Towards ServMark, an Architecture for Testing Grid Services

Mugurel Ionut Andreica, Nicolae Tapus, Catalin Dumitrescu, Alexandru Iosup, Dick Epema, Ioan Raicu, Ian Foster, Matei Ripeanu

DOC.ID ServMark -2006-002
Group Parallel and Distributed Systems
Type Technical Report
Subject ServMark
Team Mugurel Ionut Andreica, UPB
Nicolae Tapus, UPB
Catalin Dumitrescu, TU Delft/UMUE
Alexandru Iosup, TU Delft
Dick Epema, TU Delft
Ioan Raicu, U.Chicago
Ian Foster, U.Chicago
Matei Ripeanu, U. British Columbia
Date 30.07.2006
Contact servmark-dev@globus.org

Table of contents

Table of contents 1
1. Introduction 2
2. The Design of ServMark 2
3. The Implementation of ServMark 4
3.1. The ServMark controller 5
3.2. The interaction between the user and the ServMark controller 6
3.3. The modified DiPerF controller 8
3.4. The modified DiPerF submitter 8
3.5. The modified DiPerF tester 8
3.6. The modified GrenchMark 9
3.6.1. Generating a workload description file 10
3.6.2. Generating a workload file 10
3.6.3. Submitting the workload 10
3.6.3.1. The Thread pool 10
3.6.3.2. The Watchdog 11
3.7. The Database 11
3.7.1. The metric_type_mapping Table 11
3.7.2. The test_params Table 12
3.7.3. The test_logs table 12
3.7.4. The statistical_values table 13
3.7.5. The individual_values Table 13
3.8. The Database Module 14
3.9. The Metrics 14
3.10. Reliability 15
4. Validation and Testing 15
4.1. Validation 15
4.2. Testing 16
4.2.1. Experimental Setup 16
4.2.2. Test Setup Overview 16

4.2.3 Test Results 17
4.3. Undesirable behavior 20
5. Related Work 20
6. Conclusion and Ongoing Work 22
References 23
Appendix A. Installing ServMark 26
Appendix B. Installing the Web Servers used for testing 26

1. Introduction

Grid computing [23] provides a natural way to aggregate resources from different administrative domains for building large scale distributed environments [2]. The Web Services paradigm [24] proposes a way by which virtual services can be seamlessly integrated into global-scale solutions to complex problems. While the usage of Grid technology ranges from academia and research to business world and production, two issues must be considered: that the promised functionality can be accurately quantified and that the performance can be evaluated based on well defined means. Without adequate functionality demonstrators, systems cannot be tuned or adequately configured, and Web services cannot be stressed adequately in production environment. Without performance evaluation systems, the system design and procurement processes are limp, and the performance of Web Services in production cannot be assessed. In this paper, we present ServMark, a carefully researched tool for Grid performance evaluation. While we acknowledge that a lot of ground must be covered to fulfill the requirements of a system for testing Grid environments, and Web (and Grid) Services, we believe that ServMark addresses the minimal set of critical issues.

In order for the results to be significant, the ServMark must be able to create the conditions that the Grid environments (or their components) were designed to handle [4, 12]. Consider the case of a resource management system. Here, the system users submit jobs according to daily patterns [9, 15, 40], and may respond to the system's feedback (i.e., they will not continue to submit until their already submitted jobs finish) [41]. It would therefore be interesting to establish the performance of the resource management system under both real-life and extreme conditions. Because the number of resources to be found in nowadays Grids is on the order of thousands to tens of thousands [42], and because the size is expected to grow, the evaluation system must generate significant loads for the Grid environment in a scalable way. A similar situation occurs for the case of Web Services. By using a distributed approach, and a significant set of testing parameters, ServMark is able to generate a wide range of testing conditions for many Grid environments and services.

2. The Design of ServMark

ServMark is a system that integrates two previously developed evaluation systems: DiPerF and GrenchMark. DiPerF is a distributed testing system and test generator, and GrenchMark is a centralized system that can generate complex testing scenarios. ServMark makes use of the properties of both systems in order to generate truly significant testing scenarios.

The intended use for ServMark is to evaluate the performance of Grid environments and Grid and web services. Grid environments and web services have quite a different behavior in terms of response time, so different testing strategies will need to be used.

The testing process is initiated by a central controller, which distributes the testing parameters to multiple nodes. Each node generates its own test scenario based on the given parameters and then "plays" the generated scenario.

General requirements:

1. uniquely identify each test (REQ1)
2. generate a multi-node test according to the user specifications (REQ2)
3. store the test and make it available for replay (REQ3)
4. run the test and store its results (REQ4)
5. analyze the results and compute statistics (REQ5)
6. the performance evaluation must be online: results should be able to be visualized as the testing process advances (REQ6)

Figure 2-1 shows the proposed architecture for ServMark, highlighting the relationship between GrenchMark, DiPerf and the new ServMark modules. The interaction between the user and the ServMark Controller goes as follows: the user decides the parameters to be used in the testing process (see REQ2), starts the ServMark Controller, then is notified when the testing operation has completed. The ServMark Controller should generate a test ID for the testing process initiated by the user (see REQ1), update the database and send the testing parameters to the DiPerF controller. The DiPerF controller controls the testing process, by invoking the DiPerF submitter. It also updates the results into the database. The DiPerF submitter creates the tester processes and communicates with them, sending in parameters and receiving back test results. The DiPerF tester invokes GrenchMark, which performs the actual testing process and communicates with GrenchMark, sending parameters and receiving back test results. GrenchMark generates a workload according to the user parameters and then submits the generated workload for execution, computing the test results and sending them to the DiPerF tester. The test parameters are inserted into the database by the ServMark controller. The DiPerF controller inserts and updates the test results into the database as the testing process advances.

