Petr Novák

Network Repository for Performance Evaluation Results
Department of Software Engineering

Supervisor: Doc. Ing. Petr Tůma, Dr.
Computer Science Program, Software Systems
I would like to thank my supervisor, Petr Tůma, for his encouraging guidance, support and patience.

I would also like to thank my friends Kristina and Luboš for proofreading the thesis.

I declare that I have written this master thesis myself, using only the referenced sources. I give my consent to lending the thesis.

Prague, April 17, 2008

Petr Novák
Table of Contents

1 Introduction ................................................................. 5
  1.1 Content ................................................................. 5

2 Mission (How Should the Repository Look) .................................. 6
  2.1 Use Cases ............................................................... 6
    2.1.1 "Home Brew Fiddling" .......................................... 6
    2.1.2 "Serious Experiment" ......................................... 7
    2.1.3 "Automated Framework" ...................................... 7

2.2 Requirements .................................................................. 7
  2.2.1 Functional Requirements ....................................... 8
  2.2.2 Non-functional Requirements .................................. 9

3 Related Work .................................................................. 9
  3.1 Benchmarks .............................................................. 9

4 The Solution ................................................................... 14
  4.1 Data Storage ............................................................ 15
  4.2 Data Model .............................................................. 16
    4.2.1 Parsing Individual Results Formats ....................... 16
    4.2.2 Metadata ......................................................... 19

4.3 Data Aggregation ......................................................... 19
  4.3.1 Executing Queries ................................................. 20

4.4 Plots .......................................................................... 22

4.5 Report Generation ....................................................... 22

4.6 Repository Users ........................................................ 23

4.7 External Interface ....................................................... 23

5 Implementation Details ................................................... 23
  5.1 Repository Core ....................................................... 24
    5.1.1 Data Plugins ..................................................... 25
    5.1.2 Query Processing ............................................. 26
    5.1.3 Plots ................................................................. 27
    5.1.4 Custom Views .................................................. 29
    5.1.5 Task Processor ................................................ 29
    5.1.6 XML Import/Export ........................................ 30

5.2 Web Interface ............................................................ 30

5.3 Web Service API ....................................................... 30

5.4 Security ................................................................. 31

6 Evaluation ................................................................. 31

7 Conclusion .................................................................. 33

8 Bibliography ............................................................... 34

Appendix A: User manual .................................................. 35

Appendix B: CD-ROM Contents .......................................... 50
Abstrakt

Název práce: Síťové úložiště pro výsledky výkonových testů
Autor: Petr Novák
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@mff.cuni.cz

Abstrakt: Výkonové testy softwarových systémů produkují velké množství dat, které je nutno ukládat a vyhodnocovat. Cílem diplomové práce je navrhnout a implementovat úložiště těchto výsledků, poskytující zároveň funkce pro jejich zpracování a prezentaci.

Součástí práce je implementace úložiště, zpracovaná jako webová aplikace a schopná ukládat a vyhodnocovat různé druhy výsledků včetně textových a XML formátů. Prezentační funkce zahrnují tvorbu grafů a generování webových stránek popisujících naměřené hodnoty.

Klíčová slova: webová aplikace, benchmark, úložiště výsledků, grafy, prezentace

Abstract

Title: Network Repository For Performance Evaluation Results
Author: Petr Novák
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@mff.cuni.cz

Abstract: Benchmarks of software systems produce large amount of data that need to be stored and processed. The goal of the thesis is to design and implement a repository, providing also functions for presentation of the results.

Repository implementation which is part of this thesis works as a web application and supports parsing of various result formats, including plaintext and XML. Presentation functions include generation of plots as well as HTML pages describing extracted values.

Keywords: web application, benchmark, result repository, plots, presentation
1 Introduction

Many contemporary software applications rely on middleware components such as CORBA or SOAP libraries. There are many projects targeted on evaluating performance of such middleware (an overview of the projects is available in [1]). However, these efforts are still mostly fragmented, especially when considering result presentation, as can be illustrated on the list of individual middleware benchmarking projects:

- RUBiS is a prototype of an auction site, used for performance benchmarking [2]. Benchmark results are available on the website. However, there is no way to submit new measurements. The pages appear to be created manually and no new results are being added.
- Xampler is a CORBA benchmarking suite [3], providing occasionally updated website with results. Still, the update requires manual intervention and other people can't submit their results.
- Sampler (a simplified CORBA benchmarking suite [4]) has a full result repository, including option for submitting new results, coded specifically for this benchmark.
- TAO Performance Scoreboard [5] monitors the performance of TAO CORBA implementation. Results from daily builds show up regularly using a framework coded specifically for this benchmark, no third-party submissions are possible.
- SPEC result database [6] – does not parse the result except for summary data, the rest of the information is in the original report generated by the benchmark application. Result submission is paid for.
- TPC (Transaction Processing Performance Council [7]) database collects performance results of entire server configurations. As with the SPEC database, result submission is paid.

It appears that with the exception of a few specifically designed repositories, there is no open platform for publishing (and eventually comparing or otherwise processing) the experiment results. This leads to extra work for benchmark authors, who have to create their own tools for their specific results.

The lack of a results repository is also pointed out by Brebner et al. [1], who also note that an open database would also allow extracting different information from one set of results for use in different research projects. Additionally, having a result database means a possibility to reference measured results, and a possibility of uploading new results can further confirm published conclusions.

Because of these reasons, we want a repository that is capable of storing a multitude of benchmarking data from experiments executed at various locations. The repository should also be capable of parsing and presenting common result formats. Still, on the parsing side we can assume that certain level of cooperation from benchmark authors can be obtained – that is, we do not need to parse absolutely everything, but we need to parse the output that the benchmark authors are easily able to produce.

1.1 Content

The following chapter describes detailed requirements on the results repository. In Chapter 3, existing related work is discussed. Chapter 4 contains explanation of problems presented by the requirements, and the approaches I chose to solve them, while Chapter 5 describes the
repository implementation in further detail. Chapter 6 analyses the implementation with respect to the requirements; the main text of the thesis is summarized in Chapter 7.

Appendix A contains User's Manual of the application, which is supplied on an enclosed CD-ROM, together with off-line browsable snapshot of the web interface. The detailed content of the CD is listed in Appendix B.

## 2 Mission (How Should the Repository Look)

The purpose of this work is to design and implement a repository that would allow to store and present various benchmark results. The repository has to be accessible through web interface, parts of functionality that are important to third-party applications will be published through web service API.

In scope of this work, test means one type of benchmark with one format of results and test instance is set of data produced by a third-party application that is uploaded into the repository (i.e. an actual benchmark result).

### 2.1 Use Cases

This section contains use cases explaining typical uses of the repository, which were provided by the thesis supervisor in the initial stages of design. I chose to include them in this work, because I think they provide an overview of the application purpose in a slightly more accessible way than the requirements I've drawn from them.

#### 2.1.1 "Home Brew Fiddling"

I want to test a few things with a benchmark I have just devised. The benchmark produces a trivial textual output that I have just thought nice, sometimes intermingled with debugging messages, sometimes not. The output is in two files for each experiment. Each experiment also has a plain English description of whatever configuration it was run under.

After I've described the format of the results to the repository, I take the results and upload them to the repository using a simple web browser form. After I do this, I can ask the repository to plot the results in various graphs, interactively via a web form again. I can plot results from multiple experiments against each other, and filter experiments by regular expressions matched against the configuration description.

I might also want to extract the original files from the repository in a bundle, just in case I happen to lose the originals.

**Functional Requirements**

- Parse results from text files
- Store results that span across multiple files
- Upload results via a web-browser form
- Define plots via web forms
- Compare results from multiple experiments
- Filter experiments (by matching regular expressions on supplied metadata)
2.1.2 "Serious Experiment"

I get a larger benchmark experiment running. The results are again text files, but this time, I have no control over the content for I use somebody else's benchmark. The files are rather large, couple of megabytes, and stored across a few directories.

I want to upload the results in an archive rather than file by file, because that's easier. I want to use a description of the result format that came with the benchmark to make it interoperable with my repository, and just upload that rather than to fill in forms. The description would also contain some useful graphs to create based on the data, but I can still devise additional ways to plot the data using the web form.

I can provide pointers to the data to other people. The public would not be able to upload data under the same experiment header, but they would be able to browse my results pretty much in the same way as I do. I can also define a summary page that contains brief information of my choice, and an arbitrary number of detail pages that I can use to point out something in the data and send a link to the page to other people.

Functional Requirements

- upload multi-file results in one archive
- import test type descriptions (together with plot definitions)
- create public browsable presentation, define arbitrary number of pages describing the results

2.1.3 "Automated Framework"

Pretty much the same as the "Serious Experiment" use case, except I want to whip up a few tools that would process the data. The tools would be in multiple programming languages and would use interfaces typical for those languages, such as RMI or CORBA, to access the repository results.

The tools would be able to extend the results, for example by calculating some additional statistics, which would be stored back in the repository, so that the tool can reuse them if needed.

Functional Requirements

- Interface for third-party applications to access the data
- Space for additional data uploaded to the results by third-party applications

2.2 Requirements

What follows is a more detailed summary of the requirements provided by the expected use cases.


