

BEEN

User Documentation

Table of Contents

Part I: BEEN Overview.....	7
1 Introduction.....	8
1.1 License.....	9
2 Architecture Overview.....	10
2.1 Execution Framework.....	10
2.2 Benchmarking Framework.....	11
2.3 Technologies Used.....	11
3 Tasks.....	13
3.1 Task Communication and Synchronization.....	13
3.2 Task Descriptor.....	13
3.3 Task Status.....	14
4 Host Runtime.....	16
4.1 Host Runtime Lifecycle.....	16
4.2 Running Tasks.....	16
4.3 Exclusive and Context-Exclusive Tasks.....	17
4.4 Load Monitor.....	17
4.5 Data Directory Structure.....	18
5 Task Manager.....	19
5.1 Configuration.....	20
6 Software Repository.....	21
6.1 Packages.....	21
6.2 Operations Supported.....	23
6.3 Package Storage.....	23
6.4 Optimizations.....	23
7 Managing Computers on the Network.....	24
7.1 Host Database.....	24
7.2 Host Data.....	24
7.3 Database Queries	25
7.4 Host Groups	26
7.5 Software Aliases.....	26
7.6 Host Manager Configuration.....	27
8 Monitoring Computer Utilization	29
8.1 Monitoring Running Tasks.....	29
8.2 Host Utilization Monitoring.....	29
9 Benchmark Manager.....	31
9.1 Benchmark Entities.....	31
9.2 Benchmark Plugins.....	32
10 Results Repository.....	36
10.1 Results Collection.....	36
10.2 Results Repository Database.....	36
10.3 Failed Runs.....	37
10.4 Statistics Calculation.....	37
10.5 Cooperation with Benchmark Manager.....	37
10.6 Database Contents.....	38
10.7 Results Repository Export Format.....	39

11 Restriction Specification Language (RSL).....	40
11.1 Language Description.....	40
12 Related projects.....	44
 Part II: Using BEEN.....	 45
1 Installing BEEN.....	46
1.1 Requirements.....	46
1.2 Installing BEEN Execution Environment.....	47
1.3 Installing Results Repository Prerequisites.....	47
1.4 Installing Web User Interface.....	49
2 Running BEEN.....	50
2.1 Running Task Manager.....	50
2.2 Running Host Runtime.....	51
2.3 Running and Configuring Web Interface.....	51
2.4 Running Services.....	52
3 Using Web User Interface.....	53
3.1 Web Interface Structure.....	53
3.2 Packages Module.....	54
3.3 Hosts Module.....	55
3.4 Tasks Module.....	57
3.5 Benchmarks Module.....	60
3.6 Results Module.....	64
3.7 Configuration Module.....	69
3.8 Services Module.....	70
4 Benchmarking with Xampler.....	72
4.1 Introduction to Xampler.....	72
4.2 Comparison Analyses with Xampler.....	72
4.3 Regression Analyses with Xampler.....	73
4.4 Required Packages.....	73
4.5 Xampler Role Requirements.....	74
4.6 Notes.....	74
5 Benchmarking with RUBiS	75
5.1 Client Emulator.....	75
5.2 Server.....	75
5.3 Database.....	77
5.4 Using the RUBiS Plugin in BEEN.....	77
6 Executing a Simple Benchmarking Analysis.....	82
6.1 Executing RUBiS Comparison Analysis.....	82
7 Compiling BEEN.....	85
 Part III: Extending BEEN.....	 87
1 Introduction.....	88
2 Writing Custom Tasks	89
2.1 Packaging.....	89
2.2 Task Class.....	90
2.3 Jobs and Services.....	91
2.4 Existing Tasks.....	92
3 Extending Benchmark Manager.....	107
3.1 Packaging.....	108
3.2 Experiment Metadata.....	109
3.3 Configurator.....	110

3.4 Task Generator.....	112
3.5 Version Provider.....	114
4 Extending Results Repository.....	115
4.1 Validation and Conversion Tasks.....	115
4.2 R Script Packages.....	115
4.3 Writing R Functions.....	116
5 Debugging BEEN.....	117
5.1 Debugging Host Runtime and Task Manager.....	117
5.2 Debugging Tasks.....	117
5.3 Debugging Services.....	117
5.4 Debugging Load Monitor.....	118
Part IV: Appendices.....	119
Appendix A: Objects and Properties of the Host.....	120
Appendix B: Results Repository Metadata.....	124
1.1 Analysis Metadata.....	124
1.2 Experiment Metadata.....	124
1.3 Binary Metadata.....	125
1.4 Run Metadata.....	125
Appendix C: RSL Grammar.....	126
Appendix D: Source Code Layout.....	128
Appendix E: Third-party Libraries.....	129

Documentation Structure

The user documentation is divided into four parts.

In Part I, we introduce BEEN, explain the overall architecture of the system and describe its components.

In Part II we describe hardware and software requirements of the BEEN; installation, running and compilation instructions. We also describe functionality of the web user interface and both benchmarks currently supported by the BEEN.

In Part III we describe how to add support for additional benchmarking software and extension points of various components of the BEEN.

In Part IV, various appendices with supplemental information are listed.

In the documentation, we often quote article *Automated Benchmarking and Analysis Tool*¹, which we have written for the VALUETOOLS 2006 conference. The quotations are not specially marked.

