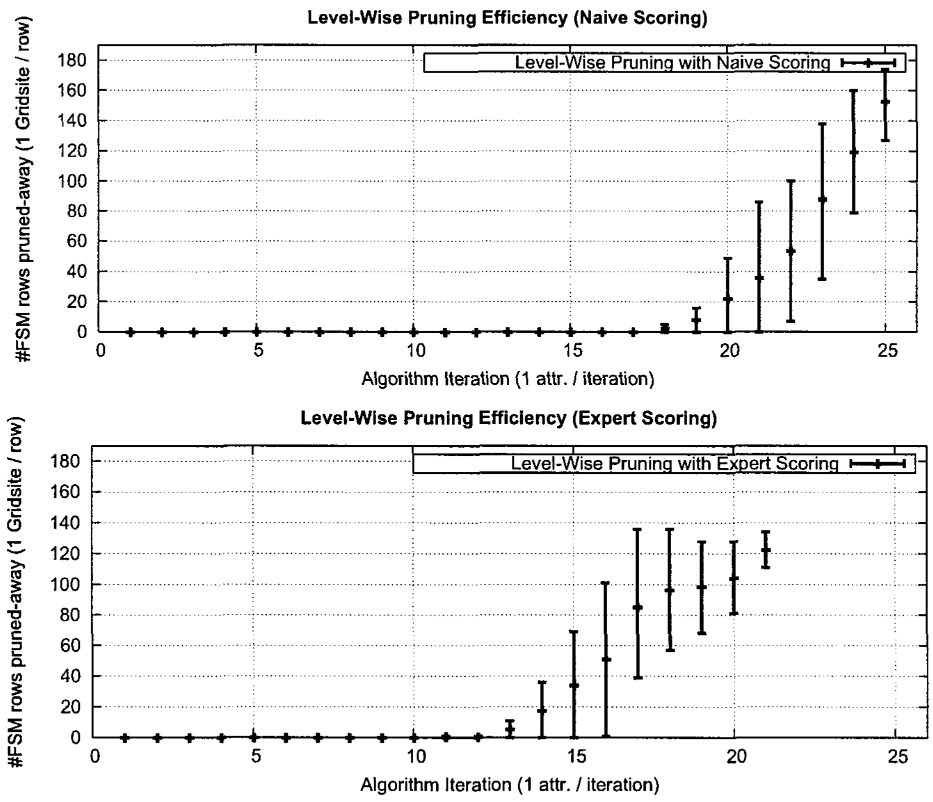

Fig. 7. **Level-Wise Pruning Evaluation:** The number of rows excluded in each iteration of the pruning algorithm using Naive Scoring (top) and Expert Scoring (bottom).

by the fact the Expert method assigns different weights to the m attributes, consequently the pruning algorithm can eliminate much faster the tuples below the τ threshold. Related to the above comment is also the observation that the Expert Method maintains this relative advantage over the Naive method for all time instances. For instance, when j=21 the expert method prunes away 61% of the rows while the naive method only 17%. ii) A second observation is that in both scoring schemes the first 10-12 iterations yield no pruning. Consequently, a real implementation can request the retrieval of these attributes in the first iteration.

7. 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:

$$sim(q, s_i) = \frac{\sum(\vec{q} * \vec{s_i})}{\sqrt{\sum(\vec{q})^2} * \sqrt{\sum(\vec{s_i})^2}} \tag{5}$$

By executing the cosine similarity for the sites in Table 1, we find that the highest similarity is $sim(s_2, s_4) = 0.97$ while the smallest is $sim(s_1, s_5) = 0.57$. 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 $P = (p_1, p_2, ..., p_l)$ and $Q = (q_1, q_2, ..., q_l)$ denote two 1-dimensional timeseries of length l (each point denotes some item s_{ij} 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 $d = |P - Q| = \sqrt[2]{\sum_{i=1}^{l} |p_i - q_i|^2}$. 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 δ (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 $O(l)$ time.

(iii) Decision Tree Learning: Given a site vector $s_i = \{a_1,...,a_m\}$, 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 {CPU≥0.70, DISK≥0.90, QUEUE≥0.85, any, 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.

(iv) Prominent Future Challenges: In order to further improve the FailRank architecture we are challenged with the task of further improving metadata information gathering. In particular, we expect that the following two tasks will significantly boost the accuracy and performance of our system:

- **Failure Exchange Interfaces:** The first challenge is to develop efficient interfaces and protocols to exchange fault information between grid sites. The development of such protocols are currently difficult as the lack of a centralized authentication and administration scheme makes it intrinsically difficult to access the remote sites and monitor failures. Furthermore, it is currently also very hard to encapsulate failure feedback mechanisms in the application logic of individual grid software as the grid is an amalgam of pre-existing software libraries, services and components with no centralized control. What is required is a generic component that can be statically or dynamically linked to the software stack of grid software and which can enable the communication of relevant data through a common interface.

- **Failure Information Schema:** The second challenge is to develop a global schema that will define the nature of information to be exchanged. The lack of common parameters that characterize failures makes it hard to obtain a global understanding regarding failures. For instance, a given monitoring system might count I/O reads and writes, defined as the distinct number of I/O operations performed, while another grid monitoring system might count I/O bytes read and write, defined as the accumulative number of bytes that were spent on the given I/O operations. Additionally, it also remains to be shown that a fine granularity is or is not appropriate for the prognosis of failures. Large content distribution networks tend to collect low-level probes (e.g., ping and trace route data) in order to enable a variety of network tomography operations. Although such probes are essential in the establishment of these services, they incur an enormous network traffic. In the context of Grids, it is still not shown that such low-level information is efficient and that it can be obtained in a viable fashion.

8. 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.

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", 2E, 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 ICNP'05.
[16] Glite middleware http://glite.org/
[17] Grid Statistics (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, pp 129-138, 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
[32] Zeinalipour-Yazti D., Neocleous K., Georgiou C., Dikaiakos M.D,, "FailRank: Towards a Unified Grid Failure Monitoring and Ranking System", In *CoreGRID Workshop on Grid Programming Models and P2P Systems Architecture (Coregrid 2007 Workshop) Heraklion, Crete, Greece, June 12-13, 2007,*
[33] Zeinalipour-Yazti D., Neocleous K., Georgiou C., Dikaiakos M.D., "Identifying Failures in Grids through Monitoring and Ranking", In *The 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA'08), July 2008,*