2.2.1 Functional Requirements

**FR1 Result Formats**
The repository has to be able to parse multiple result formats, including plain-text files (with syntax similar to CSV or formats with “measurement type = result”).

In addition, XML-based result formats have to be supported. The resulting repository has to be able to parse results from Sampler, Xampler, RUBiS, ECperf, TAO and OVM benchmarks (per thesis assignment).

The application has to accept test results that consist of multiple files. This requirement is also forced by RUBiS, ECperf, and OVM benchmarks result format.

The set of supported result formats should not be closed, but instead remain flexible enough to accommodate potentially appearing new formats.

**FR2 Configuration Information**
The application has to provide means to collect and store metadata together with uploaded results. These data can either be extracted from the results (configuration information) or entered by the uploader (contact information or the configuration, in case it isn't provided in the results).

**FR3 Plots**
The repository has to provide customizable support for multiple display formats including box and whisker plots, density plots and history plots.

At least Sampler, Xampler, TAO and OVM display formats need to be supported.

**FR4 Web Interface**
Repository functions, including results upload and definition of plots need to be accessible from the web interface.

When uploading multi-file results, an option for uploading all the files in one archive (for example tar) has to be supported.

**FR5 Presentation Functions**
The web interface needs to be able to provide a browsable public presentation of results, including customizable summary pages and an arbitrary number of detail pages, further describing the results.

The presentation has to be able to provide comparisons of results from multiple experiments. User has to be able to search in the results, at least by filtering the experiments by regular expressions (matched on metadata belonging to the experiments).

**FR6 Availability of Original Data**
The repository has to provide options for downloading the original result data.
**FR7  Import/Export of Test Descriptions**

In addition to defining the test format and presentation (including public pages and plots) by web forms, the repository has to provide means to upload this description from an external file.

Also, exporting an already defined test description into the same format should be supported.

**FR8  Interface for Other Applications**

The repository needs to provide interface for other applications to access the data. In addition to uploading new results, the third-party applications have to be able to store additional (computed) data to already existing results and retrieve it afterwards.

### 2.2.2 Non-functional Requirements

**NFR1  Platform**

The product will run on Linux operating system, and will not use relational database for critical data (per assignment).

**NFR2  Performance**

Application's web interface has to be reasonably responsive (within seconds at most). Result parsing may be done in the background, but should not take more than a few minutes and the rest of the interface still needs to be usable.

**NFR3  Documentation**

The product will be accompanied with both administrator and user documentation, describing installation of the repository application and instructions for how to set up a repository for benchmark results.

### 3  Related Work

Apart from the commercial databases mentioned in Chapter 1, there already exists an open result repository implementation in Java [8], providing a web service API. Unfortunately, when familiarizing myself with said implementation, I have encountered issues with Enterprise Java portability which have made the repository nontrivial to install.

Also, I believe that relying on a dynamic scripting language such as Python opens new options to implementing some of the repository features which are difficult to achieve in Java, which is why I have decided to abandon the original implementation entirely. This has the obvious drawback of potentially adding more work, but, as I outline in the feature comparison in the concluding section, the result is at least as functional as the original implementation.

### 3.1 Benchmarks

Due to the nature of the work, which involves parsing benchmark results, middleware benchmark applications should also be considered related to the repository. Although the following list is based on the thesis assignment and is thus not exhaustive, I believe that it covers most of the result formats that the product will be required to handle.
Sampler

Sampler [4] produces one XML file with description of the configuration, followed by sets of measurement results.

```xml
<?xml version="1.0"?>
<Results VersionMajor="1" VersionMinor="1">
  <Configuration Uniquifier="709897192072008178914961242247341618957125097562388359439506064
  807131098618603690772794765543405366172986181773155971112610527409972767518
  272122458149751638"></Configuration>
  <Client VersionMajor="1" VersionMinor="13">
    BenchmarkClient 1.13 [--/--/--]
  </Client>
  <Identity>
    <Processor>
      <Vendor>Unknown</Vendor> <Family>Sparc</Family> <Model>Sparcv9</Model>
      <Clock>360</Clock>
    </Processor>
    <Memory PhysicalFree="86990848" PhysicalTotal="536870912"
    VirtualFree="471760896" VirtualTotal="957898752"></Memory>
    <Timer Scale="1000000000" Granularity="444"></Timer>
    <System>
      <Vendor>Sun</Vendor> <Family>Unix</Family> <Model>Solaris</Model>
      <VersionMajor>5</VersionMajor> <VersionMinor>7</VersionMinor> SunOS 5.7 Generic_106541-20</System>
    <Compiler>
      <Vendor>Opensource</Vendor> <Family>C++</Family> <Model>GNU C++</Model>
      <Optimized>No</Optimized> GNU C++ 2.95.3 20010315
    </Compiler>
    (snipped)
  </Client>
  <Session>
    <Benchmark Type="Instances Parallel Sequence In">
      <Measurement Size="0" Count="1" Simul="1">
        <Sample LoadBefore="0" LoadAfter="2">481790 481790 481790</Sample>
        <Sample LoadBefore="0" LoadAfter="1">519197 519197 519197</Sample>
        <Sample LoadBefore="0" LoadAfter="1">485754 485754 485754</Sample>
        <Sample LoadBefore="0" LoadAfter="3">505078 505078 505078</Sample>
        <Sample LoadBefore="0" LoadAfter="2">513754 513754 513754</Sample>
      </Measurement>
    </Benchmark>
    (snipped)
  </Session>
</Results>
```

Illustration 1: Example of Sampler output
Xampler

Xampler [3] outputs set of text files, where one files contains result of one measurement. The files contain a series of annotated values, which can be either a string or a set of integers.

```
[BENCHMARK] [STR] Version Xampler post 1.10 post 04/02/02 [STR]
Suite Invocation Static Client
[CONFIGURATION] [1] Scale 2193560000 [1] Granularity 1
[MEASUREMENT] [1] Samples 1 [1] Threads 1
[SAMPLE] [3] MemoryApplicationResident 5025792 5025792
5025792 [3] MemoryApplicationSwapped 13873152 13873152
13873152 [3] MemoryKernelUsed 96382976 96382976
96391168 [3] MemoryTotalPhysicalUsed 311914496 312003379
312041472 [3] MemoryTotalPhysicalFree 215420928 215459020
215547904 [3] MemoryTotalSwapUsed 0 0 0 [3]
NetworkBytesSent 8587000 865216 873986 [3]
NetworkBytesReceived 858759 865246 873986 [3] NetworkPacketsSent 8256 8318 8403 [3] NetworkPacketsReceived 8257 8319
8403 [3] ProcessorApplicationThreads 2 2 2 [3]
299844 1052740 [5000] HalfwaySolo 318632 303884 294260
295008 296624 295792 301708 296648 294184 296712 291232
296152 (...)
```

Illustration 2: Example of Xampler output

RUBiS

RUBiS [2] benchmarks produce human-readable HTML, including plots. The role of the repository in this case is limited to parsing summary information and generating overview pages, linking to already-generated HTML.

```
Illustration 3: Part of RUBiS Performance Report
```

Illustration 3: Part of RUBiS Performance Report
TAO

TAO (The ACE ORB) [9] is an open-source CORBA implementation. Its included benchmarks do not save any files by default, but produce plain text output with measured values, which can be piped into files and then possibly parsed.

```
-./ACE/TAO/performance-tests/Latency/Single_Threaded $ ./run_test.pl
================ Single-threaded Latency Test
server (18251|3081414352): user is not superuser, test runs in time-shared class
client (18252|3082004176): user is not superuser, test runs in time-shared class
test finished
High resolution timer calibration....done
Total latency :  91[45765]/108/34042[200634] (min/avg/max)
Total throughput:  9247.53 (events/second)
(18251|3081414352) server - event loop finished
```

*Illustration 4: Example of TAO performance test output*

ECperf


ECperf produces several text files, including direct output of executed commands and summaries of the performance figures.

```
ECPerf Summary Report
Version : ECperf 1.1 Final Release

Run Parameters :
runOrderEntry = 1
runMfg = 1
txRate = 5
rampUp (in seconds) = 480
rampDown (in seconds) = 180
stdyState (in seconds) = 600
triggerTime (in seconds) = 30
numOrdersAgents = 1, numMfgAgents = 1
dumpStats = 0
Benchmark Started At : Thu Nov 24 01:52:32 GMT 2005

Orders Summary report is in : Orders.summary
Orders Detailed report is in : Orders.detail
Orders Transaction Rate : 280.60 Transactions/min

Manufacturing Summary report is in : Mfg.summary
Manufacturing Detail report is in : Mfg.detail
Manufacturing Rate : 158.40 WorkOrders/min

ECperf Metric : 439.00 BBops/min
```

*Illustration 5: Example of ECperf summary*
**ECPERF Detailed Report**

**Manufacturing Throughput**

<table>
<thead>
<tr>
<th>TIME</th>
<th>COUNT OF TX.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>35</td>
</tr>
<tr>
<td>90</td>
<td>30</td>
</tr>
<tr>
<td>120</td>
<td>40</td>
</tr>
<tr>
<td>150</td>
<td>67</td>
</tr>
<tr>
<td>180</td>
<td>67</td>
</tr>
<tr>
<td>210</td>
<td>80</td>
</tr>
<tr>
<td>240</td>
<td>67</td>
</tr>
</tbody>
</table>

(...)