For clarity, some technical details are omitted from the user documentation and the reader is referred to the automatically generated Javadoc documentation, available in electronic form.

¹ http://nenya.ms.mff.cuni.cz/publications/Submitted_1404_BEEN.pdf

Part I

BEEN Overview

1 Introduction

When developing and evaluating software, it is often useful to automatically measure performance of the software and compare it with other similar products or with older versions of the same software. One of the possibilities of performance measurement is to run a model application (a benchmark) which simulates the behavior of real application and provides performance data. Usually, such benchmarks are conducted using tools built ad-hoc for each project, resulting in minimal code reuse and waste of resources. When benchmarking inherently distributed software, such as client-server applications or middleware, the construction of a benchmarking framework is a non-trivial task, as the framework has to cope with tasks such as distributed deployment, synchronization, monitoring and collection of results. Clearly, there is a place for a general framework for conducting benchmarks in a distributed environment.

A special application of benchmarking is automatic performance evaluation of software during development – *regression benchmarking*. It is useful mainly to detect code changes with significant positive or negative impact on software performance. Regression benchmarking requires precise measurement, complicated statistical evaluation and full automation of the benchmark execution (downloading the software, compilation, etc.).

We have solved the outlined problems in BEEN and created a generic tool for regression benchmarking in a heterogeneous distributed environment. BEEN presents a common execution environment suitable for running many kinds of different benchmarks (i.e. its architecture is not tied to any specific benchmark). The notable features include:

- **Support for heterogeneous environment.** BEEN can be run on Windows and Linux platforms and in limited way on any platform capable of running Java.
- **Regression benchmarking.** It is easy to setup BEEN to scenarios such as download of the daily snapshot of a product from the source code repository, execution of specified benchmarks on the snapshot and processing of the results.
- **Automation.** Everything is fully automatic from downloading and compilation of the benchmarked software, deploying software to hosts and running the benchmarks; to collecting and processing results. This allows for unattended execution of regression benchmarks.
- **Complete statistical analysis and visualization of the results.** Results of the benchmarks can be statistically processed and various graphs can be generated from the results. Standard statistical environment R is used at the core of the results processing component. The results are always stored as a raw data; no information is lost by the system.
- **User-friendly web interface.** The whole framework is controlled through a unified web interface – there is no need to install any special applications and the framework can be controlled from any computer with a web browser. Special care was taken to make the interface as easy to use as possible.
- **Logging and debugging facilities.** Status of all BEEN components can be easily monitored. Almost every action in the system is logged and the logs are sent to a central repository, where they can be inspected via a web interface.
- **Extensibility.** BEEN can be extended to support additional benchmarks via plugins. The statistical processing is customizable by writing scripts in the R language. User may use a rich library of built-in functions while working with the benchmark results.

BEEN supports two benchmarks out-of-the-box:

- **Xampler**² – a CORBA middleware benchmark
- **RUBiS**³ – an EJB server benchmark

BEEN was developed in close cooperation with Distributed Systems Research Group⁴ at the Faculty of Mathematics and Physics of the Charles University in Prague. The project was also accepted as an ObjectWeb⁵ project and is hosted on ObjectWeb Forge⁶.

The project website is located at **<http://been.objectweb.org/>**.

1.1 License

BEEN source code is distributed under the terms of the GNU Lesser General Public License (LGPL). Full text of the license can be found in the `LICENSE` file in BEEN source directory.

BEEN uses several third-party libraries. See *Third-party libraries (Part IV, Appendix E)* for the list of the libraries, their files and licenses.

² <http://dsrg.mff.cuni.cz/~ceres/prj/CCPsuite/>

³ <http://rubis.objectweb.org/>

⁴ <http://nenya.ms.mff.cuni.cz/>

⁵ <http://www.objectweb.org/>

⁶ <http://forge.objectweb.org/>

2 Architecture Overview

Main design goal of BEEN is to support automated benchmarking in a distributed heterogeneous environment. The automated benchmarking involves compilation of software to be benchmarked, compilation of benchmarks, deployment, execution of the benchmarks and collection, evaluation and visualization of the results. To achieve those goals, the BEEN itself had to be built as a distributed system. Most of the goals are common for automated execution of software in a distributed environment in general. In an attempt to keep BEEN general enough to support execution of any distributed software we split framework into two parts – the *execution framework* and the *benchmarking framework*.

The execution framework is a general framework designed to execute *tasks* in a distributed system abstracting away differences between various platforms. The benchmarking framework is built on top of the execution framework and covers benchmark-specific functionality. Both frameworks are described below.

All components of BEEN can be monitored and controlled from the web user interface. The web interface provides both high-level operations (such as starting a benchmark and viewing its results) as well as low-level operations (such as execution of a specific task). User can view detailed information (logs, status, etc.) about all processes being executed by the BEEN from the user interface. The user interface runs independently and can be shut down while other BEEN components are running.

2.1 Execution Framework

The execution framework of BEEN is a general framework designed to execute *tasks* in a distributed environment while hiding differences between various operating systems and platforms.

Task is a basic unit of execution in BEEN. Depending on its mode of execution, each task is either a *job* or a *service*. A job is a batch task created to perform a particular action – it finishes as soon as the action it was created for was performed. A service is a long-running task that waits for requests from other tasks and performs actions upon those requests. Most of BEEN components are implemented as services: Software Repository, Host Manager, Benchmark Manager and Results Repository.