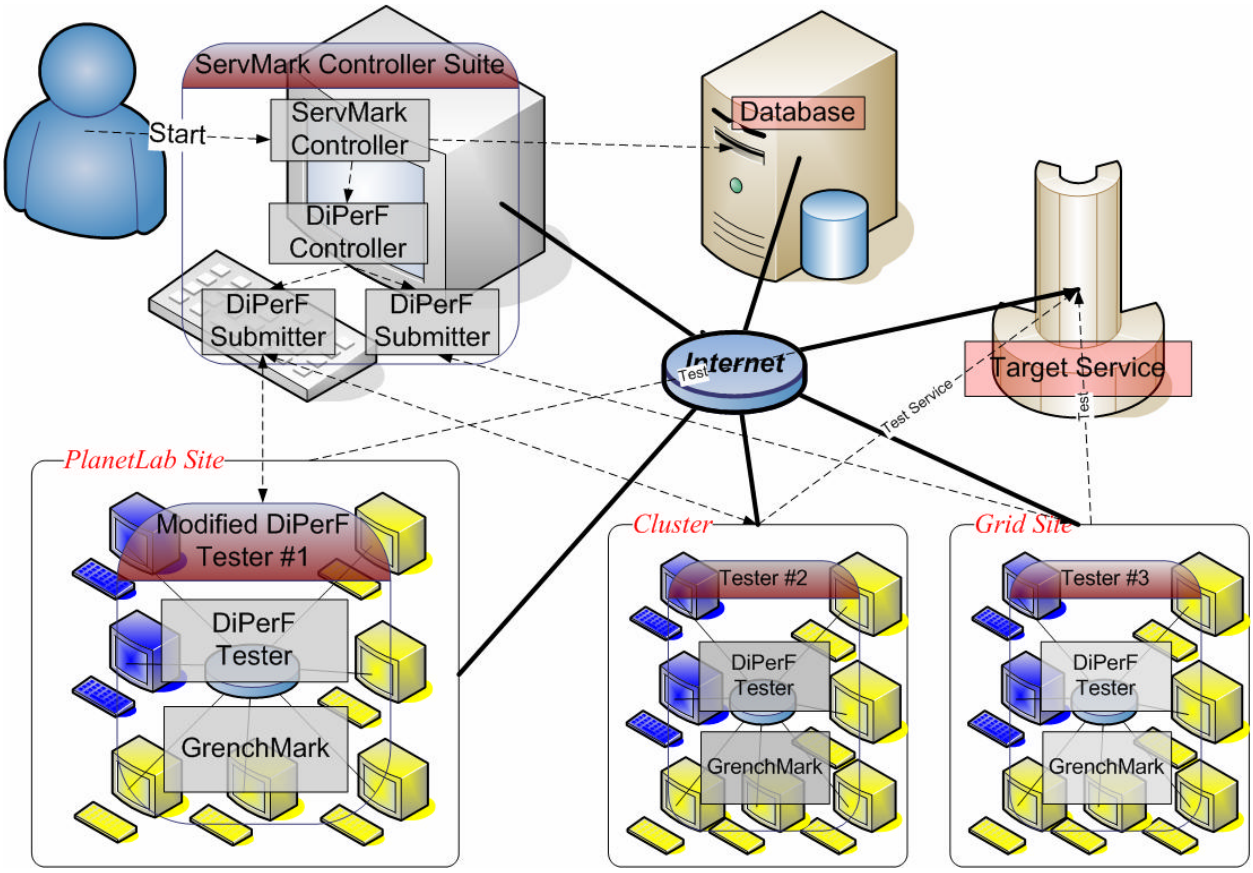

Figure 2-1. The Proposed ServMark Architecture

3. The Implementation of ServMark

Figure 3-1 shows the architecture of the implemented system. The ServMark Controller interacts directly with the database, in order to insert general information about the testing scenario, while the DiPerF controller interacts with the database through a database module, in order to insert or update the information gathered during the testing process. You can also see a more detailed description of GrenchMark, which is composed of two major modules: the workload generator and the workload submitter. The workload generator schedules the execution times of the jobs which compose the testing scenario.The workload submitter is a multi-threaded module which manages the job submission process, computes metrics and sends the results back to the DiPerF tester.

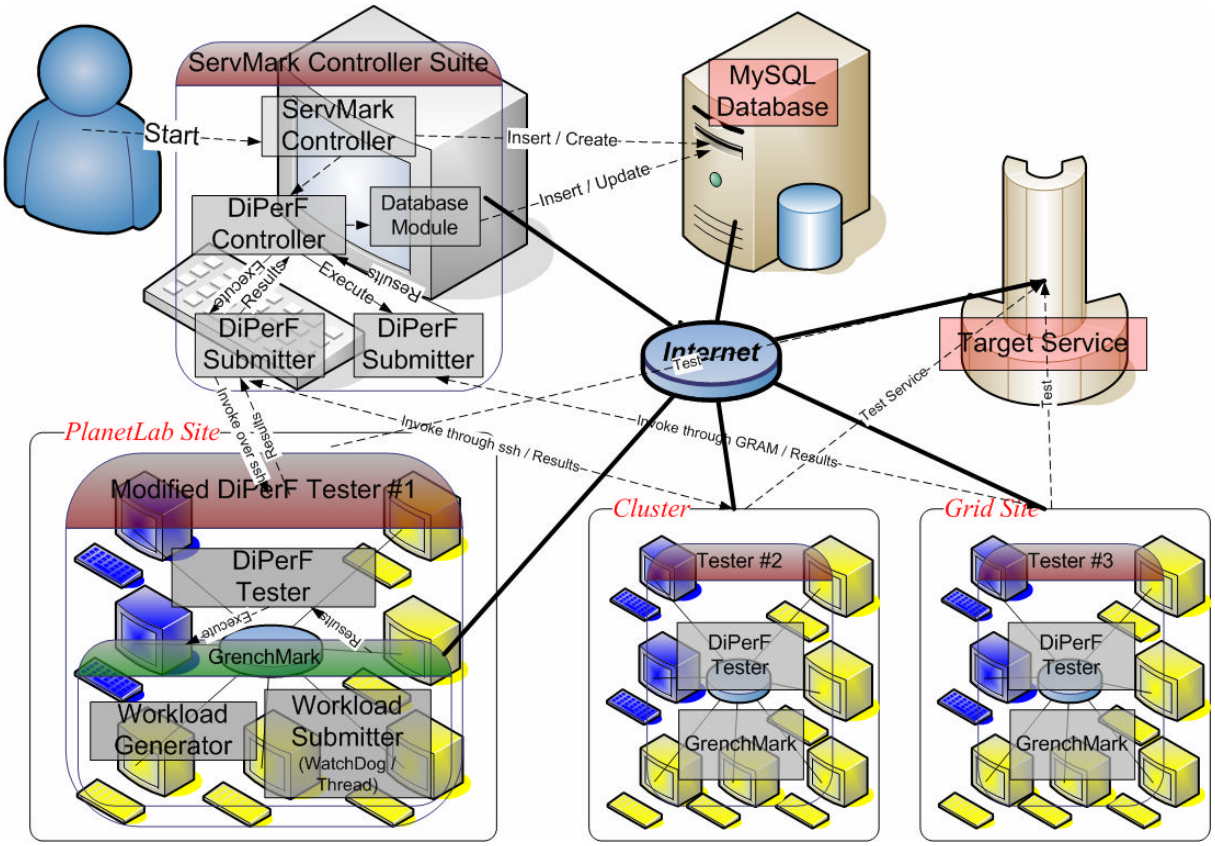

Figure 3-1. The ServMark Architecture

3.1. The ServMark controller

The ServMark controller consists of 2 files: one is a Python file (**controller.py**) and one is a bash script (**runtest.sh**). The controller is started by executing the file **runtest.sh**.

```
#!/bin/sh

rm -f diperf.cfg
rm -f sites_diperf.txt
python controller.py

cp diperf.cfg ./DiPerF.v2.0/
./DiPerF.v2.0/diperfRun.pl -a no
```

Listing 3-1. The runtest.sh file of the ServMark controller.

It parses the file containing the user-specified parameters (**test-params.in)** and assigns default values to the parameters which are not present in the file. It uses the given parameters to generate a file containing the machines on which the testers will be spawned (the file is called **sites_diperf.txt**). The file is in a format specific to DiPerF. It then generates an input file for the DiPerF controller (the file is called **diperf.cfg**).

Finally, the controller creates the corresponding tables in the database (if they do not already exist), inserts into the database the test parameters, thus generating a unique test ID and invokes the DiPerF controller.

3.2. The interaction between the user and the ServMark controller

The user places all the parameters in a file called **test-params.in**. A sample test-params.in file is the following:

```
Granularity=custom # comment 1
MonitoringInfoGathering=push    #comment 2
PushPeriod = 3000  # msec
ExecutableFileName=wget
CommandLineArguments=http://141.85.99.160:8080
NumberOfTesters=50
JobsPerTester=100
WorkloadDistribution=Poisson(1000)
SitesFile=planetlab-serv.txt
JobType=exe
LogFile=single
ProjectID="servmark project"
DBServerName=myserver
DBUserName=myusername
DBPassword=mypassword
DBName=mydb
```

Listing 3-2. A sample test-params.in file.

The `Granularity` parameter refers to the testing strategy. When testing Grid systems, the jobs usually have a run time of the order of minutes, whereas when testing web services, the jobs have a running of time of the order of tens of milliseconds. Other differences also exist, based on the way the results are sent back and the frequency at which the results are sent. This is specified by the parameters `MonitoringInfoGathering` and `PushPeriod`. Currently, however, these parameters are ignored and there is only one way used to send the results back, using a fixed period.

The `ExecutableFileName` parameter specifies the executable file name. When testing grid environments, this should be the name of the job to be executed. When testing web services, this should be the name of the client which will use the web services. The name must contain the complete path (if it cannot be located using the $PATH environment variable). The executable file must already be located on each machine on which a tester will be spawned. Currently, ServMark does not copy the executable from some location to the machines on which the testing process takes place.