*Illustration 6: Example of ECperf detailed report*

**OVM**

Ovm Project was an open source framework for building programming language runtime systems [11]. The project web site is no longer available, but according to [12], its benchmarks focused on measuring latency.

Samples of OVM results, provided by the thesis supervisor, contain whole directory structure, where one directory contains index files with descriptions of test environments and links to files in other directories that contain the measured data.

```plaintext
name=testEvent
noiseThreads=0
heap=false
nhrt=false
warmup=10
bound=false
help=false
noheap=false
TAG=REGRESSION
filter=rtjunit.examples.latency.EventFilter
N=1100
display=false
plotter=rtjunit.output.plotters.TermPlotter
noiseClass=rtjunit.examples.SimpleNoiseMaker
JVM=OVM
noisePriority=12
priority=36
fileName=1097650220009
```

---------------------

```plaintext
name=testEvent
noiseThreads=0
heap=false
(...)
```

*Illustration 7: Example of OVM index file*
4 The Solution

In the design of the repository, there are a few non-trivial questions to solve. The main problem is how to extract data for plots and summary pages in a way that is both generic enough to cover a wide range of possible inputs and usable in the real world. Additional problems include choosing the structure to store the result data in the first place, and formatting the data for presentation afterwards.

The collected requirements suggest architecture of a repository with a central data model, filled by an extensible input parser based on plugins and queried by a similarly extensible output generator. The architecture is outlined on Diagram 1, further chapters deal with more detailed function of the individual parts of the repository.

Diagram 1: Outline of core repository components

This chapter deals with the repository design in “chronological” order, starting with storing of the results and then continuing through data processing and generating plots towards final presentation of the results.
4.1 Data Storage

The main entities that the repository should know about are test types, with description of test data and ways to present them, as well as uploaded test instances that contain result files and aggregated values computed from them.

All these entities can be arranged into a hierarchical structure:

- test type contains
  - test description and
  - instances, which contain
    - uploaded data and
    - aggregated values or other metadata

In the assumed workload of the repository, queries will outweigh updates by several orders of magnitude. Therefore, the storage needs to be optimized for reading rather than for fast writes. Also, the storage should be robust enough so that in case of failure, even an administrator who isn't familiar with the details of the data structure is able to extract the stored data.

Finally, the structure of stored data is variable, as we don't know in advance what kind of result data will be used and what types of metadata should be stored alongside them.

Options

The two options I considered was a relational database and directory structure in a regular file system.

The obvious advantage of relational database is the search speed, its drawbacks are nontrivial extraction of data in case of failure and complicated storing of data with unknown structure – either the database structure would have to be adjusted dynamically to accommodate new result formats, or the result data would have to be stored in BLOB columns. Similarly for metadata, we'd have to either dynamically alter the table structures to follow the test definitions, or employ a universal schema, which would mean losing the advantage of effective searching.

On the other hand, file systems are naturally designed to handle hierarchies of large files of unknown content, but their access times are much slower compared to relational databases, especially since they offer no index capabilities except for indexed directory entries. Search for a test instance with a certain attribute value would mean going through attributes of all existing instances.

Decision

My choice for data storage was directory structure in a regular filesystem, because of the easier recovery of result data from regular files in case of failure. I opted for including metadata with the result data for sake of design simplicity.

Evaluation

In the later phases of development, I have found out that listing of larger number of test instances (several hundred) can be slow due to the time spent traversing the individual directories. I've therefore added support for caching of the list of instances and their metadata, keeping the duration of listing (and searching in) instances below 2 seconds (as opposed to
approximately 10 seconds on the same platform when reading the metadata directly from individual files).

The other significant operation of the repository, which is generation and loading of a detail page, was tested on a page that only uses metadata from one instance and contains three plots. This operation took between 1 and 2 seconds. (All values were measured on a mildly loaded 1.8 GHz Xeon server.)

It can be expected that processing times of requests that list data of more test instances will grow linearly with the number of instances contained in the test. Number of metadata items has impact as well, because of the increased size of records that need to be fetched.

4.2 Data Model

As was mentioned earlier, there are many possible formats of benchmark results. However, data contained in them can always be structured as a tree (see Chapter 4.2.1 below). Conversion to a tree is trivial for XML data, and can also work on results consisting of multiple files – output tree is identical to file/directory structure in the upper levels, and individual files turn into nodes that have parsed file contents under them.

![Illustration 9: Transformation of result files into data tree](image)

Transforming individual files into their data trees depends on the file format. Description of a test type would then consist of a list of files that make up its result, together with specification of plugin which should be used to parse the file. Using pluggable modules to parse result data also makes adding support for new data formats relatively easy.

It can be argued that representing results as a tree increases memory requirements. For example, an array of thousand integers takes less space than tree with the same data, because of the overhead of the data structure. Although this is a valid point, the repository implementation does not keep the tree in memory and rather processes it nodes sequentially, as described in Chapter 4.3.1. Therefore, size of the result representation is not an issue.

4.2.1 Parsing Individual Results Formats

This section explains how results of benchmarks described in Chapter 3.1 can be transformed into a tree structure.
Sampler

Sampler produces XML data, whose element structure already is a tree. The attributes and text content of the elements form leaves of the output.

The only problematic part are the measured values, which occur as sequences of whitespace-separated integers in the text content of “Sample” elements. This is solved by splitting content of such elements by said whitespace and outputting one leaf per every integer value found.

Xampler

Xampler files consist of several sections, introduced by upper-case heading enclosed in square brackets, and followed by several set of values, consisting of type specification (either “[STR]” for string, or “[n]” for set of n integers), value identifier, and the value itself.

The resulting tree contains one branch for every section, under which there are either leaves representing singular values or other branches which contain integer sets.

Illustration 10: Example of Xampler output and its corresponding tree representation

RUBiS

The role of repository in case of this test is limited to generation of summary pages from information extracted from the RUBiS HTML output. This can be done by matching regular expressions on the file content and generating tree nodes on individual matches and their groups.

For example, when regular expression `(?:<h3>([^<]+)</h3>)|(?:Average throughput</div><TD colspan=6 align=center><B><[^-]<[^>]+>\(([0-9]+) req/s)` is matched on “Performance Report” report file from Illustration 3, the output is a tree in which for every throughput figure, the name of the statistics to which it is related can be found by looking on the previously matched heading. (Please note that RUBiS output is not optimized for easy data extraction.)

The regular expression above contains two parenthesized groups, first is inside “<h3>” tag, second is “([0-9]+)” near the end of expression. The content matched on these groups in the used to form leaf nodes with the data. The nodes also have serial numbers which can be used to distinguish which group the node represents. Another used syntax is `(?:...)` for non-matching groups. Python regular expressions also allow naming the groups, using syntax `(?: P<name>...).`
**TAO**

TAO benchmarks do not save any files by default. Had their output been saved into files, measured values could be extracted and organized in a way similar to parsing RUBiS data, only perhaps cleaner since there is no need to get around HTML tags.

**ECperf**

Again, text files produces by ECperf can be parsed using regular expressions. As with the previous cases, every match would generate a branch with leaves containing the matched groups.

For ECperf summary file from Illustration 5, the regular expression looks like this:

```regex
```

**OVM**

OVM benchmark outputs whole directory containing files whose names are not known in advance, but are defined in the index files of the same test instead. This poses a problem, because the repository design assumes that all files in the test are known and defined (for example to find out how to parse a particular file in the result). Changing this approach would make result parsing complicated, as some files would have to be parsed in advance to retrieve a complete list of files that make up the result, which is hard to describe in a generic way.

I have solved this problem by packing the whole directory structure into one tarball, which is uploaded and parsed as a single file with a special format. Code specific for this “OVM tarball” then unpacks and parses contained index files (which contain sets of “name=value” lines, always ending with fileName item) and also data files to which they link.

The resulting tree has one branch for every time of benchmark run (corresponding to one index file) with “info” branch containing configuration information from the index files and “data” branch with parsed contents of the data file (that is, series of integers, slightly similar in structure to those found in Xampler output).

**Conclusion**

With four types of parser plugins (XML, Xampler-specific, regular expression matching and OVM-specific), it's possible to parse all six result formats and transform data contained in them into a tree structure for further processing.

When considering support for additional result formats, I believe that most formats whose primary recipient is a human reader can be handled by the existing regular expression matching plugin. A result format targeted at a human reader would rarely contain large volumes of raw data (since the reader would not be able to process it anyway) and is likely to contain descriptive labels alongside the data (otherwise the reader would not be able to interpret the data). The regular expression matching plugin should be able to locate the data based on the descriptive labels.

The situation is different when considering formats whose primary recipient is a machine processor. Such formats can contain large volumes of raw data without descriptive labels and the ability of the repository to process such formats very much depends on their basic structure. With XML currently in the spotlight, chances are that the basic structure would rely on XML and the existing XML plugin would be able to handle the results. If the basic structure were to rely on a different format, such as CSV or NetCDF, a new plugin would...
have to be coded. Such a plugin, however, would be similarly generic as the XML plugin, and therefore would be potentially reusable again.