Each participating host has to run a *Host Runtime*. The Host Runtime is responsible for execution and management of all tasks on its host and serves as a proxy for all communication between BEEN components and tasks running on the host. Integral part of the Host Runtime is the *Load Monitor*, which monitors utilization of various resources (memory, disk space, network bandwidth, etc.) on the host.

The information about hosts in the benchmarking environment is maintained by the *Host Manager*. This service maintains a list of hosts, their status, hardware/software configuration and resource utilization. Other BEEN components can lookup hosts based on various criteria.

The execution of tasks in the distributed environment is coordinated by the *Task Manager*. The Task Manager allocates hosts to tasks based on the task requirements, monitors the running tasks and resolves task failures.

Code and data of the tasks are stored in the *Software Repository*. It stores software sources, binaries for different platforms as well as tasks and static data. By using the Software Repository, the execution framework avoids relying on a distributed file system which might

be difficult to set up in a heterogeneous environment. When executing benchmarks, the presence of a distributed file system could also distort the benchmark results.

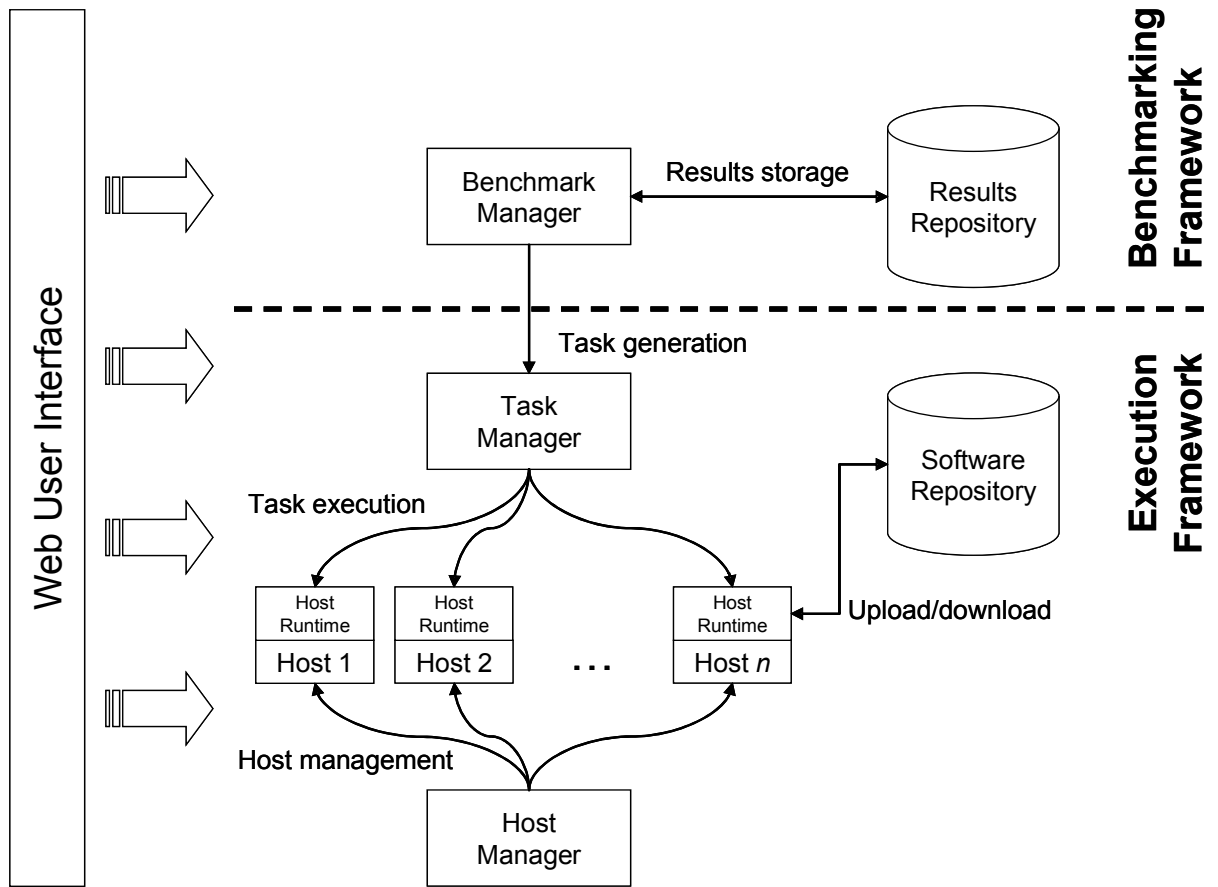


Diagram 1: BEEN Architecture.

2.2 Benchmarking Framework

The benchmarking framework is designed to support two different kinds of performance analysis – a traditional one-time evaluation of performance and repetitive evaluation of performance for regression benchmarking. The main components of the benchmarking framework are the *Benchmark Manager* and the *Results Repository*.

The Benchmark Manager maintains all information needed to run benchmarks – this typically involves compilation of benchmarked software, compilation of the benchmarks itself, deployment and execution of the benchmark. It utilizes a system of *plugins*, which add support for specific benchmarks.

Experiment results are stored in the Results Repository in a raw format that contains individual benchmarks measurements. The results can be statistically processed and visualized in a customizable way.