The parameter `CommandLineArguments` specifies the command line arguments which will be passed to the executable file (they can be enclosed between ‘”’ if they contain white space – like `“-a x y –g no”`; if no command line arguments are given, the user must specify “”).

The parameter `NumberOfTesters` specifies the number of testers and the parameter `JobsPerTester` specifies the number of jobs which will be issued by each tester.

The parameter `WorkloadDistribution` specifies the distribution of the times at which jobs are submitted. This parameter must be given in a format specific to GrenchMark (see [18]).

The parameter `JobType` refers to the type of job and is information used by GrenchMark. Type `exe` represents a stand-alone application. Currently, there are several other types of jobs, all of which use the Koala Grid Resource Manager [44,45]. In order to test web services, type `exe` is the most likely to be used. In order to test Grid environments, a type of job must be defined for its resource manager (a job generator and a job description file printer, which will generate JDF files in a format specific to the resource manager). GrenchMark is an extensible framework and new types of jobs can easily be defined.

The parameter `LogFile` is used by GrenchMark and can take one of two values: `single` and `multiple`. The value `single` means that all the jobs write their standard output and standard error to the same file, while `multiple` means that each job uses its own file. When many jobs are issued, it is more appropriate to use only one file, in order to avoid the creation of too many files.

The parameter `ProjectID` is used as a project identifier. It is part of the primary key of some of the tables in the database, together with an auto-generated test id. It is useful in order to group together several testing processes which are part of the same project.

The parameter `SitesFile` represents the name of a file which contains a weighted list of machines on which testers will be spawned, one element on a line. A sample file is the following:

```
fs3.das2.ewi.tudelft.nl/20
s8.diperf.cs.uchicago.edu/10
alice01.rogrid.pub.ro/5
```

Listing 3-3. A sample sites file.

The numbers are separated by the names by a ‘/’ character. The number represents a weight (and can be a real number). When deciding on which machines the testers will be spawned, these weights will be considered. For instance, considering the above file and using 7 testers, 4 of them would be spawned on fs3.das2.ewi.tudelft.nl, 2 of them on s8.diperf.cs.uchicago.edu and 1 on alice01.rogrid.pub.ro.

The parameters `DBServerName`, `DBUserName`, `DBPassword` and `DBName` refer to the database where results will be stored. `DBServerName` represents the name of the machine where the database server is installed (currently, only MySQL is supported), `DBUserName` and `DBPassword` represent the username and password used to connect to the database server, and

`DBName` represents the name of the database. The database tables used by the testing process will be created (if they do not already exist) by the ServMark controller.

3.3. The modified DiPerF controller

The DiPerF controller has been slightly modified. In the command line invocation, it receives extra parameters, which will be sent to the tester. The controller invokes the DiPerF submitter and the standard output of the submitter is connected to a pipe from which the controller will read back the results. The DiPerF controller keeps reading characters from the pipe. For every complete line it receives (ended by a newline character), it checks if it is a line containing results and, if so, it sends it to the database module. A line containing results has a specific prefix, called LOGFILE_PREFIX. When sent to the database module, this prefix is stripped off.

3.4. The modified DiPerF submitter

The DiPerF submitter receives extra parameters in its command line invocation. These extra parameters will be sent to each tester. Except for these parameters, each tester receives a unique ID from the submitter (the tester IDs are consecutive integer numbers ranging from 0 to the number of testers minus 1).

The tester invocation part of the submitter has not been changed. Each tester is invoked through a SSH connection on the machine on which it needs to be executed. Its standard output is connected to a pipe. The submitter reads from the pipes connected to each tester's standard output and sends the lines read to the controller (by writing them to its own standard output).

DiPerF allows for two modes of executing a tester. In the first mode, the tester receives the name of an executable file which will be executed and needs to be located on the machine of the tester. In the second mode, the tester receives through its standard input a .tar archive which is decompressed and then a file is executed which is contained inside the archive. This is the only way files can be transferred from the machine of the controller to the machine on which each tester is executed. In ServMark, only this second mode is used. The archive contains the GrenchMark files.

3.5. The modified DiPerF tester

The DiPerF tester is given extra parameters in the command line, which are passed to GrenchMark. The tester decompresses the .tar archive given through its standard input and then executes a bash script from the archive. The bash script is located inside the archive and it provides the GrenchMark functionality. The bash script is named **runtest.sh** (it should not be confused with the file having the same name, used by the ServMark controller).

3.6. The modified GrenchMark

The execution of GrenchMark is coordinated by the commands in the bash script **grenchmark/runtest.sh** (shown in Listing 3-4).

```
#!/bin/sh

cd grenchmark
# $1 - logfile (single/multiple)
# $2 - pushperiod (msec)
# $3 - command_line_arguments
# $4 - workload distribution
# $5 - monitoring info gathering (alwasy push ?)
# $6 - executable file name
# $7 - jobs per tester
# $8 - job type
# $9 - test id
# ${10} - tester id

LOGFILE=$1
PUSH_PERIOD=$2
CMD_LINE_ARGS=$3
WL_DISTR=$4
MON_INFO_GATH=$5
EXE_NAME=$6
NUMJOBS=$7
JOBTYPE=$8
TEST_ID=$9
PROJECT_ID=${10}
TESTER_ID=${11}
START_TIME=${12}

# generate a work-load description file

./echo-params $TEST_ID$TESTER_ID unitary $NUMJOBS $JOBTYPE single 1 *:? $WL_DISTR
"cmdline=$EXE_NAME $CMD_LINE_ARGS" >wl-desc.in

if [ -d ./out/ ] ; then cd ./out/ ; rm -f -r * ; cd .. ; else mkdir out ; fi

python wl-gen.py -j $JOBTYPE-jdf

if [ -d ./out/run/ ] ; then cd ./out/run/ ; rm -f -r * ; cd ../.. ; else mkdir out/run ; fi

if [ "$LOGFILE" = "single" ] ; then ONEFILE=--onefile ; fi

python wl-submit.py out/wl-to-submit.wl --nobackground $ONEFILE --testid=$TEST_ID --
testerid=$TESTER_ID "--projectid=$PROJECT_ID" --starttime=$START_TIME 2>errlog.err
```

Listing 3-4. The runtest.sh file of GrenchMark.

The script receives 12 parameters in its command line invocation (the significance of the parameters is well-explained inside the file).

3.6.1. Generating a workload description file

The executable file **echo-param**s (the source file is **echo-params.c**) is invoked in order to create a workload description file (called **wl-desc.in**). The workload description file is written in a format specific to GrenchMark.

3.6.2. Generating a workload file

The GrenchMark file **wl-gen.py** is used to generate a workload file from the workload description file. In order to do this, jobs must be of a known type. Currently, there are several types which use the Koala Grid Resource Manager and one type for executing stand-alone applications (type exe). In order to use other Grid resource Managers, new JDF (Job Description File) printers must be written, which produce JDF files in the format specific to the Grid Resource Manager. In order to define new types of jobs, new workload generators must be defined: these must be python modules, implementing a given interface.

3.6.3. Submitting the workload

The Python file **wl-submit.py** reads the XML file written by **wl-gen.py** and actually submits the jobs for execution. The wl-submit.py file has been modified to receive extra parameters (test id, project id, tester id, start time) used for reporting the results.

The jobs are submitted for execution at specific times and a thread pool is used for submitting the jobs. A watchdog is used to check for threads which might be blocked waiting for their job to execute and in order to report periodic statistical results for each thread.

3.6.3.1. The Thread pool

The original behavior of the worker threads in the thread pool has been modified. In the original GrenchMark, each thread would get a job request from a job request queue and then execute a callback function given as a parameter in the job request. The callback function would actually submit the job for execution, compute all the needed values and return a result object to the worker thread. Now, the work of the callback function is partly done inside the worker thread. The callback functions is passed as a parameter a function of the worker thread (called **runningProcess**), which is called right before submitting the job. This function inside the thread actually submits the job and computes the most important values and the callback function compute only the remaining values contained in the result object.