### 4.2.2 Metadata

In addition to uploaded result data, the repository also stores information about the uploader (and the level of result trustworthiness, which is one of “submitted by registered user”, “submitted with valid e-mail address” and “submitted anonymously”; see Chapter 4.6 for details). Apart from these data it’s possible to define an arbitrary number of so-called *instance attributes*, either to be entered during the upload, or extracted by queries similar to the ones which gather data for plots (this process is explained in more detail in the following chapter).

Metadata extraction can be used both for providing aggregated values (such as “maximal CPU load occurring around measurements”) and descriptions to be used in the result presentation.

### 4.3 Data Aggregation

To create useful plots, data need to be supplied in a format similar to a table, or key:value pairs. For example, a graph showing dependence of middleware performance on the number of clients can be plotted from a set of \(<\text{number of clients } n>, \text{<performance with } n \text{ clients}>\) pairs.

A successful repository implementation has to provide a way of aggregating results data into output suitable for plotting and presentation. At this point of processing, output from all type of benchmarks is fairly similar and the aggregation process only has to pick the right leaf values (according to information available in neighboring branches) and summarize their content.

For example of Xampler output on Illustration 11, for getting table of average “ThruputSolo” values grouped by number of threads, we just need to average all values in “ThruputSolo” branches and match them to “Threads” value in the preceding “MEASUREMENT” section.

One way to do this is to store the tree as XML data and use an XQuery processor to produce the aggregated output. This approach can benefit from using an existing technology, but requires the processor to parse whole XML document into memory, which can be time-consuming and also limiting potential size of the result data.

Another possible solution is to read data and aggregate values sequentially, keeping only those that are necessary for computation of aggregated results. Even though this approach means necessity to design another way of describing queries, I considered avoiding the necessity to keep whole tree of result data in memory a useful possible benefit, which is why I chose to explore this option further.

---

**Illustration 11**: Parsed Xampler output
4.3.1 Executing Queries

One of important tasks of the repository is to transform result data (which can be generally looked up as a tree) into a form suitable for generating plots (two-column table). This chapter focuses on the possibility of doing so by sequentially processing the tree nodes, as they are produced by the parser.

Assuming that for aggregating the data, random access to the tree is not vital, I'm going to try to specify a query (or rather, instructions for data filtering) that would allow to extract useful data for plots during sequential traversal of the parsed tree.

First of all, let's take a typical tree output we get from a Sampler benchmark:

First of all, let's take a typical tree output we get from a Sampler benchmark:

Illustration 12: Part of Sampler result data tree and aggregated output suitable for a plot

For example shown on Illustration 12, data extraction can be done in a following process:

1. Take all leaves that contain measured values (those can be found at the end of the path “/Results/Session/Benchmark/Measurement/Sample/<something>”, empty name)
2. For every such node, take pair [<size of corresponding measurement>, <node value>] (such as [400, 185610])
3. Group all pairs with the same measurement together, and produce set of pairs [X, <average of all pairs with measurement size X>]

Or, more generally:

1. Filter leaves
2. Produce [key, value] pair from the leaves
3. Group pairs with the same key and aggregate their values

This can be done sequentially without random access to the whole result tree, with the exception of the second step, where information from one node is rarely to get anything useful. On the other hand, I believe that for most applications, it's enough to know some of the
preceding values. In the above example, it's sufficient to know the last value of leaf with path “/Results/Session/Benchmark/Measurement/Size”.

Therefore, sequential processing of the data is still possible if the routine remembers some information on its way through the tree, presuming that these “additional information” (usually describing type of the measurement and conditions under which the data were obtained) are placed before the actual data. Since the assumption holds for all result formats in the specification, I've considered limitations of this approach as acceptable.

In the repository implementation, a query consists of three expressions:

1. “where”, used to filter the leaves
2. “group by”, used to compute grouping keys
3. “value”, returning the value to be aggregated

Additionally, query specifies data aggregation function, and definition of “bonus” paths, whose values will be stored during the tree traversal and made available to the above expressions.

**Output of Other Benchmarks**

The process described for Sampler trees works equally well on Xampler output mentioned above, as well as OVM.

For files parsed by regular expressions, such as RUBiS output on Illustration 13, we just need to select nodes which correspond to the second group of the regular expression, and assign them to nodes that are result of first matched group. As all the nodes are numbered, this can be done by looking on the leaf numbers, which correspond to the groups.

A query extracting the overall statistics from RUBiS would look like this:

- as the “bonus”, save values of paths `/None/s=1` (nameless branch, and leaf with serial number 1 – i.e. first regexp group)
- take all nodes with path `/None/s=2` (second regexp group)
- as “value”, take the value of selected nodes
- group them under the saved bonus value, and aggregate them using aggregation function “Last” (which simply returns last entered value)

ECperf and eventual TAO output are parsed by the same regular expression plugin and aggregation would be done in a similar way.

**Conclusion**

Data aggregation using sequential access to the parsed tree is able to extract information from all analyzed result formats. I've chosen to employ this approach for the repository not only because it does not have to maintain parsed results in memory, but also because it's able to avoid writing the intermediate data anywhere at all.
Therefore, all we have to do extract data from benchmark result is to pipe parser output to the
data aggregator and then pick up the result.

This also means that amount of data that can be contained in one test instance is not limited
by main memory, as long as the data necessary for computing the result fit in.

### 4.4 Plots

An output of a regular query is a set of [key, aggregated value] pairs, which is then used for
plotting.

The plots that the repository has supports can be divided into three groups:

- **Regular**, where keys come on X axis and aggregated values on Y axis. This includes
  most of regular plots including box and whisker plots, density plots and history plots
  (provided that the tests data contain time information to be user for X coordinates).
- **Bar charts**, which differ in that the keys are not numeric coordinates but rather names
  of groups whose values are displayed by the bars.
- **Histograms**, where optional filtering of the values is employed, to get more
  meaningful output. If the histogram is drawn for whole interval of the values, it's
  prone to being thrown completely off by one extreme measurement. Therefore, the
  plotting functions are able to remove extremes from the intervals.

With the exception of histograms, the plots consist of several “plot elements”. Every plot
element takes data from one data query, and displays them in the plot in the specified form
and color. That makes it possible to draw for example both minimal and average value of one
experiment into the same plot.

Repository implementation uses Biggles package to produce actual graphical plots.

### 4.5 Report Generation

The presentation functions are another part of the network repository. They include creation
of multiple overview and detail pages for each test, and ability to search within the results.

It can be assumed that the pages would:

- present information contained in test instance attributes (which can be almost
  anything)
- display data plots
- link to each other

Because it's difficult to foresee what kind of presentation structure would be appropriate for
any type of test, the presentation pages should be as flexible as possible.

It is worth noting that there is already a whole area of software specialized on formatting
content for presentation, especially on the World Wide Web; that is, web template systems. In
context of the repository, the templates are going to be supplied by test authors, and then put
together with data obtained from the results to form the presentation.

In the repository, user-supplied templates are called *custom views*. There are three types of
custom views, differentiated by what kind of data forms their input:

- **per-test**, which display data that are global for the whole test (such as instructions for
  instance upload or a list of instances)
Instances to be shown in a multi-instance view can be either explicitly named (by their id's) or filtered by a user-defined expression, combining instance metadata and user input. This way, it's possible for example to search in the results by matching input from web form with specified instance attributes.

4.6 Repository Users

User management in the repository has two goals:

- Prevent unauthorized users from changing the data
- Establish result trust based on their source

Because of the first task, some kind of user authentication has to be maintained, so the repository implementation will have some structure of user accounts with rights to edit certain data.

The structure I chose is simple: some users are marked as repository administrators, and can edit descriptions of all tests, delete instances and so on. Other users can do this only in test that they have created. Since it is not presumed that any data in the repository are confidential, every user can read all data contained in the test.

Regarding result trust, there are three possible classes of results:

1. result uploaded by registered users
2. results whose author has entered a valid e-mail address
3. completely anonymous results

All results uploaded by unregistered users are marked as anonymous at first. If the uploader enters an e-mail address, a confirmation code is sent to that address. If the uploader later enters the code (by visiting hyperlink contained in the e-mail), the result is marked as having a valid e-mail.

Status of the result is included in its metadata, making it possible to filter the results by their trustfulness.

4.7 External Interface

Supposed usage of repository by other applications include automatic upload of results after obtaining them, additional data processing and possibly alternative means of display. Therefore, other applications have to be able to retrieve list of tests and their descriptions, including files they consist of, and defined instance attributes, plots and custom views. Additionally, means to upload new instances have to be provided, as a well as options for reading and writing custom data.

5 Implementation Details

Environment chosen for the implementation was Python, coupled with Django framework, which was used for web interface, as well as generation of custom views. Other used packages are elementtree (XML parsing), Biggles (plotting) and ZSI (web service).
The basic parts of the implementation are:

- repository core, providing access to the on-disk objects and containing all the application logic for parsing data and evaluating queries, plotting, etc
- web interface, a Django application providing access to all functions
- web service API, providing access to some of the repository functions

The following chapters also serve as an introductory part of programmer's documentation. Please refer to directory `doc/pydoc/` on the enclosed CD-ROM for detailed reference.