2.3 Technologies Used

In the BEEN implementation we used the Java language and runtime environment (version 5.0 is required), because of its multiplatform nature and easy way of code distribution (Java programs compile into a bytecode which could be easily transferred between the hosts and

executed by the Java Virtual Machine). Also, Java provides built-in mechanism for remote procedure call: Remote Method Invocation (RMI). We use RMI for all communication between components in the distributed environment.

We also use Java Native Interface (JNI) technology as a means of communication with native libraries which provide low-level access to the operating system needed by the Host Manager and Load Monitor.

The web interface is also written in Java using the Java Servlets technology and the Java Server Pages (JSP) template language. This allows great interoperability with the rest of BEEN and easy deployment using Apache Tomcat, an open-source servlet container.

Parts of the Results Repository which deal with statistical processing of the benchmarks results are written in the R language, which is a standard open-source tool for statistical computations.

3 Tasks

The purpose of tasks is similar to ordinary program or process, i.e. they perform the low level functionality used by benchmarks, e.g. compilation of software source code.

Each task is either a *job* or a *service*. A job is a batch task created for a particular action – it finishes as soon as the action it was created for was performed. A service is a long-running task which waits for requests from other tasks and performs actions upon those requests. It can be compared to Unix daemon. Most of BEEN components are implemented as services.

Task's code, data and additional information required by BEEN are stored in the Software Repository in *task packages*. The tasks are managed by the Task Manager, which decides when and where to run a task. When a task is about to be started, its package is transported over the network to a Host Runtime on the target host. The Host Runtime creates a new Java Virtual Machine for the task (so that crashing task does not crash the Host Runtime) and manages the task execution.

Each group of cooperating tasks is enclosed in a single *context*. Tasks and contexts are identified by a textual ID – context ID is globally unique, task ID is unique within a context. Special context – *system context* – in which all BEEN services are running is created automatically by the Task Manager. The system context is never deleted.

3.1 Task Communication and Synchronization

Synchronization and communication between tasks is handled via *checkpoints* which are managed centrally by the Task Manager. Each checkpoint has a textual name and optionally a value which can be any Java object capable of transferring over the network (*Serializable* object). A task can *set a checkpoint* by registering it in the Task Manager. Another task can then query for checkpoints and their values.

A checkpoint can be used by a task to indicate a state. For example, a task can set a checkpoint with the name `database_running` to indicate that a database has been successfully started. The value of the checkpoint can be used to send information to other tasks, for example it could be a CORBA IOR reference.

To get the value of a checkpoint, a task has to specify a task ID of the task that set the checkpoint and the name of the checkpoint. A task can also wait for a checkpoint to be set by a specific task. Each task can only synchronize with the task from the same context or from the *system* context.

3.2 Task Descriptor

To run a task, a *task descriptor* must be submitted to the Task Manager. The task descriptor is a data structure containing all information necessary for starting the task. It can be stored in an XML file or in a Java object. For detailed description of the XML file format of the task descriptor, see file `/resources/dtd/taskmanager/TaskDescriptor.dtd` in the BEEN source code.

Following information is stored in the task descriptor:

- **Package** – identification of a package in the Software Repository that contains the task code and data. The package can be identified by its name, or by an RSL expression

which can contain requirements on the package version, supported platforms, etc. For more information about RSL, see section *Restriction Specification Language (RSL) (Part I, II)*.

- **Task Properties** – the equivalent of command-line arguments for tasks are the *task properties*. Each property of a task has a name and a value. Name of the property is string, value of the property can be any object. Task properties are not transferred via command-line arguments thus avoiding various problems caused by the limitations of the command-line on various platforms.
- **Host** – the task descriptor must specify a host on which the task should be executed. The host can be specified directly by its hostname or it can be selected by RSL query. The RSL can contain hardware and software requirements on the host. The query will be evaluated by the Host Manager, the task will then be executed on one of the matching hosts.
- **Dependencies** – to ensure a desired order of task execution, the start of a task can depend on another task setting a checkpoint. A task is started only when all its dependencies are satisfied. A dependency on a checkpoint contains this information:
 - **Task ID** – identification of the task that must set the checkpoint.
 - **Checkpoint name** – name of the checkpoint that must be set (e.g. database started).
 - **Checkpoint value** – the checkpoint must have this value to satisfy the dependency. If the requested value is `null`, any value is acceptable.

Several checkpoints are set by the BEEN automatically. When a task is started, the Host Runtime sets the `task started` checkpoint to a `null` value. When a task finishes, Task Manager sets the `task finished` checkpoint to the return value of the process in which the task was running.

- **Exclusivity** – for certain tasks it may not be possible to run them in parallel on the same host since they may interfere with each other. To prevent this, tasks can be *exclusive* or *context-exclusive*. If a task is exclusive, no other task can run on the same host at the same time. If a task is context-exclusive, only tasks from the same context can run on the same host at the same time. Exclusivity and context-exclusivity are used by tasks which are important for the results of the benchmarking to prevent other tasks distorting the results.
- **Failure policy** – a task can be automatically restarted by the Host Runtime after a crash. Maximum running time of a task can be defined. If the task does not finish before the timeout ends, it will be automatically stopped.
- **Detailed load collection** – a task can request detailed load monitoring during its execution. Detailed load is typically requested by the benchmarking tasks to provide more data about the performance of the benchmark. Data collected during detailed mode are uploaded to the Results Repository alongside regular results produced by the benchmark.