Each worker thread has a Cstats object for each metric it computes. This object is fed individual values computed for each job and is used to compute statistical values for the corresponding thread.

In order to obtain the process ID of the executed job and a pipe to its standard output, the worker thread uses an object of type subprocess, contained in the popen5 package[1]. Currently, Python does not offer any possibility to obtain both the process ID and a pipe connected to the standard output.

A job is not executed directly. Instead, a wrapper is being used, called **waiter.py**. This wrapper changes its process group ID and then executes the job. This is useful in case the job spawns many processes and then blocks, because using a single kill command, all the spawned can be killed, because they would be part of the same (known) process group. This approach is useless in case the job changes its process group ID itself, in which case the process group will not be known or if each process spawned by the job changes its process group ID.

3.6.3.2. The Watchdog

The WatchDog is implemented as an extension of the class of worker threads, because it has a similar behavior. It periodically checks if any threads are blocked waiting for their associated job to finish execution. If there are any threads blocked for a period longer than a specified amount of time, the job is killed by the watch dog. After the job is killed, the worker thread regains control as if the job had terminated normally. By inspecting the return code, the worker thread could notice that the job was, in fact, killed by the watchdog.

The watchdog has another important function. It periodically collects statistical information from the worker threads, for every computed metric. Currently, there are 5 metrics computed: Run Time, Response Time, Waiting Time, Time to Job Failure and Time To Job Completion. Each metric is computed on a per thread basis.

3.7. The Database

We used a MySQL database containing 5 tables, which will be described next.

3.7.1. The metric_type_mapping Table

This table contains the name of the metrics and their types (the type is given as a string: for instance, “float” or “int”).

The SQL command used to create this table is:

```
create table metric_type_mapping (metric_name varchar(50) NOT NULL, metric_type
VARCHAR(50), PRIMARY KEY(metric_name));
```

Listing 3-5. SQL command for creating the metric_type_mapping table.

[1] The popen5 package can be downloaded from www.lysator.liu.se/~astrand/popen5.

3.7.2. The test_params Table

This table contains the most important parameters of a testing process. Entries are being added by the ServMark controller. A test ID is generated automatically when a new entry is inserted into this table.

The SQL command used to crate this table is:

```
create table test_params (testid INT AUTO_INCREMENT NOT NULL, projectid VARCHAR(40)
NOT NULL, date DATETIME, test_params BLOB, PRIMARY KEY(testid, projectid));
```

Listing 3-6. SQL command for creating the test_params pping table.

The projectid field is part of the primary key and can be used in order to group together multiple testing processes which are part of the same process. The date field records the moment when the entry was inserted into the table.

A row of this table could look like this:

11	awhttpd	2006-08-01 05:32:10	DBUserName=servmark LogFile=single PushPeriod=3000 CommandLineArguments=http://141.85.99.160:59876/ WorkloadDistribution=Poisson(1000) ProjectID="awhttpd" DBPassword=dbpass JobsPerTester=100 DBName=servmark MonitoringInfoGathering=pull SitesFile=planetlab-serv.txt DBServerName=localhost Granularity=custom NumberOfTesters=50 ExecutableFileName=wget JobType=exe

3.7.3. The test_logs table

This table contains all the information written to standard output or standard error by the submitted jobs. This information can later be used in order to compute extra information.

The SQL command used to create this table is:

```
create table test_logs (testid INT NOT NULL, projectid VARCHAR(50) NOT NULL, testerid INT
NOT NULL, jobid INT NOT NULL, log BLOB, PRIMARY KEY(testid, projectid, testerid, jobid),
```

```
CONSTRAINT FOREIGN KEY (testid, projectid) REFERENCES test_params(testid, projectid) );
```

Listing 3-7. SQL command for creating the test_logs table.

A row of this table could look like this:

6	"testing web services"	3	7816277	Took 0.167 seconds… Execution terminated successfully

3.7.4. The statistical_values table

This table contains all the statistical information computed during the testing process.

The SQL command used to create this table is:

```
create table statistical_values (testid INT NOT NULL, projectid VARCHAR(50) NOT NULL,
testerid INT NOT NULL, xid INT NOT NULL, metric VARCHAR(50) NOT NULL,  instime
DATETIME NOT NULL, time DOUBLE (20,8) NOT NULL, min DOUBLE(10,8), max
DOUBLE(10,8), avg DOUBLE(10,8), stddev DOUBLE(10,8), cov DOUBLE(10,8), nsamples INT,
total DOUBLE(10,8), PRIMARY KEY(testid, projectid, testerid, xid, metric, instime, time),
CONSTRAINT FOREIGN KEY (testid, projectid) REFERENCES test_params(testid, projectid),
CONSTRAINT FOREIGN KEY(metric) REFERENCES metric_type_mapping(metric_name)  );
```

Listing 3-8. SQL command for creating the statistical_values table.

The xid field represents the ID of the worker thread for which the metric was computed. The metric field represents the metric name. The instime field represents the moment when the entry was inserted into the table. The time field represents a moment, in seconds, when the statistical information was collected. This moment is synchronized among all the testers, that is, the same value on two different testers represents the same moment in time.

The statistical information gathered contains the minimum and maximum value, the average, standard deviation, covariance, the sum of all the values and the number of samples which were used to compute the information.

A row of this table could look like this:

11	tst	20	6266	Response_Time	2006-08-01 05:38:51	13.2	1.7	4.1	2.7	1.07	0.39	4	10.99

3.7.5. The individual_values Table

This table contains important individual values. The SQL command used to create the table is:

```
create table individual_values (testid INT NOT NULL, projectid VARCHAR(50) NOT NULL,
testerid INT NOT NULL, xid INT NOT NULL, metric_name varchar(50) NOT NULL, instime
```

```
DATETIME NOT NULL, time DOUBLE(20,8) NOT NULL, value
 VARCHAR(50), PRIMARY KEY(testid, projectid, testerid, xid, metric_name, instime, time),
CONSTRAINT FOREIGN KEY(testid, projectid) REFERENCES test_params(testid, projectid),
CONSTRAINT FOREIGN KEY(metric_name) REFERENCES
metric_type_mapping(metric_name));
```

Listing 3-9. SQL command for creating the individual_values table.

Currently, the only individual values recorded are job failures. The "value" of a failure is the return code of the job (this way, we can distinguish between failures generated by jobs running for too long and "normal" failures).

A row of this table could look like this:

10	jetty	20	82892	Failure	2006-08-01 04:20:18	1154441759.75	-9

3.8. The Database Module

The database module is implemented in Python (the file **dbpy.py**). It receives as a single command line argument a line which contains information to be entered into the database. Information is encoded. The fields are separated by the character having ASCII code 1 and the line may contain a prefix which specifies the table into which the information will be inserted (or updated). The DiPerF controller invokes the database module every time it receives a line containing information to be entered into the database (such a line has a particular prefix).

3.9. The Metrics

Currently, there are 5 metrics computed: Run Time, Response Time, Waiting Time, Time To Job Completion and Time To Job Failure. All of them are computed on a per thread basis. Currently, because of insufficient information, the waiting time is always considered to be 0 and the run time is always equal to the response time. The relations ship between them is: Response Time = Waiting Time + Run Time. However, once a job is submitted, there is no module implemented to measure the waiting time (get it from the resource manager), so we consider the waiting time to be 0.

The Time To Job Failure metric is computed for approximately equal intervals of time. For each failed job, the difference between the moment it failed and the previously moment when a job has failed (or the beginning of the time interval) is computed and passed to the corresponding Cstats module. This metric is a measure of how frequently job failures occur.

The Time To Job Completion metric is computed in a similar way. For every correctly completed job, the difference between the previous moment when a job was completed correctly (or the beginning of the time interval) is computed and passed to the corresponding Cstats module.