## 5.1 Repository Core

Repository core is built around `GeneralEntity` objects, which are defined in `backend.py` file and provide means to persistently store their attributes in the file system (usually as Python dictionaries, serialized using the pickle module). Each `GeneralEntity` object lives in a filesystem directory and can contain third-party data in addition to its own properties.

In `repository.py`, `GeneralEntities` are subclassed into `Repository`, `User`, `Test` and `TestInstance` classes, which represent the main entities of the repository. The file also contains definition of `DataPlugin` class, which defines interface of parsing plugins, responsible for transforming various data files into results trees that can be further processed. The main part of their interface is “applyFunction” method, which parses a given data file and calls specified callback function on all leaf nodes it finds within it.
TestInstance classes also provide applyFunction method, which calls the method on individual files (using the parser plugins assigned to them) contained in the instance and alters the node paths so that the resulting tree contains branches for all files in the test.

### 5.1.1 Data Plugins

Data plugins handle conversion of result files to their tree representations (as shown on Illustration 10). The plugin is a Python class that subclasses repository.DataPlugin class as implements three methods:

- **getParamDesc()** – static method that returns dictionary of fields for entering parameters that may alter behavior of the plugin. The field classes are defined in file *fieldtypes.py*.

- **__init__(self, params)** – constructor; the “params” argument is a dictionary of parameter values defined by the getParamDesc function

- **applyFunction(self, dataFile, f)** – parses given file and calls function “f” on every leaf it encounters.

The “f” function gets list of PathElement objects as its only argument. PathElement objects are defined in file *pathelements.py* and have two attributes: name and a serial number. Both of these values can be used for identifying the nodes during aggregation.

- Name can be any string, Python `None` constant is usually filled in for nodes which have no name associated with them

- Serial number is an integer. Children of a node have to be numbered in increasing sequence, usually beginning with zero. (For example, regular expression plugin uses group index in a match as serial number even though it does not always output nodes for unmatched groups.)

Every path ends with a Value object that also has a name and a serial number. In addition to those, it also has a “value” attribute which contains the actual value, which can be any Python basic type such as integer, long integer, float or a string. The Value class constructor automatically converts string which contain integer or float value to the right types.

Assigning correct serial numbers to the nodes is critical, because many tests contain several nodes of the same name and serial number is the only way to identify them. It's also important to remember that the “f” function calls represent paths to leaves of the tree.

![Diagram 3: Logical relations between main repository objects](image-url)
Subdirectory `plugins/` contains a few of the parsing plugins, including an example in file `dummy.py`.

To add a new plugin to the repository, copy its source code to plugins directory and add a new record to `DATA_PLUGINS` tuple in file `skladr/config.py`. The record is a tuple:

`('sourcefilename.ClassName', 'human readable name')`

### 5.1.2 Query Processing

Query processing is done in aggregator objects, which provide methods for entering individual nodes of the tree, and compute various results from them.

Aggregators are defined in file `agreg.py` and include:

- **SimpleAggregator**, which processes data queries, as described in Chapter 4.3.1 (Executing queries)

- **MultiAggregator**, which groups more **SimpleAggregator**s to one object, and thus allows processing multiple queries per parser run.

- **ListChildren**, which outputs children of selected node in the tree and is used in the web interface for previewing parser output.

---

**Diagram 4: Classes that take part in query processing**

The data extraction process consists of initializing an aggregator and calling test instance `applyFunction` method with aggregator's data entry function. **TestInstance** objects
creates appropriate parsing plugins and runs them on result files, sending tree nodes to the aggregator, which processes them as they come. After the parsing is finished, the resulting data are fetched from the aggregator. This process is displayed in more detail on Diagram 5 below.

Two other modules are related to query processing: *query.py* contains classes that represent queries and provide interfaces for their execution, and *aggfunc.py* defines individual aggregation functions (such as `Max` or `Avg`) that are used for grouping values within *SimpleAggregator*.

![Diagram 5: Execution of a query](image)

### 5.1.3 Plots

Plotting is implemented in module *plots.py*, which contains **TestPlots** class, responsible for managing plots assigned to test type, as well as their rendering through Biggles package. Plot records are represented by **Plot** class, which contains all “elements” of the plot. For regular plots, plot elements are various values that should be represented in the plot, for example by a line or group of points. Every element has assigned display type and color as well as query that is used to fetch its values.
Diagram 6: Classes related to plot management and rendering

For bar plots, the elements correspond to values that should be displayed for every group.

Illustration 16: Regular plot with two visible elements (line and dots), both displaying the same query. The groups from the output are put on X axis, resulting values on Y axis.
Third plot type is histogram. Histogram has only one query and no elements, and is able to alter groups that come from the input for optimal display, set either by constant number of groups, or by setting a percentage of value that the biggest group has to contain. Cutting of extreme values from the measurement is supported as well.

5.1.4 Custom Views

For data presentation, I've chosen to employ Django template system, because it is flexible and secure.

It supports simple loop and branching, as well as resolving content by calling object methods. On the other hand, the template code has only access to objects that it has assigned, and it is possible to declare destructive methods as forbidden for execution from inside a template.

For example, if a template gets an object representing test in a repository, it can retrieve all its instances and output a table of their attributes, but cannot call “delete” method of the test object. The templating system isn't limited on HTML and can be used to produce any text-based format.

Custom view-related functions are contained in module `customviews.py`, which defines CustomView class representing a custom view and provides means to render it via Django templating system.

5.1.5 Task Processor

Since evaluating queries can be time consuming, the repository has a simple routine to run these tests on background, without delaying operations in user interface. Task processor uses relational database table (through Django database access layer) to maintain queue of information that need to be refreshed.

Items are added into the queue by user-interface code when update is necessary (on plot definition change, upload of new instance, etc.) and then processed by a simple application...
that can be run on background. This process also clean up old instances created for upload, but not finished.

Task processor is contained in file `task_dispatcher.py`, which fetches the tasks from the database, and `task_processor.py`, which handles the processing. The process itself is launched from Django management utility that sets the database and other environment.

5.1.6 XML Import/Export

Functions for importing and exporting test definitions into XML format are defined in modules `xmlstuff.py` and use Python `elementtree` package to handle parsing and output of XML.

5.2 Web Interface

Web interface is built on top of Django, which is a Model-View-Controller framework (although it uses a slightly different terminology). I chose to use Django development version instead of last current stable (0.96.1), because some parts of API has changed since the stable release, and I believe having the application compatible with upcoming releases is preferable. The last version of Django the application was tested with was trunk revision 7420.

Core repository objects provide data instead of database layer. Django itself uses a set of regular expressions to define URL patterns (file `skladr/urls.py`), and dispatches HTTP requests to “view” functions (in directory `skladr/views`) that fetch repository objects, processes changes coming through POST and passes the objects to templates which format the final HTML output. Detailed explanation of Django framework is available at [13].

Web interface uses jQuery library for handling Ajax and extjs TreeView control to display preview of parsed result trees.

5.3 Web Service API

Web service API provides access to some of the repository functions, including:

- getting list of tests
- getting description of test, including available plots, instance attributes, and custom views
- getting list of test instances
- getting URL of a rendered plot
- test instance upload
- rendering custom views
- access to custom application data, stored either per-test or per-instance

For operations that alter data (instance upload, changes of custom data), the client needs to have a valid API session, which is generated from digest of a API key, that can be assigned to the user account in the web interface. Precise specification of the function calls is defined in file `api/Sklad.wsdl`.

Access control of third-party files is trivial: everyone can read all the data and all users with valid API key can replace any data. Therefore, only trusted users should get write access to the API.
API server code is located in file `api/server.py`; there is also a simple command-line application for uploading new instances in file `api/upload.py`.

Web service code uses version 2.0 of ZSI library. The service has its own server, which is launched using Django command-line management utility.

### 5.4 Security

Presented repository design relies on the dynamic nature of the environment, enabling customization by evaluating user-entered Python expressions. Because this is also a potential security threat, it is necessary to analyze potential impact of malformed inputs to the system.

The inputs to the repository are the following:

- Uploaded result data – Files are saved in designated data directories. Data-parsing plugins output no executable code, only series of node paths.
- Test descriptions are entirely supplied by the user and contain:
  - paths of files that are uploaded in the tests – these need to be checked for attempts to access data outside of the directories; paths are checked for occurrence of string “../” in the functions that handle adding/reading of files in TestInstance class
  - expressions in queries and instance filters are executed by Python `eval` function, which allow to limit global and local symbol available to the code; the expressions are therefore evaluated in environment where only access to a few safe functions is available
- User-supplied custom views are rendered by Django template system, which allows the template code to access properties of the objects representing the repository entities, including execution of some of their methods. For security reasons, the system refuses to resolve attributes or methods whose names begin with underscore, or methods marked with “alters_data” attribute. The repository implementation uses these mechanisms to hide sensitive data (such as user auth information and absolute directory paths) and protect destructive methods.
- User data files, accessible through web service API, are also present directly in the file system, so their paths are checked for attempts to access different directory.
- Parsing errors of imported XML test definitions are checked. Testing of semantic validity of supplied data (such as file paths and callback expressions) is not necessary, because this is handled later during the processing.