3.3 Task Status

Each task goes through several states during its life cycle. Following task states are defined:

- **Submitted** – the task has been submitted to the Task Manager. It is not running yet.
- **Scheduled** – the task has been submitted to the Task Manager. Its dependencies are satisfied and it can be started now.

- **Running** – the task is running on a host.
- **Finished** – the task has terminated. Both successful and unsuccessful termination are represented by this state.
- **Aborted** – the task has been killed either by the user from the web interface or by some BEEN component.

4 Host Runtime

The Host Runtime is a BEEN component which must be run on every host participating in the system. It is responsible for running and management of all tasks on the host and serves as a proxy for all communication between BEEN components and the tasks running on the host. Integral part of the Host Runtime is the *Load Monitor*, which monitors utilization of various resources on the host.

For instructions how to run the Host Runtime, see section *Running Host Runtime (Part II, 2.2)*.

4.1 Host Runtime Lifecycle

The Host Runtime cooperates closely with the Task Manager. After the start, the Host Runtime registers on the Task Manager that is running the host specified by the command-line parameter. After the Host Runtime is started, it waits for instructions – which mostly contain requests for executing a task.

The Host Runtime can be stopped by explicitly killing its process by the user. The shutdown procedure kills all running tasks executed by the Host Runtime and unregisters the Host Runtime from the Task Manager.

All important events during the execution of the Host Runtime are written to the standard or error output. These messages are not stored centrally.

4.2 Running Tasks

To run a task on the Host Runtime, the Task Manager connects to the remote interface of the Host Runtime via RMI and sends a request that contains *task descriptor* of the task to run. Task descriptor contains all data needed by the Host Runtime to execute a task. For more details about the data stored in the task descriptor see *Task Descriptor (Part I, 3.2)*.

The Host Runtime then downloads the task package from the Software Repository, extracts the task's code, prepares task directories and attempts to start new process for the task. All potential errors are reported back to the Task Manager.

Packages downloaded from the Software Repository are cached by the Host Runtime. Several core packages (namely packages of the BEEN services and the detector task) are not downloaded from the Software Repository, they are stored permanently on the Host Runtime.

Each task runs in its own Java Virtual Machine. This prevents faulty task from crashing the Host Runtime. All task's communication with other BEEN components is routed through a *task's port* – RMI interface available to every task.

The task can use the task's port to send log messages to the Host Runtime which will then forward messages to the Task Manager. The Task Manager stores messages for later review by the user. The Host Runtime also captures the standard and error output of the task and forwards them to the Task Manager while saving them into local files to allow manual inspection by the user.

4.2.1 Timeouts and Restarts

Various limitations can be placed on the task's execution time and restart policy. User can limit total running time of the task. When the timeout expires, the task is killed by the Host Runtime and its execution is considered unsuccessful. The task execution is also considered unsuccessful if it exists normally, but its process return value is non-zero.

Crashing tasks are automatically restarted by the Host Runtime. Maximum number of restarts of the task can be set in its task descriptor. If the task is restarted more times than allowed, it is considered as unsuccessful. Both execution time limit and restart count limit are handled transparently by the Host Runtime and only the result of the last task execution is reported to the Task Manager.

Execution time limit and restart count limit allow for greater stability of the BEEN as a whole since the tasks that are hung do not block execution of the rest of the benchmark and crashing tasks have higher chance of succeeding.

4.3 Exclusive and Context-Exclusive Tasks

The tasks can be run in three modes: *non-exclusive*, *exclusive* and *context-exclusive*. These modes implement simple policies which control parallel execution of tasks on a host.

In non-exclusive mode, the task can be executed in parallel with other tasks on the same Host Runtime without any regard to possible interference between the tasks. Logging messages, standard and error output of the task are sent immediately to the Task Manager.

In exclusive mode the task is guaranteed to be the only task running on the specified Host Runtime. This mode allows the benchmarking tasks to execute without any interference that could potentially distort the benchmarking results. Logging messages and standard/error output of the task executed in exclusive mode are buffered by the Host Runtime and sent to the Task Manager after the task is terminated. This prevents distortion of the results of the tasks that monitor or are sensitive to the network traffic.

The context-exclusive mode is similar to the exclusive mode, except that it allows parallel execution of other tasks from the same context as a context-exclusive task on the same Host Runtime. This may prove useful when designing benchmarks with complex interactions between different tasks.

4.4 Load Monitor

To provide user with the details about utilization of hardware resources on the host, *Load Monitor* collects data from the underlying operating system. Load Monitor starts automatically with the Host Runtime and monitors usage of the following resources:

- free memory
- process count
- process queue length
- processor utilization of each processor
- utilization of all disk drives (bytes read and written per second)
- utilization of network interfaces (bytes received and sent per second)

Load Monitor can run in two modes – *brief mode* and *detailed mode*. The brief mode is designed to provide overview of the host activity while the detailed mode provides more

precise data that can be stored with the results of the benchmarking tasks. Sampling rate of the Load Monitor in brief mode can be set for all hosts in the environment in the Host Manager's configuration (see *Configuration Module (Part II, 3.7)*). Detailed mode is enabled only on request of the task. Each task can set its own sampling interval independently.