3.10. Reliability

The reliability of the current implementation is bounded by the reliability of the design. The details of the implementation introduced new challenges, however, but solved some of the problems regarding potential crashes which could not be addressed at design level.

The ServMark controller is not expected to crash at all, unless the database is not accessible, in which case the testing process should not go any further, anyway. The DiPerF controller is not expected to crash except when the DiPerF submitter cannot be properly located, which denotes an improper configuration. In this case, the testing process should not go any further. As soon as the testing process begins, the DiPerF controller is not expected to crash. The DiPerF submitter is expected to crash only in case some of the command-line arguments are invalid. However, as soon as the testing process starts, no crashes should be expected. The Database module may crash in case the database becomes inaccessible. The database may become inaccessible at any time (unless further guarantees are given). If the database crashes during a testing process, the testing process will carry on normally, except that most of the gathered information will be lost. Some bits of information are also stored on the testing nodes, i.e. most of the information which is normally stored in the test_logs table of the database. The DiPerF tester is expected to crash only in the case of invalid command-line arguments, but not after the testing process has successfully started.

The execution of GrenchMark has two major points of failure. One is at the moment when the workload description file is generated. If the given arguments are inappropriate, a workload description file might not be generated. Without a workload description file, no amount of testing will take place. The second point of failure is the job submission. If the job execution parameters printed in the workload description file are invalid, the jobs will not be executed.

All the crashes that were mentioned are “silent” crashes and represent the natural behavior for the corresponding situation. No uncotrolled crashes are expected to occur anywhere in the implementation of ServMark. In case a controlled crash occurs, the whole testing process needs to be restarted and the causes of the crash need to be addressed.

4. Validation and Testing

4.1. Validation

We have validated the implementation on the DAS-2 environment [1], a wide-area distributed system consisting of 200 Dual Pentium-III computer nodes. The environment is built out of clusters of workstations, which are interconnected by SurfNet, the Dutch university Internet backbone for wide-area communication, whereas Myrinet, a popular multi-Gigabit LAN, is used for intra-cluster communication. The clusters are located at five Dutch Universities and from this point of view it can be considered as an experimental Grid system operating in the Netherlands. The validation focus was to show that ServMark can operate correctly, that is, that it can generate complex tests involving several test nodes, run the tests, obtain and analyze the results, and store all the produced output. We have used one node in each cluster to validate our

implementation, by running on each of them several ServMark test nodes. Throughout the validation tests, ServMark displayed the expected functionality.

4.2. Testing

In order to test the ServMark implementation, we chose to evaluate the performance of 6 web servers: Apache, Null HTTPD, Apache Tomcat, Nweb, Jetty and Awhttpd. The purpose of this testing scenario was to prove the capabilities of our system and not to establish which of these web servers is the best, from an absolute point of view.

4.2.1. Experimental Setup

The ServMark "core" was installed on **s8.diperf.cs.uchicago.edu** , a machine located at the University of Chicago Computer Science Department. The characteristics of this machine are presented in table 4-1.

OS	Linux SuSE
GCC version	3.3.3
Python version	2.3.3
Database Server	MySQL
MySQL version	4.0.18

Table 4-1. The characteristics of the machine on which the ServMark "core" was installed

The web servers were started on **alice01.rogrid.pub.ro**, a machine located at the Politehnica University of Bucharest, Faculty of Computer Science. The characteristics of this machine are presented in table 4-2.

OS	Linux
GCC version	3.2.3
Java version	1.5 SE

Table 4-2. The characteristics of the machine on which the web servers were started

4.2.2. Test Setup Overview

For every test, we used 22 testers, each executing 100 requests, generated using a Poisson distribution. The testers were spawned on machines which are part of PlanetLab.

PlanetLab currently consists of 693 machines, hosted by 335 sites, spanning over 25 countries. Most of the machines are hosted by research institutions, although some are located in co-location and routing centers (e.g., on Internet2's Abilene backbone). All of the machines are connected to the Internet. All PlanetLab machines run a common software package that includes a Linux-based operating system; mechanisms for bootstrapping nodes and distributing software updates; a collection of management tools that monitor node health, audit system activity, and control system parameters; and a facility for managing user accounts and distributing keys. The

advantage to researchers in using PlanetLab is that they are able to experiment with new services under real-world conditions, and at large scale.

For each test, the testers were selected to run on hosts from North and South America, Asia, and Europe, simultaneously.

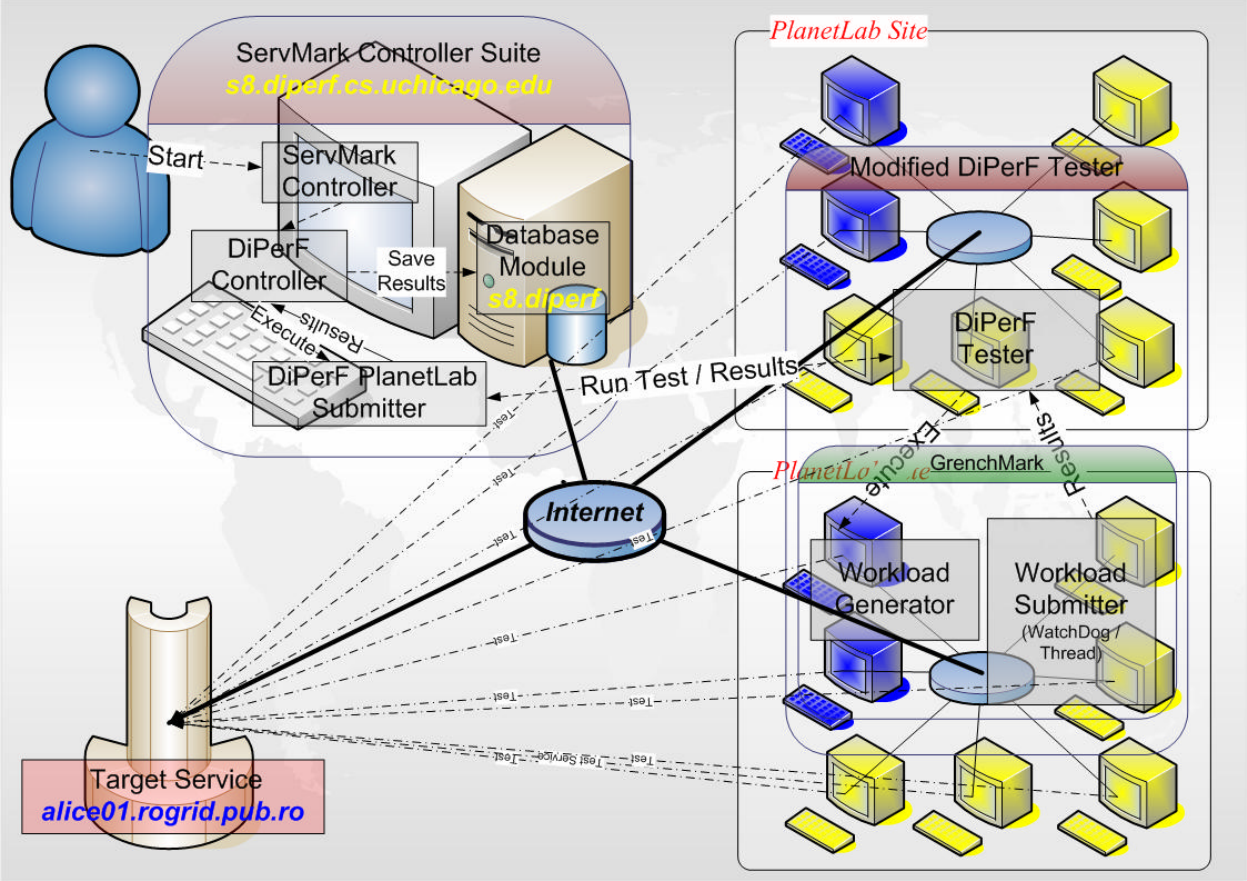

Figure 4-1. The Test Setup Overview

A job which was running for more than 25 seconds was considered to be blocked and was, subsequently, killed.