### 6 Evaluation

Presented repository implementation is able to process data from all benchmarks and produces highly customizable output. During testing, I was able to reproduce most of the functionality¹ of original Sampler result repository [4]. Resulting Sampler test definition for the repository application is available in file `misc/Sampler.xml` on the supplied CD.

---

¹ The Sampler repository features which the thesis implementation is not able to reproduce are detection of linear dependencies in the results and computing additional values by combining extracted instance attributes.
Illustration 18: Screenshots from the web interface: test description, custom view with overview of uploaded instances, custom view with instance details

Summary of requirements from Chapter 2.2:

- FR1 Result Formats – as was shown in Chapter 4.2.1, the repository is able to parse all specified result formats, and can be set to accept other XML or plaintext input. The application can also be extended to accept new complex data formats by adding new parser plugins.

- FR2 Configuration Information – the repository can be set to store additional data with the test instances. These data can either be entered by the uploader, or extracted from the results. All kinds of metadata can be used in the presentation interface.

- FR3 Plots – the application supports configurable plots, including bar charts, box and whisker plots, density plots and histograms. Ability to reproduce display formats was not checked for all specified benchmarks, because results of some benchmark were no longer available in the time of writing.

- FR4 Web Interface – all repository functions are accessible and configurable through a web interface; optional uploading of multi-file test results in one archive is supported

- FR5 Presentation Functions – the repository employs Django templating systems to provide customizable views of instances and their metadata. It is possible to create template comparing results from one or more test instances. It is possible to filter instances for displaying using user-entered expressions, including regular expression matching.

- FR6 Availability of Original Data – it is possible to download original result data through the web interface

- FR7 Import/Export of Test Descriptions – in the web interface, it is possible to import and export test descriptions (including defined plots and custom views) in a specialized XML format

- FR8 Interface for Other Applications – the implementation provides web service API, allowing third-party applications to fetch basic test information, render custom views, upload instances, and read and write custom data associated with the results

- NFR1 Platform – the repository runs on Linux, using Python/Django and few other commonly available libraries

- NFR2 Performance – installation with Sampler results and approximately 300 uploaded instances has shown that the user interface is responsive enough for browsing, when cached data are used. Still, filtering many instances by unspecified
metadata is going to make regular usage problematic for tests with more than 1000 uploaded results.

- NFR3 Documentation – in addition to user/administrator documentation in Appendix A of this text and programmer's documentation in Chapter 5 and on the CD-ROM, the web interface offers context help for nontrivial tasks, such as designing queries and custom views.

In the end, the product's significant advantage of customizability also brings its drawback, which is problematic performance on large number of test instances. While the processing speed is not a problem for result parsing, faster filtering of already parsed data would be beneficial for those cases. This could be done by saving some of the instance metadata to indexed relational database, which would speed up the filtering, at the expense of significantly complicating the process.

The only other known work dealing with similar assignment is Network Repository for Performance Evaluation Results [8], which uses Java platform. Its metadata structures are more rigid, and it supports indexing of test instance attributes. On the other hand, it does not offer web interface and therefore by itself isn't usable for result presentation.

## 7 Conclusion

While there are many activities which focus on middleware benchmarking, the knowledge they produce is mostly scattered. Most benchmark applications store their results in their own separate repositories which allow only very limited submission of additional data. Result databases which accept results of multiple test types are commercial a do not perform automated data extraction. The notion that follows is that middleware benchmarking could benefit from an open repository, able to handle many types of benchmark results and present their statistics on the web.

This work proposes a repository that converts result data to tree form as a universal representation, and processes tree nodes sequentially, extracting summary values required for plots and presentation data. It is shown that using this approach, output from Sampler, Xampler, RUBiS, TAO, ECperf and OVM can be successfully processed.

The repository implementation, coded in Python on top of Django framework, supports several types of plots, including bar chars, box and whisker plots and histograms. Presentation features are provided by Django templating system and allow simple branching, loops as well as access to metadata extracted from the results.

Additionally, the implementation also allows web service interface for automatic upload of new results, as well as retrieving basic test information and attaching custom data to the results.

In the implementation, alternative Sampler result repository was made, successfully duplicating all its key features, such as result filtering and plot display. It has been found that the design with completely universal metadata structures makes filtering of more than several hundred uploaded results slow due to lack if indexes. However, the resulting work is still usable in most contexts and thus simplifies the task of benchmark result storage and presentation.
8 Bibliography

2. RUBiS, Auction site prototype, http://rubis.objectweb.org/
Appendix A: User manual

Table of Contents

A1 System Requirements ........................................................................................................ 35
A2 Installation and Setup .................................................................................................... 35
A3 Overview ....................................................................................................................... 37
A4 Test Files ....................................................................................................................... 37
A5 Instance Attributes ........................................................................................................ 39
A6 Queries ........................................................................................................................... 40
A7 Plots ................................................................................................................................ 42
A8 Custom Views ................................................................................................................ 45
A9 Import and Export ......................................................................................................... 48
A10 User Accounts and Web Service API ......................................................................... 48

A1 System Requirements
The repository is designed to work on a GNU/Linux. All required packages were found in Debian and Gentoo distributions, but the software should be available on most Linux flavors.

The requirements are:

- Python 2.3 and higher with database bindings:
  - either psycopg for PostgreSQL (python-psycopg2 in Debian, psycopg in Gentoo)
  - or MySQLdb >=1.2.1p2 (python-mysqldb in Debian, mysql-python in Gentoo)
- Apache and mod_python
- Biggles (python-biggles package in both Debian and Gentoo)
- elementtree (python-elementtree in Debian, elementtree in Gentoo, built-in since Python 2.5)
- ZSI 2.0 (python-zsi in Debian – it's necessary to pull testing version, zsi in Gentoo; please not that ZSI 2.1 has different API and is probably incompatible with current implementation)
- Django development version (This version is not commonly available in distribution packages. You can use instructions below, or see detailed installation instructions at http://www.djangoproject.com/. Last tested revision was 7420.)

A2 Installation and Setup
We shall perform the installation under root account. The daemon processes will then be running under different account. For the purpose of this guide, let's assume that we don't want to deploy development software directly into our system, so we create a directory, and put everything there.
# mkdir /home/repository; cd /home/repository

First, copy the Django source tree from enclosed CD-ROM:

```
/home/repository# cp -R /mnt/cdrom/django-trunk ./
```

Alternatively, you can use subversion to get the fresh trunk version:

```
/home/repository# svn co \ http://code.djangoproject.com/svn/django/trunk/ \ django-trunk
```

Then, you need to copy repository source codes from the CD:

```
/home/repository# cp -R /mnt/cdrom/sklad ./
```

Create directory for site-wide templates and add a HTTP 404 error message:

```
/home/repository# mkdir templates
/home/repository# cp /mnt/cdrom/misc/404.html templates/
```

Create directory for static document root and link repository media there:

```
/home/repository# mkdir webroot
/home/repository# ln -s ../sklad/skladr/media/ webroot/media
```

Set up Django site. This is done by Django admin script, for which we need to set up the Python paths, which we are going to do in the environment at first:

```
/home/repository# export PYTHONPATH=/home/repository/django-trunk
/home/repository# django-trunk/django/bin/django-admin.py \ startproject website
```

This will create a new directory (called “website”), where the most interesting file is settings.py which contains settings of your new site. Open it up in an editor and adapt it to your needs, using file `misc/settings.py.example` on the CD as a reference.

The alteration of Python path we made in the current environment is not going to last, so in the following step, we are going to fix this by adding the paths to file `website/manage.py` that we'll be using to access various function from the command line.

Open manage.py and add the following two lines just below `#!/usr/bin/env python`:

```python
import sys
sys.path = ['/home/repository', '/home/repository/django-trunk'] + sys.path
```

Then, we can use the manage.py utility to create data structures in the database, as well as the repository itself in the filesystem, together with its admin account:

```
/home/repository# cd website
/home/repository/website# python manage.py syncdb
```

The script will ask you if you want to create a superuser for Django auth system. You don't need to do this as repository has its own user management.

```
/home/repository/website# python manage.py sklad-createrepo \ --admin-password <new_admin_password>
```

If you have previously set the site to cache information in a database table, you also need to create the table:

```
/home/repository/website# python manage.py createcachetable cachetable
```
Now we're almost finished. You can run Django built-in webserver to check that the web interface is working:

```
# python manage.py runserver
```

When the server starts up, you can browse the web interface (although without styles and images) on port 8000 to check it's working.

Then, copy the init script from misc/sklad-services.init somewhere to you system (most people prefer `/etc/init.d/`) and edit the paths inside it to point to your installation.