4.5 Data Directory Structure

The Host Runtime stores its data in `data/hostruntime` directory in the BEEN installation directory. Several subdirectories are present in this directory:

- **tasks** – contains directories with task and context data
- **cache** – cache of task packages downloaded from the Software Repository
- **boot** – contains core BEEN packages (namely packages of the BEEN services and the detector task)
- **native-lib** – contains native libraries used by the Load Monitor for supported platforms
- **load** – contains computer utilization data obtained by the Load Monitor in detailed mode

4.5.1 Tasks Directory

The tasks directory contains all data required and generated by the tasks that are running on the Host Runtime. For each task that is to be executed on the Host Runtime, the directory with name `context-id/task-id` is created (where `context-id` and `task-id` are the context ID and task ID of the task being executed). This directory is called the *task base directory*.

In the task base directory, several files and directories are created:

Files created:

- `standard-output` – text file which stores standard output of the task
- `error-output` – text file which stores error output of the task

Directories created:

- `task` – contains the unpacked contents of the `files` directory from the task's package. This directory is deleted after the task is finished
- `working` – stores data created by the task which must be accessible after the task is finished; it is deleted when closing parent context of the task
- `temporary` – stores temporary data created by the task; it is deleted after the task is finished
- `service` – used by the Host Runtime when unpacking the task's package

5 Task Manager

The Task Manager is a BEEN component responsible for scheduling execution of tasks, communication and synchronization of the tasks via checkpoints, and managing information about the tasks. The Task Manager itself is not a task, and only one instance has to be run in the BEEN environment.

For instructions on how to run the Task Manager, see section *Running Task Manager (Part II, 2.1)*.

To run a task, a task descriptor representing the task must be submitted to the Task Manager. Task will be started when all prerequisites specified in the task descriptor are satisfied.

The Task Manager performs following functions:

- **Scheduling of tasks** – after submitting a task, the Task Manager checks if the task dependencies are satisfied (i.e. required checkpoints are set). When the task is exclusive or context-exclusive, the Task Manager will also ensure the exclusivity limitations are satisfied. If the task cannot be run immediately, the Task Manager will store task descriptor and attempts execution of the task when conditions in the environment change (e.g. new checkpoint is reached or another task finished).
- **Host selection** – tasks must specify hosts where they will run. Hosts can be specified in two ways:
 - **List of hostnames** – list with names of hosts on which task can be run can be specified in the task descriptor. The Task Manager will select one host. The host selection algorithm is not specified and may be nondeterministic.
 - **RSL** – a task can specify a host suitable for its execution via an RSL condition. Condition is evaluated by the Host Manager and the Task Manager will select one of the hosts returned.

For more information about RSL, see section *Restriction Specification Language (RSL) (Part I, 11)*.

- **Log Storage** – the Task Manager contains a sub-component which stores logs and standard and error output generated by tasks. The logs can be inspected from the web interface and may be uploaded to the Results Repository with the results of a benchmark.

The logs are stored as text files in the `data/taskmanager/logs` directory. The directory contains subdirectories for each context. The directory of a context contains one subdirectory for each task belonging to the context.

Several files are stored in the task's directory:

- `task.log` – stores log messages from the task. Each message contains information about the log level and timestamp. New log messages are appended at the end of the file
- `stdout.txt` – stores the task's standard output
- `stderr.txt` – stores the task's error output
- `hostname.txt` – this file contains only one line on which the name of the host on which task was running is stored

Each log message carries information about its importance – the *log level*. Following table lists log levels defined in the BEEN (in decreasing order of importance):

- **FATAL** – a critical error occurred. Task is unable to run, this message is always followed by the termination of the task
- **ERROR** – a non-critical error occurred. Task was able to recover from the error and may continue to run
- **WARNING** – a warning about a potential problem.
- **INFO** – information about normal operation of the task. The message should provide more information about what the task is currently doing (e.g. “Package was downloaded.”).
- **DEBUG** – debugging information useful during the development of the task, a benchmark plugin, or other BEEN component. The message is usually not useful for the normal user since it is intended for the developer of the task
- **TRACE** – a very detailed debugging message, used when there is a need to track the execution of the task's code precisely
- **Host Runtime Management** – the Task Manager maintains a list of Host Runtimes running in the BEEN environment. Each Host Runtime registers and unregisters itself with the Task Manager when it is starting up or shutting down. Other components can be informed of Host Runtime registration. For example, the Host Manager uses this feature to automatically scan newly connected host
- **Naming Service** – maintains list of running services in BEEN environment. The Task Manager provides means to search for a specific service in this list to other components of BEEN
- **Task Information** – the Task Manager stores information about existing contexts and tasks, their status, checkpoints reached so far, etc.

For more information about the features provided by the Task Manager see the Javadoc documentation for the `TaskManagerInterface` interface.

5.1 Configuration

The Task Manager stores configuration settings for the each Host Runtime participating in the BEEN environment. Changes in the configuration are automatically distributed to all Host Runtimes currently registered with the Task Manager. The configuration can be changed via the web interface – see section *Configuration Module (Part II, 3.7)*.

Available configuration options include:

- **Package cache size limit** – the size limit of the Host Runtime's package cache, in bytes
- **Number of closed contexts kept** – number of closed contexts for which the Host Runtime should keep data on the disk. Host Runtime will automatically delete data for old contexts

The configuration settings are stored in the `data/taskmanager/configuration.xml` file.

6 Software Repository

The Software Repository is a service for storage and retrieval of all software run by the execution framework. It stores the software binaries for different platforms, as well as the static data and the sources. By using the Software Repository, the execution framework avoids relying on a distributed file system, which might be difficult to set up in a heterogeneous environment. When executing benchmarks, the presence of a distributed file system could also distort the benchmark results.