The watchdog gathered statistical information from the worker threads approximately every 15 seconds.

4.2.3 Test Results

Web Server	Average(Standard Deviation)	Minimum	Maximum	Weighted Average
Apache	1.0779 (0.647)	0.0810	16.5440	1.0969
Null HTTPD	0.9442 (0.482)	0.1244	30.4872	0.9495
Apache Tomcat	1.3617 (0.732)	0.1724	24.2665	1.3930
Nweb	0.9731 (0.565)	0.1293	10.9908	1.0152
Jetty	10.0745 (1.210)	0.2651	35.4375	9.0297

Awhttpd	1.1739 (0.558)	0.1242	29.5580	1.0117

Table 4-3. The Response Times computed for the 6 web servers (in seconds).

Table 4-3 presents the statistical values for the response time of the 6 web servers we tested. The SQL command used to obtain these values from the 'statistical_values' table is: **"select testid, avg(avg), avg(stddev), min(min), max(max), sum(avg * nsamples) / sum(nsamples) from statistical_values where nsamples > 0 and metric = 'Response_Time' group by testid;"**

For the selected test scenario, the results have shown the existence of three classes of web servers: very fast, fast and slow. The very fast class contains the fastest web server Nweb, with Null HTTPD and Apache coming close second and third, respectively. The fast class contains the Apache Tomcat web server, which is 30% slower than its non-services-enabled counterpart, and Awhttpd. Finally, the slow class contains the Jetty web server, which is at least 8-10 times slower than all the others.

We notice very large response times in the case of the Jetty web server, compared to the other 5 servers. These response times could be explained by the fact that Java code is usually slower than C/C++ code and the Java version used on the machine where the web servers were started is Standard Edition, which does not provide a lot of code optimization. Besides, the PlanetLab environment is being used for the testing and development of many projects and it is possible that during the testing process of the Jetty web server, the machines used for testing may have been extra loaded.

The web server achieving the fastest average response time was Null HTTPD, a tiny web server, followed by Nweb, but the web server obtaining the minimum response time among all the requests is Apache. Looking at the variability of the response time, the observed standard deviation lies within 10% of the average, for each server. However, the maximum response time outliers, which show the robustness of the response time, range from 10-15 times higher than the average (e.g., NWeb and Apache) to 20-35 times. We conclude that, for the selected test scenario, NWeb and Apache are the best performers, followed by Null HTTPD, Apache Tomcat, and Awhttpd (with lower performance or robustness), and then, at some distance, Jetty.

Web Server	Average(Standard Deviation)	Minimum	Maximum	Weighted Average
Apache	3.8803 (1.975)	0.0022	13.5419	3.6702
Null HTTPD	3.9409 (1.922)	0.0177	11.7235	3.7446
Apache Tomcat	4.0902 (2.061)	0.0034	13.8347	3.8399
Nweb	4.0870 (2.008)	0.0393	14.1707	3.8613
Jetty	6.4677 (1.582)	0.0010	15.0310	5.9648
Awhttpd	4.1798 (2.041)	0.0106	13.9180	3.9005

Table 4-4. The Times To Job Completion computed for the 6 web servers (in seconds).

Table 4-4 presents the statistical values for the time to job completion of the 6 web servers we tested. The SQL command used to obtain these values from the 'statistical_values' table is: **"select testid, avg(avg), avg(stddev), min(min), max(max), sum(avg * nsamples) /**

sum(nsamples) from statistical_values where nsamples > 0 and metric = 'Time_To_Job_Completion' group by testid;"

The average TTJC is higher than the average Response Time due to the workload structure and of the environment performance. TTJC, which is the time difference between two consecutive successful job finishes, is computed on a per thread basis, so it is affected by the inter-arrival time difference, and by failures. Furthermore, the testing nodes were not multi-processor machines, so the TTJC was also affected by the thread scheduling policy of the operating system and; therefore, it is natural that the TTJC is higher than the response time.

Although affected by factors which do not depend on the tested web server, the results based on TTJC measurement seem to be consistent with our previous conclusions, that Apache, Nweb and Null HTTPD achieved the best performance for this test scenario.

Web Server	Average(Standard Deviation)	Minimum	Maximum	Weighted Average
Apache	No Failures	-	-	-
Null HTTPD	2.7893 (0.000)	0.0000	5.5786	2.7893
Apache Tomcat	No Failures	-	-	-
Nweb	No Failures	-	-	-
Jetty	1.4840 (0.000)	0.000	17.8760	1.4840
Awhttpd	No Failures	-	-	-

Table 4-5. The Times To Job Failure computed for the 6 web servers (in seconds).

Table 4-5 presents the statistical values for the time to job failure of the 6 web servers we tested. The SQL command used to obtain these values from the 'statistical_values' table is: **"select testid, avg(avg), avg(stddev), min(min), max(max), sum(avg * nsamples) / sum(nsamples) from statistical_values where nsamples > 0 and metric = 'Time_To_Job_Failure' group by testid;"**

Analyzing the Time To Job Failure, we notice that in the case of NullHTTPD and Jetty, some failures did occur. Examining the individual_values tables, we concluded that all of these failures occurred because the requests exceeded the allotted time of 25 seconds. This could have happened for several reasons: either the machine on which the failure occurred was too loaded and the request was delayed or the machine on which the web server was running became too loaded. Ideally, we would not want the machines on which the testers were running to become too loaded, but we have little control over the load of the machines which are part of PlanetLab.

The amount of data stored in the database generated by ServMark for each web server was estimated to be about 1.6 megabytes. The test_logs table contained approximately 1 megabyte for each test case. Information about each of the 2200 jobs (22 testers x 100 jobs/tester) was stored in the test_logs table. The size of this information is 450 bytes for each submitted job. This value depends on the output of each test job, which is application-specific, and cannot be reduced by the testing infrastructure (i.e.g, by GrenchMark).

The statistical_values table contained approximately 615 kilobytes for each test. This includes statitical information about all the 5 metrics, gathered every 15 seconds on a per thread basis. The amount of data stored in this table depends on the number of testers and the overall duration of the testing process. If the testing nodes are fast and not too loaded, less data will be stored in the database. The number of jobs affects the amount of information only indirectly, as more jobs will take more time to complete. This information also depends on the number of worker threads in the thread pool used by GrenchMark, but this value is currently set to 5 and there are no options by which it could be changed by the user.

The test parameters we chose (22 testers and 100 queries per tester) were large enough to make good use of the resources available at the testing nodes. However, they may not have been stressing enough to make the web servers use all of their resources. More realistic testing parameters would be on the order of 10,000 testing nodes and 1,000 queries generated by each testing node. Grid resource managers were not tested at all. Some good choices for a realistic test of a Grid resource manager would be 5,000 testing nodes and 15,000 jobs per node.

4.3. Undesirable behavior

During the testing process, we noticed several peculiar behavior pattern. One such undesired behavior is represented by the DiPerF tester blocking indefinitely while waiting for data to be written by GrenchMark to the standard output. GrenchMark submits all the jobs properly and writes to the standard output statistical information, as well as logging information. The tester receives this information and passes it up to the submitter. However, the last pieces of information written to the standard output, right before GrenchMark terminates gracefully, do not reach the tester, although they should. If this failure occurs, the tester never terminates and these last bits of information do not reach the database. This behavior was noticed repeatedly, but not as long as the tester and the submitter were executed on the same machine.

By examining the data stored in the individual_values table (which, at the moment, only stores individual failures), we noticed that the value field was, in the case of 4 entries, equal to ‘None’, instead of the “normal” value ‘-9’, which denotes that the corresponding job has timed out. This shows that there is a problem with obtaining the return code of the executed job. This is probably due to the implementation of the popen5 library we used for executing the jobs, where the return code might be retrieved after the job finished executing and the control was passed to the worker thread which executed the job. A simple solution to this problem would be to repeatedly poll the return code (with some amount of “sleep” in between), until its value is different from ‘None’.