After that, you can start and stop repository web service and task processor by running

```bash
# /etc/init.d/sklad-services.init start
# /etc/init.d/sklad-services.init stop
```

Finally, we need to set up Apache and mod_python. Use example virtual host configuration in `misc/vhost.conf.example`

When Apache is set up, restart it and the repository should be working. You can start exploring it by importing Sampler test definition from `misc/Sampler.xml`

### A3 Overview

New test can be created by any logged-in user from the main page of web interface. You need to fill in test name and description (for overview on the main page, both can be changed later) or you can import test from local file.

In the repository, definition of a test consists of the following attributes:

- **Files** – files the make up the test results
- **Instance attributes** – attributes that are assigned to instances of the test (i.e. uploaded results)
- **Queries** – used for extracting data from the results, either for automatic filling of instance attributes or as base for the plots
- **Plots** – graphical summaries of the result data
- **User (or Custom) views** – defines presentation pages for the results

### A4 Test Files

“Files” section contains definition of files that are contained in every test instance. Every uploaded instance has to contain these files. For every file, there is also a definition of plugin that is used to convert its data to a tree structure that is subsequently processed by data queries.
By default, the following plugins are available:

- **NOOP** – does not parse anything; can be used for files that are part of the results but contain no useful data
- **plaintext grep** – parses file by given regular expression. Python `re` module is used for parsing, detailed description of used syntax is available at [http://docs.python.org/lib/re-syntax.html](http://docs.python.org/lib/re-syntax.html). For every successful match, the plugin produces a branch with leaf values filled in by matched groups of the expression.
- **XML data** – parses an XML document; the tree nodes correspond to document elements, leaf values are generated for element attributes and text contents. Optionally, elements whole path corresponds to configured format can be split to several leaves.
- **Xampler** – file is treated as Xampler output, produces nodes for found inside, with leaf values for strings and single integers. Multiple integer values are represented as nodes with leaves.
- **OVM tarball** – file is treated as tar-compressed directory containing OVM output. The parser reads the index files, and produces a branch with “info” a “data” nodes, where “info” contains all config information found in the index, and “data” values from raw output file linked by particular the index section.

For results with one file, the parser produces directly output of the parsing plugin. For multi-file results the parser outputs one branch for every file that contains trees produces by the respective plugins.
You can check the parser output by uploading an instance and expanding the tree root found under “Data” header. For performance reasons, the display is limited to 50 children per node.

**A5 Instance Attributes**

Instance attributes are metadata which are stored together with uploaded test instances. They can be either filled in by the uploader, or computed from the results.

User-entered attributes can be either string or boolean, third type is “query” which identifies attribute that takes its value from the result of specified data query.

*Illustration 21: Defining instance attribute and upload form with attribute entry field*
A6 Queries

Queries describe how to extract values from result data tree. Their output is used for plots, and can also be set fill in instance attributes.

The main parts of a query are three expressions: “where”, “group by” and “value”, and one aggregation function. The query processing works as follows:

1. For every leaf of the tree, “where” expression is evaluated. If the resulting value is False, processing of the leaf is cancelled. Otherwise, we move onto the next step.

2. “Group by” and “value” expressions are evaluated. If neither of the results is None the computed value is aggregated under “group by” key by the chosen aggregation function.

The final result is a set of grouping keys, together with aggregated values.

For example, if we had used the plaintext grep plugin to parse our phonebook (with results similar tree in Illustration 23), and wanted to get a statistics of how many of our friend's names start with which letter, we could have used the following query:

- “where” expression would be used to filter nodes that contain the names. We can:
  - test name of the last node: `path[-1].name == 'name'`

Illustration 22: Definition of a query.

Illustration 23: Example of a result data tree
o use shortcut function to check whole path matches a given format: 
\[\text{simpleLike(path, '/None/name')}\]

- We want to group our values by the first character of the name, so for “group by” we use another shortcut function to get the value of last node and then take its first character: \[\text{val(path)[0]}\]

- As “value”, we enter 1 (constant one), and choose to aggregate value using “Sum” function. (We can also choose to aggregate using “Count” function, which ignores the value)

During the query editing, you have option to execute the query on a data of some instance to check the results. The result is a Python dictionary, in our case \{'A':2, 'C':1, 'E':1, 'D':1, 'J':2, 'M':2, 'W':1\}.

However, having information only from one node path at a time is not enough for most situations. For example, in the snippet of parsed Sampler output on Illustration 24, we also need other information from preceding nodes to fully appreciate the data we get in the marked leaf. For one, we need to know that the benchmark type is “Parallel” to be able to choose the right results. Also, we might want to know the value of “Simul” attribute to be able to group the results.

To access these additional data, the queries have a facility of “bonus” data, which allow you to define paths whose values you want to have available for the main expressions of the query.

An example of query using bonus data would be the query which collects data for “Multithreading plot” in the Sampler results repository:

- Bonus values are comma-separated typed and have format “name=path”, in this case it's \[\text{type=\ldots/\ldots/Type, timer=\ldots/\ldots/Timer/Scale, simul=\ldots/\ldots/Simul}\] which, in addition to already mentioned useful field, uses absolute path to get value of timer, to transform the numbers to a meaningful unit

- Where is \[\text{simpleLike(path, '/Results/Session/Benchmark/- Measurement/Sample/s=3')}\] and \[\text{bonus['type']]=='Parallel'}\]; here we access the 'type' value from the “bonus” dictionary. Another new concept is matching the last node in the path by its serial number (which can be seen in the tree display in the parentheses). This can be also used in path definition for bonus values.

- Group by expression is another bonus value: \[\text{bonus['simul']}\]

- We want the value in microseconds, so we have to multiply number of clock cycles and then divide it by timer frequency: \[\text{float(val(path) * 1000000) / bonus['timer']}\]

- Aggregation function is Avg or Median.

The result of the query is dictionary where keys are 'Simul' values (i.e. number of running threads) and values average (or median) duration of test run in microseconds.

In addition to fields described above, you can also fill in unit description for both “group by” and “value” expressions. These descriptions are used in some of the plots.
Queries can also be marked as “simple”. Simple queries have fixed “group by” expression, and their result is therefore a single number. Such queries are useful to fill in instance attributes, where values such as “broker used on server” or “average ping” are commonly desired. However, assigning non-simple queries to instance attributes is possible as well.

**Symbols Accessible Within the Expressions**

In addition to `path` and `bonus` arguments and language constructs built in Python, the query expressions have access to the following symbols:

- **simpleLike(path, format)** – checks if path fits in specified format. *
  * matches any path element, `None` matches nameless elements, `s=n` matches element with given serial number. Whole format is slash-separated string, such as `/Results/Session/Benchmark/*/Sample/s=3`

- **tailLike(path, format)** – similar to `simpleLike`, but checks only is path ends with a sequence specified to the format

- **serialPrefix(path, format)** – checks if the serial numbers of path elements have given prefix. Mostly for internal use.

- **ifelse(condition, on_true, on_false)** – simple in-expression branching construct, can be used as slightly more readable alternative to `{True: on_true, False: on_false}[condition]`

- **val(path)** – value contained in the last element of the path, shortcut to `path[-1].value`

- **isData(path)** – checks if the last element of the path is nameless, shortcut to `path[-1].name == None`

- **int, float, long, str, True, False** – standard Python types and constants

- **min, max, abs** – min, max and abs functions

- **math** – Python math module, with functions such as `math.log` and `math.floor`. See [http://docs.python.org/lib/module-math.html](http://docs.python.org/lib/module-math.html) for complete reference.

**Plots**

Plots are present result data in graphical form. In the repository, there are three types of plots:

- regular plots, which draw keys of the query results on X axis and the values on Y axis
- bar charts, which show values of key by the height of bars
- histograms, which treat values of the result as number of occurrences of their keys
Illustration 25: Definition of a Plot

What is drawn in a regular plot is defined by its set of elements. The repository supports the following types of elements:

- **points** – can display either symbols such as “dot” or “circle”\(^2\) or a single character
- **curve** – a line connecting the points of the result
- **ErrorBarsY** – vertical line connecting two values (takes results from two queries)
- **Median + error bars for quartiles** – draws a point at median value and error bars for quartiles (takes values from one query, which has to use “MedianQuartilesExtremes” aggregation function to produce multiple values for this component)
- **Median + quartiles + extremes** (as previous, but draws box with whiskers)
- **Bars** – element for use in bar charts

Regular plot can use data from more queries, but these queries have to share the same set of keys.

\(^2\) The full set of supported symbol types is: none, dot, plus, asterisk, circle, cross, square, triangle, diamond, star, inverted triangle, starburst, fancy plus, fancy cross, fancy square, fancy diamond, filled circle, filled square, filled triangle, filled diamond, filled inverted triangle, filled fancy square, filled fancy diamond, half filled circle, half filled square, half filled triangle, half filled diamond, half filled inverted triangle, half filled fancy square, half filled fancy diamond, octagon, filled octagon
Bar Charts

Bar charts do not arrange key value by their size, so you can use strings as group names. The only other difference is that only the “bar” elements have to be used.

Illustration 26: A regular plot with four elements showing points and curves for two queries

Histograms

The last type of plot is a histogram, which can only show data from one query, but compensates this drawback by manipulating the data to provide optimal display.

The histograms takes the keys of its input as values, and the actual values contained in the dictionary are looked upon as count of key occurrences in the set.