6.1 Packages

The basic storage unit in the Software Repository is a package. There are four types of packages:

- **Source package** – contains software source code, such as source of an application to be used for benchmarking.
- **Binary package** – contains compiled software.
- **Task package** – contains task code and data required by the execution framework when executing a task
- **Data package** – contains any static data files, such as an initial database snapshot for a database benchmark.

6.1.1 Package Structure

The package is in fact a ZIP archive. It contains the `files` directory with data files of the package and `metadata.xml` file, which contains description of the package contents – package metadata. Task packages also contain `config.xml` file with description of the task for the Host Runtime. The `metadata.xml` file is described below, the `config.xml` file is described in section *Packaging (Part III, 2.1)*.

6.1.2 Metadata Format Description

Package metadata consists of attributes, which have a name and a value. The list of possible attributes is fixed and it is specified which attributes are allowed and required for each package type. Every attribute has a type (Java class) – one of `String`, `Date`, `Version`, `PackageType` and `List<String>`.

In runtime, package metadata is represented by the `PackageMetadata` class. In the packages, the metadata is stored in the `metadata.xml` file. This is an XML file with root element `<package>`. Inside this element, there is one subelement for each attribute. The attribute value is stored as content between element's tags.

The metadata attributes are summarized in the following table. *Source/binary/task/data* flags indicate, whether given attribute is used in source, binary, task or data packages. Upper-case letter means it is required, lower-case letter means it is optional.

Name	Java class	Description	Source Binary Task Data	Notes
name	String	canonical name of the package	SBTD	⁷
version	Version	package version	SBTD	
hardwarePlatforms	List<String>	hardware platforms	sbtD	⁸
softwarePlatforms	List<String>	software platforms	sbtD	⁹
type	PackageType	determines whether package is source or binary	SBTD	¹⁰
humanName	String	human-readable package name	SBTD	¹¹
downloadURL	String	source of this package, if downloaded from some external site	sbtD	¹²
downloadDate	Date	download date and time, if downloaded from some external site	sbtD	¹³
sourcePackageFilename	String	filename of the source package in the software repository	bt	¹⁴
binaryIdentifier	String	BID of the binary in experiment, which this package is used in	b	
buildConfiguration	String	name of the build configuration, in which this package was built	b	

6.1.3 Package Names

The package name will be chosen by the Software Repository after the package is uploaded (if the package didn't already exist in the Software Repository on startup). This name is guaranteed to be unique. It has following format:

name.hardwarePlatforms-softwarePlatforms-suffix.hash.bpk

Most parts of the name are created from the package metadata.

- *hardwarePlatforms* and *softwarePlatforms* are made up as a concatenation of specified platforms with plus signs ("+") between them (example: win32+linux)
- *suffix* is:
 - `src` if the package is a source package

- ⁷ Package name consists of parts separated by hyphens ("-"). Every part can contain only lower-case letters or digits. Written as Perl-like regular expression, it must match `[a-z0-9]+(-[a-z0-9]+)*`
- ⁸ Contains zero or more `hardwarePlatform` elements, whose textual content denotes platforms. Platform should be written in lower-case letters. Some predefined values: `i386`, `sparc`.
- ⁹ Contains zero or more `softwarePlatform` elements, whose textual content denotes platforms. Platform should be written in lower-case letters. Some predefined values: `win32`, `linux`
- ¹⁰ Allowed values are: `source`, `binary`, `task`, `data`.
- ¹¹ This name is intended to be displayed in UI.
- ¹² If not specified, package can't be downloaded from external site or this site is unknown.
- ¹³ This attribute is in format specified by RFC 1123, section 5.2.14. It looks like this: Sun, 06 Nov 1994 08:49:37 GMT. This representation was chosen because it is current IETF standard, used in protocols like HTTP, and Java can deal with it easily using its `Date` class. If not specified, package can't be downloaded from external site or this site is unknown or download date and time is unknown.
- ¹⁴ If not specified, package wasn't compiled from other package stored in software repository or the source package is unknown.

- `bin` if the package is a binary package
- `task` if the package is a task package
- `data` if the package is a data package
- *hash* is a globally (across all packages ever stored) unique number chosen by the Software Repository.

Extension `.bpk` (Been PacKage) was chosen to differentiate packages from other files.

6.2 Operations Supported

The Software Repository supports following operations with packages:

- upload of a new package
- query packages by their metadata via a callback interface, including querying using the RSL query language – see section *Restriction Specification Language (RSL) (Part I, II)*.
- download of specific packages
- deletion of a package

Notably, instead of modifying a package, a new version of the package has to be created. This restriction helps to maintain coherency of package caches used by the Host Runtimes.

Package queries allow clients to search package metadata using arbitrary criteria. The client supplies a class with callback method (this class must implement `PackageQueryCallbackInterface` interface), whose code is executed by the Software Repository to determine list of matching packages. An implementation of the callback interface with support for RSL queries is provided.

6.3 Package Storage

Packages are stored in the Software Repository working directory. No additional files are required by the Software Repository, except `counter.dat` file, which stores value of the hash appended to name of the next package uploaded.