5. Related Work

A significant number of projects have tried to tackle the Grid performance assessment problem from different angles: modeling workloads and simulating their run under various environment assumptions [3, 5, 15], attempting to produce a representative set of grid applications like the NAS Grid Benchmarks [8], creating synthetic applications that can assess the status of grid services like the GRASP project [4] and the Grid Exerciser[2], and creating tools for launching

[2] The Grid Exerciser (GEx) is available online at http://www.cs.wisc.edu/condor/tools/exerciser/

benchmarks/application-specific functionality tests like the GridBench project [13] and the NMI[43] projects [43]. ServMark is the natural complement to these approaches, by offering a much larger application base, more advanced workload modeling features, and the ability to replay existing workload traces. In addition, ServMark can be used for much more than just Grid performance evaluation.

The modeling/simulation approach is almost exclusively based on traces which are now part of the Parallel Workloads Archive. The major hurdle for this approach is to prove the representativeness of simulation results for real grid environments.

In [8], the authors propose a small set of parallel applications as Grid benchmarks. Simple workloads are defined for the applications, in that the running parameters and the order in which the applications are to be run are fixed. The drawbacks of this approach are that the applications are only representative for a restricted research area (here, computational fluid dynamics), make very little use of Grid components (only Grid-enabled MPI and a scheduler), and cannot adapt to the dynamic behavior of Grids (they require fixed resource sizes, and have no fault-tolerance, migration, or check-pointing features).

In [4], a small set of applications are specifically designed to test specific aspects of Grids functionality (*probes*). The applications assume the existence of common Grid components, like a global information system, or a file-transferring service. No attempt to form workloads with these applications is made.

In [13], a benchmark launching tool is proposed. This tool has the ability to launch benchmarks and display their results, and can be coupled with many of the existing HPC benchmarks. However, it has very limited workload modeling features, and cannot replay real traces.

NMI [43] facilitates the definition and run of functionality tests. It currrently lacks the ability to define complex workloads, specific for performance and scalability testing.

Many studies have investigated the performance of individual Grid services. As an example, Zhang et al. [26] compare the performance of three resource selection and monitoring services: the Globus Monitoring and Discovery Service (MDS), the European Data Grid Relational Grid Monitoring Architecture (R-GMA), and Hawkeye. Their experiment uses two sets of machines (one running the service itself and one running clients) in a LAN environment. The setup is manual and each client node simulates 10 users accessing the service. This is exactly the scenario where ServMark would have proved its usefulness: it would have freed the authors from deploying clients, coordinating them, and collecting performance results, and allow them to focus on optimally configuring and deploying the services to test, and on interpreting performance results.

The Globus Toolkit's job submission service test suite [27] uses multiple threads on a single node to submit an entire workload to the server. However, this approach does not gauge the impact of a wide-area environment, and does not scale well when clients are resource intensive which means that the service will be relatively hard to saturate.

The Network Weather Service (NWS) [28, 29] is a distributed monitoring and forecasting system. A distributed set of performance sensors feed forecasting modules. There are important differences to ServMark. First, NWS does not attempt to control the offered load on the target service but merely to monitor it. Second, the performance testing framework deployed by ServMark is built on the fly, and removed as soon as the test ends, while NWS sensors aim to monitor services over long periods of time. Similarly, NETI@home [30], Gloperf [31], and NIMI [32] focus on monitoring service or network level performance.

NetLogger [33] targets instrumentation of Grid middleware and applications, and attempts to control and adapt the amount of instrumentation data produced in order not to generate too much monitoring data. NetLogger is focusing on monitoring, and requires code modification in the clients; furthermore, it does not address automated client distribution or automatic data analysis. Similarly, the CoSMoS system [34] is geared toward generic network applications.

GridBench [35] provides benchmarks for characterizing Grid resources and a framework for running these benchmarks and for collecting, archiving, and publishing results. While DiPerF focuses on performance exploration for entire services, GridBench uses synthetic benchmarks and aims to test specific functionalities of a Grid node. However, the results of these benchmarks alone are probably insufficient to infer the performance of a particular service.

Finally, Web server performance has been a topic of much research. The Wide Area Web Measurement (WAWM) Project for example designs an infrastructure distributed across the Internet allowing simultaneous measurement of web client performance, network performance, and web server performance [36]. Banga et al. [37] measure the capacity of web servers under realistic loads. Both systems could have benefited from a generic framework such as ServMark.

6. Conclusion and Ongoing Work

In this paper we have presented ServMark, a distributed system for testing Grid environments and Grid and web services. We have described its design and we have successfully implemented the system. The implementation was tested first on DAS and then, using PlanetLab to deploy the testers, we have evaluated the performance of 6 web servers. The system measured up to its expectations.

Currently, we are working on improving ServMark in several directions. First, we are trying to improve the interface between the user and the ServMark controller. Second, we are thinking about alternative ways to send the information from the testers to the controller, without using pipes connected to the standard output. Third, we are working towards making ServMark a fault-tolerant grid service.

We are going to improve the database module in order to support more database servers, not just MySQL. In terms of provided functionality, we are thinking about ways to create more elaborate testing scenarios (at the very least, be able to specify different parameters for each tester, in order to use different workloads).

ServMark is, basically, composed of GrenchMark, DiPerF and an interface between these two. We are trying to create a more flexible interface, so that DiPerF and GrenchMark be more loosely coupled. A flexible interface between DiPerF and GrenchMark would allow ServMark to easily make use of future improvements in both DiPerF and GrenchMark (future versions would be easily integrated).

Acknowledgements

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). Part of this work was also carried out in the context of the Virtual Laboratory for e-Science project (http://www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC\&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). This work was also supported by the EU-NCIT – NCIT leading to EU IST excellency project, EU FP6-2004-ACC-SSA-2.

References

[1] H. E. Bal et al. The distributed ASCI supercomputer project. *Operating Systems Review*, 34(4):76-96, October 2000.

[2] F. Berman, A. Hey, and G. Fox. *Grid Computing: Making The Global Infrastructure a Reality*. Wiley Publishing House, 2003.

[3] A. I. D. Bucur and D. H. J. Epema. Trace-based simulations of processor co-allocation policies in multiclusters. In *Proc. of the 12th IEEE HPDC*, pages 70-79. IEEE Computer Society, 2003.

[4] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. In *IPDPS*. IEEE Computer Society, 2004.

[5] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On advantages of grid computing for parallel job scheduling. In *CCGRID*, pages 39-49. IEEE Computer Society, 2002.

[6] C. Ernemann, B. Song, and R. Yahyapour. Scaling of workload traces. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *JSSPP*, volume 2862 of *LNCS*, pages 166--182. Springer, 2003.

[7] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *JSSPP*, volume 1459 of *LNCS*, pages 1-24. Springer, 1998.

[8] M. Frumkin and R. F. V. der Wijngaart. Nas grid benchmarks: A tool for grid space exploration. *Cluster Computing*, 5(3):247-255, 2002.

[9] H. Li, D. Groep, and L. Wolters. Workload characteristics of a multi-cluster supercomputer. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *JSSPP*, LNCS, vol.3277, pages 176-194. Springer, 2004.

[10] H. Mohamed and D. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *Proc. Of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*, May 2005.