For example, a dictionary result of \{1: 1, 2: 2, 3: 5, 4: 3\} would be interpreted as set of 11 values, where 1 is present once, 2 twice, 3 five times and 4 three times.
As to the data manipulations: the plots automatically scale its display, so that all values fit in. This is usually a good thing, but it means that one extreme value can reduce the interesting part to one tiny bar on the side, and thus completely throw off whole plot. To prevent this, the repository can be set to discard certain percentage of extreme values from the set. This way, the plot can contain distribution of (for example) 97% of the values, ignoring random measurement errors.

Because this approach can also cut out sensible values on the borders of the interval, it's also possible to extend the interval again, after adjusting it to fit set percentage of values. Therefore, you can both discard the extremes and keep top and bottom end of the error-free result distribution.

Finally, you can select into how many groups the histogram should divide the obtained interval, either by setting the number directly, or by letting the plot processor start with small groups and join them together until the biggest one contains certain percentage of values.

### A8 Custom Views

Custom views are a feature you can use to create presentation pages for result data. In essence, custom views are templates that format test and test instance metadata for public viewing. For processing custom views, the repository uses Django templating system, which is documented in detail at [http://www.djangoproject.com/documentation/templates/](http://www.djangoproject.com/documentation/templates/)

The following text describes the basic aspects of the template language, and its extensions specific to the repository.

The template engine is able to perform variable substitution, and well as simple branching and loops. Let's start with an example:

```
<html><head><title>{{ test.name }}</title></head><body>
<h1>This is my test, {{ test.name }}</h1>
<p>We have {{ test.getDictInstances|length }} instances.</p>
<table>
<tr><th>id</th><th>broker</th><th>average ping</th></tr>
{% for instance in test.getDictInstances %}
<tr><td>{{ instance.id }}</td>
<td>{{ instance.broker }}</td>
<td>{{ instance.avg_ping }}</td>
</tr>{% endfor %}
</table></body></html>
```

*Illustration 29: Example of a custom view*

This is basically code of HTML page, although custom views can be used to generate any-text base data. We can see that variable substitution is done by double curly braces, and that `for` tag can be used to iterate a list.

The template system is able to look up attributes, items and execute some object methods. All custom views get `test` variable, containing object representation of current test. Among other properties, the object has “name” attribute, which contains name of the test, and also
provides `getDictInstances` method, which returns a list of dictionaries describing the instances. The for tag loops over the list: for every list item, we have the dictionary assigned to variable “instance”. The dictionary contains the following items:

- **id** – instance identifier
- **uploaded** – timestamp of instance creation
- **trust** – one of “registered” (registered repository user), “email” (has verified e-mail address) or “anonymous” (completely anonymous result)
- **owner** – name of user who uploaded the instance (for those uploaded by a registered user)
- **parsed_ok** – flag indicating the results were successfully parsed. Can be `True`, `False` or `None` (for instance which weren't parsed yet)
- one value for every instance attribute set for the test (as “broker” and “avg_ping” in the example)

It is possible to “pipe” substituted value through various filters, such as length in the example. There are many filters available, one of which is `dictsort` that can sort the list of dictionaries by key given in its argument. For example, if we used `test.getDictInstances|dictsort:”avg_ping”` in the specification of the loop, the displayed rows would have been sorted by `avg_ping` attribute.

### Types of Custom Views

There are three types of custom views:

- **per-test** – display data global for the whole test (such as general information or the list of uploaded instances)
- **per-instance** – display data of one instance, also get `instance` variable with dictionary contents describing the selected instance
- **multi-instance** – display data of selected instances, get `instance_list` variable which contains list of dictionaries, describing the instances. Multi-instance view can be invoked either by naming instance id's in the querystring (for example by using form with checkbox for each instance) or by choosing the instances by `instance filters`, which are described below.

All custom views get these variables:

- **test** – test object, containing `.name` and `.desc` properties, `.owner` who owns the test, and most interestingly `.getDictInstances` method that returns list of dictionaries that represent instances of the test and is described above.
- **repo_user** – currently logged-in user, has properties such as `.name`, `.email`, `.isAnon` (is anonymous) and `.admin` (is administrator)
- **repository** – represents whole repository, has methods `.getUserList` (returns list of users) and `.getTestList` (returns list of tests)

If there is a per-test custom view named `index.html`, it will be offered as a default page for test when selecting it from the repository main page. Additionally, it's also possible to set a per-test custom view to be displayed to the user who has cancelled uploading of a new instance, and a per-instance custom view for display upon successful upload.
Additional Tags

In addition to tags and filters built in to the template processor (full list is in above-mentioned template documentation), the repository has some additional template tags:

- **test_view** – URL of a test view, for example \{% test_view "index.html" %\}

- **instance_view** – URL of an instance view: \{% instance_view instance.id "instance.html" %\} is expanded to link to “instance.html” view showing the mentioned instance

- **multiinstance_view** – URL of a multi-instance view. Multi-instance views expect list of instance identifiers in their GET or POST arguments, so in reality the URL would look something like \{% multiinstance_view "more.html" %\}?inst_id_1&inst_id_2 or you would use a form with sequence of checkboxes.

- **filtered_view** – URL of multi-instance view that has its list of instances filtered by an instance filter. Instance filter takes its arguments from GET or POST variables. Example: \{% filtered_view "id_prefix" "more.html" %\}

- **instance_plot** – URL of an instance plot image. Example: \{% instance_plot instance.id "my_plot" %\}

Instance Filters

Instance filters are Python expressions, choosing what should be displayed in a multi-instance view. The expression evaluates two dictionaries: **instance** (that contains attributes of an instance) and **args** (that contains optional argument to the filter). Instances for which the expression evaluates to True are displayed in the view.

[Image: Illustration 30: Definition of an instance filter]
For example, filter for instances uploaded by registered users and with a particular user-specified broker “foo” attribute:

```python
instance['foo'] == args['foo'] and instance['trust'] == 'registered'
```

In web interface, the arguments can be entered through GET or POST queries; this makes the filters usable for search forms defined in custom views.

In addition to `instance` and `args` dictionaries, the following symbols are available to the filter expressions:

- `ifelse(condition, on_true, on_false)` – simple branching shortcut
- `int, float, long, str, True, False` – standard Python types and constants
- `min, max, abs` – min, max and abs functions
- `re` module for regular expressions, including function `re.match(pattern, string)`; see [http://docs.python.org/lib/module-re.html](http://docs.python.org/lib/module-re.html) for detailed module description

**A9 Import and Export**

The web interface provides the option to export whole or partial definition of test into an XML file. This file can then be used to create a new test, or selected parts of it can be added to another test, allowing transfer of plots etc. between test and/or repository installations.

**A10 User Accounts and Web Service API**

Repository administrator can add and edit user through “manage users” link on the main page of the web. From the interface, the administrator can change user's name and e-mail addresses, switch their “administrator” status and change login passwords. It's also possible to disable login, which prohibits the user from logging into the web interface.

The “API key” section can set and disable API key, which is used for write access through the
web service. If the user has an API key, this section also shows web service URL and link to WSDL files (which is also available on the CD, in file sklad/api/Sklad.wsdl, and contains details specification of input and output formats used).

The web service provides the following functions:

- **getSessionChallenge** – input is a user id, output a challenge string
- **registerSession** – input is a user id and key, computed as hexadecimal MD5 digest of the challenge and user's API key; output is a session identifier for write operations
- **getTestDetails** – no input, output is a sequence of test information, containing id, name and description
- **getPlotUrl** – input is an identifier of test, plot and instance, the output is a URL of a plot and boolean flag indicating whether it is successfully rendered
- **createInstance** – input is a session id, test id and set of instance attributes and data files, the output is identifier of the new instance
- **renderTestView** – input is a test id and id of the view, output is content type and the rendered view
- **renderInstanceView** – input is a test id, view id and identifier of the instance, output is content type and the rendered view
- **renderMultiinstanceView** – input is a test id, view id and sequence of instance identifiers, output is content type and the rendered view
- **renderFilteredMultiinstanceView** – input is a test id, view id, filter id, and sequence of argument names and values, output is content type and the rendered view
- **listTestAppFiles** – input is test id and application identifier, output is list of file names
- **listInstanceAppFiles** – input is test id, instance id and application identifier, output is list of file names
- **getTestAppFile** – input is a test id, application identifier and file name, output is content of the file
- **getInstanceAppFile** – input is a test id, instance id, application identifier, file name and contents; there is no output
- **putTestAppFile** – input is a session id, test id, application identifier, file name and contents; there is no output
- **putInstanceAppFile** – input is a session id, test id, instance id, application identifier, file name and contents; there is no output

In case of error, the web service returns a generic fault message, where the error condition is provided in text description part. This is due to limitations of used web service framework.
Appendix B: CD-ROM Contents

- `sklad` – repository source files
- `doc`
  - `thesis.pdf` – electronic version of this thesis
  - `pydoc` – generated source documentation
- `testdata` – data used for regression tests
- `misc` – additional files for deployment
- `django-trunk` – development version of Django framework
- `offline-copy` – snapshot of repository web interface for offline viewing