[11] W. Smith, I. Foster, and V. Taylor. Predicting application run times with historical information. *J. Parallel Distrib. Comput.*, 64(9):1007-1016, 2004.
[12] A. Snavely, G. Chun, H. Casanova, R. F. V. der Wijngaart, and M. A. Frumkin. Benchmarks for grid computing: a review of ongoing efforts and future directions. *ACM SIGMETRICS Perform. Eval. Rev.*, 30(4):27-32, 2003.
[13] G. Tsouloupas and M. D. Dikaiakos. GridBench: A workbench for grid benchmarking. In P. M. A. Sloot, A. G.Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *EGC*, volume 3470 of *LNCS*, pages 211-225. Springer, 2005.
[14] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency & Computation: Practice & Experience.*, 17(7-8):1079-1107, June-July 2005.
[15] A. M. Weil and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529-543, 2001. [16] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Rec.*, 34(3):44-49, 2005.
[17 http://diperf.cs.uchicago.edu/
[18] http://grenchmark.st.ewi.tudelft.nl/
[19] L. Peterson, T. Anderson, D. Culler, T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", The First ACM Workshop on Hot Topics in Networking (HotNets), October 2002.
[20] A. Bavier et al., "Operating System Support for Planetary-Scale Services", Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI), March 2004.
[21] Grid2003 Team, "The Grid2003 Production Grid: Principles and Practice", 13th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC-13) 2004.
[22] The Globus Alliance, www.globus.org.
[23] Foster I., Kesselman C., Tuecke S., "The Anatomy of the Grid", International Supercomputing Applications, 2001.
[24] I. Foster, C. Kesselman, J. Nick, S. Tuecke. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration." Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.
[25] The Globus Alliance, "WS GRAM: Developer's Guide", http://www-unix.globus.org/toolkit/docs/3.2/gram/ws.
[26] X.Zhang, J. Freschl, J. M. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems", Proceedings of HPDC-12, June 2003.
[27] The Globus Alliance, "GT3 GRAM Tests Pages", http://www-unix.globus.org/ogsa/tests/gram.
[28] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service", Journal of Cluster Computing, Volume 1, pp. 119-132, Jan. 1998.
[29] R. Wolski, N. Spring, J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," Future Generation Computing Systems, 1999.
[30] Charles Robert Simpson Jr., George F. Riley. "NETI@home: A Distributed Approach to Collecting Endto-End Network Performance Measurements." PAM 2004.

[31] C. Lee, R. Wolski, I. Foster, C. Kesselman, J. Stepanek. "A Network Performance Tool for Grid Environments," Supercomputing '99, 1999.
[32] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. "An architecture for large-scale internet measurement." IEEE Communications, 36(8):48–54, August 1998.
[33] D. Gunter, B. Tierney, C. E. Tull, V. Virmani, On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service, 4th International Workshop on Grid Computing, Grid2003, November 2003.
[34] Ch. Steigner and J. Wilke, "Isolating Performance Bottlenecks in Network Applications", in Proceedings of the International IPSI-2003 Conference, Sveti Stefan, Montenegro, October 4-11, 2003.
[35] G. Tsouloupas, M. Dikaiakos. "GridBench: A Tool for Benchmarking Grids," 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 2003.
[36] P. Barford ME Crovella. Measuring Web performance in the wide area. Performance Evaluation Review, Special Issue on Network Traffic Measurement and Workload Characterization, August 1999.
[37] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation), 1999.
[38] N. Minar, "A Survey of the NTP protocol", MIT Media Lab, December 1999, http://xenia.media.mit.edu/~nelson/research/ntp-survey99
[39] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, "A Resource Management Architecture for Metacomputing Systems", IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.
[40] Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and Practice in Parallel Job Scheduling. In Feitelson, D.G., Rudolph, L., eds.: Proc. of the 3rd Int'l. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). Volume 1291 of Lecture Notes in Computer Science., Geneva, Springer-Verlag (1997) 1–34.
[41] A. Iosup, D.H.J. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, R. Yahyapour, On Grid Performance Evaluation using Synthetic Workloads, In The 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), held in conjunction with SIGMETRICS'06, Jun 26, 2006, Saint Malo, FR.
[42] A. Iosup, C. Dumitrescu, D.H.J. Epema, H. Li, L. Wolters, How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications, The 7th IEEE/ACM International Conference on Grid Computing (Grid), Barcelona, September 28-29, 2006.
[43] Andrew Pavlo, Peter Couvares, Rebekah Gietzel, Anatoly Karp, Ian D. Alderman, and Miron Livny, The NMI Build & Test Laboratory: Continuous Integration Framework for Distributed Computing Software, The 20th USENIX Large Installation System Administration Conference (LISA), Washington, D.C., December 3–8, 2006 (accepted)
[44] H.H. Mohamed and D.H.J. Epema, An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters, CLUSTER 2004, IEEE Int'l Conference Cluster Computing 2004, September 2004.
[45] H.H. Mohamed and D.H.J. Epema, Experiences with the KOALA Co-Allocating Scheduler in Multiclusters, Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005), Cardiff, pp. 784-791, May 2005.

Appendix A. Installing ServMark

The installation steps are described in Listing A-1.

```
wget http://diperf.cs.uchicago.edu/ServMark-sources/ServMark.tar.gz
tar xzvf ServMark.tar.gz
cd ServMark
autoconf
./configure
make img
-> edit test-params.in in order to setup your own test parameters
./runtest.sh
```

Listing A-1. Installation steps for ServMark.

Appendix B. Installing the Web Servers used for testing

Apache

Apache is a widely used web server. It can be downloaded from http://www.apache.org/. Installation instructions can be found on the same web site.

Awhttpd

Abyss Web Server is a compact web server available for Windows, MacOS X, Linux, and FreeBSD operating systems. Despite its small footprint, it supports HTTP/1.1, dynamic content generation through CGI/FastCGI scripts, ISAPI extensions, native ASP.NET support, Server Side Includes (SSI), custom error pages, password protection, IP address control, anti-leeching, and bandwidth throttling.

Installation steps are given in Listing B-1.

```
wget http://www.hcsw.org/awhttpd/awhttpd-3.0.6.tgz
tar xzvf awhttpd-3.0.6.tgz
cd awhttpd
make
-> edit some sources to include <string.h> and <stdlib.h> (error.c, permcheck.c and misc.c)
make (again)
./awhttpd server_root_directory port
```

Listing B-1. Installation steps for awhttpd.

Null HTTPD

Null httpd is a tiny web server which is designed to be very small, simple, multithreaded, and available for Linux and Windows

Installation steps are given in Listing B-2.

```
wget http://switch.dl.sourceforge.net/sourceforge/nullhttpd/nullhttpd-0.5.1.tar.gz
tar xzvf nullhttpd-0.5.1.tar.gz
cd nullhttpd-0.5.1/src
make
cd ../httpd/etc
cp httpd.cfg-sample httpd.cfg
-> edit httpd.cfg : set another port
cd ../bin
./httpd
```

Listing B-2. Installation steps for null httpd.

Nweb

Nweb is a very small web server, consisting of only 200 lines of C code. It provides error checking and only handles static pages so it is safe.

Installation steps are given in Listing B-3.

```
-> copy the server source code from http://www-128.ibm.com/developerworks/eserver/library/es-nweb.html
-> compile the code
./nweb-server port_number root_directory
```

Listing B-3. Installation steps for nweb.

Apache Tomcat

Apache Tomcat (formerly under the Apache Jakarta Project; Tomcat is now a top level project) is a web container developed at the Apache Software Foundation.

Installation steps are given in Listing B-4.

```
-> download the sources from http://tomcat.apache.org/
-> enter the bin directory and execute the command line: ./catalina.sh start
```

Listing B-4. Installation steps for Apache Tomcat.

Jetty

Jetty is an open-source, standards-based, full-featured web server implemented entirely in java. It is released under the Apache 2.0 licence and is therefore free for commercial use and distribution.

Installation steps are given in Listing B-5.

```
wget http://surfnet.dl.sourceforge.net/sourceforge/jetty/jetty-6.0.0rc0.zip
unzip jetty-6.0.0rc0.zip
cd jetty-6.0.0rc0
java -jar start.jar etc/jetty.xml
```

Listing B-5. Installation steps for Jetty.