On startup, the Software Repository looks into the `data/softwarerepository` directory in the BEEN installation directory and copies any files it finds there to its working directory. It searches its working directory for `.bpk` files then and attempts to add them as packages. Unrecognized `.bpk` files are ignored.

6.4 Optimizations

The Software Repository is designed for transfer of large packages and for a fast package lookup. The transfer optimizations include asynchronous communication and a special interface provided for monitoring of the transfer progress. To improve the query performance, package metadata are cached (the cache is created when the Software Repository starts).

7 Managing Computers on the Network

The Host Manager is responsible for maintaining an up-to-date list of all hosts in the benchmarking environment and maintains database which stores data about hardware and software configuration of each host.

The Host Manager is also responsible for monitoring availability and utilization of hosts in the environment. It provides tools to generate basic statistics about the utilization of a specific host over time. For more details about monitoring facilities of the Host Manager see section *Host Utilization Monitoring (Part I, 8.2)*.

7.1 Host Database

The Host Manager stores all details about hosts in the *host database*. Host database contains list of hosts in the benchmarking environment as well as a hardware and software configuration of each host.

Host Manager automatically updates hardware and software configuration of the host when host connects to the benchmarking environment. Each update of the host configuration data adds new entry to the configuration history of the host. This allows for easy review of the hardware and software changes and provides means to relate configuration changes to the potential changes in the benchmark results. Configuration of any host can also be refreshed on user's request at any time.

Host Manager uses special tasks, *detectors*, to collect data about host. Since Java does not provide direct access to the underlying operating system, which is required by the detector tasks, native libraries that collect data are provided. These libraries have to be written for each platform separately. Currently Windows and Linux platforms are fully supported. On platforms where native library is not available only limited data about system is collected using facilities provided by Java.

7.2 Host Data

Data about each host in host database is organized into a tree-like structure. Inner nodes of the tree are called *objects*, leaves are called *properties*. Structure of the tree is based on the way various hardware and software components relate to each other with each child node adding more details about the parent. For example, object which represents on hard-drive can have several child objects representing different partitions present of the drive.

Each node in the tree has its own name which does not need to be unique within the tree. Name of an object consists of two parts: *object's type name* and *index*. Type name of an object should provide a hint about the data that the object contain. For example, type name of the object which stores details about memory is `memory`. Index of an object is non-negative integer which identifies objects with the same type name within their parent object. For example, if the disk drive contains two partitions, first one will have index of zero, second one will have index of one. Name of an object is then written as `typename(index)` (e.g. `Partition(2)`).

Name of a property contains only type name (indexed properties are not allowed) and has to

be unique within its parent object.

All objects and properties in the tree can be uniquely identified by their *full name*. Full name of the object consists of names of all objects that are on the path from the root node to the object. Names of objects on the path are separated by the period character. Full name of a property consists of full name of its parent object and type name of the property separated by the period character. For example, full name of the object representing second partition of the first drive is `drive(0).partition(1)` and full name of the property which represents size of that partition is `drive(0).partition(1).size`.

Most of the properties and objects in the database contain data collected by the detector. However, several properties that reflect current status of the host and are therefore not available to the detector (for example group membership property) are added by the Host Manager. Since various components of the BEEN may require storage of custom data for specific host, Host manager provides an API which allows storage and manipulation of user-defined properties and objects in the database.

All data about the host is stored in the properties. Object by themselves provide only the structure of the tree. Each property has to have a value. Value of the property can be one of the following types:

- `ValueBoolean`
 - Represents boolean value (`true` or `false`).
- `ValueInteger`
 - Represents integer equivalent to the `long` data type in Java.
- `ValueDouble`
 - Represents floating-point number equivalent to the `double` data type in Java.
- `ValueString`
 - Represents string.
- `ValueVersion`
 - Represents version of an application. Version consists of three version numbers (major, minor and release) and a build identification string.
- `ValueRange<T>`
- Represents interval of values of basic type (`ValueBoolean`, `ValueInteger`, `ValueDouble`, `ValueString`, `ValueVersion`).
 - Interval can be closed, open or half-open and can have infinite endpoint on either side.
 - Both endpoints have to be of the same type.
- `ValueList<T>`
- Represents ordered list of values of basic type (`ValueBoolean`, `ValueInteger`, `ValueDouble`, `ValueString`, `ValueVersion`).
 - All elements in the list have to be of the same type.

For a complete list of all objects, properties and their types see *Objects and Properties of the Host (Part IV, Appendix A)*.

7.3 Database Queries

Host Manager provides two different methods of searching the host database: *RSL queries* and *custom query interface*.

RSL queries provide means to specify conditions using logical expressions which may contain properties, objects and constants. For more elaborate description of the RSL and examples see section *Restriction Specification Language (RSL) (Part I, 11)*. RSL queries are mostly suitable for selecting hosts that belong to a specific group (see below) or when selecting hosts for specific task. RSL queries are directly supported by the Web Interface and therefore do not require user to write code.

For cases of more complex queries that cannot be expressed in terms of properties or objects from the database the Host Manager provides means to use custom class implementing query interface. To use this facility user is required to implement class derived from the `HostQueryCallbackInterface`, which contains only one method `match`. The `match` method rejects hosts that do not meet conditions. All data about one host is accessible from within this method and since it is written in Java, it can use all facilities provided by the language (for example RMI queries to various components of the BEEN). Host Manager calls `match` method on each host in the database and compiles list of all hosts that pass the test.

7.4 Host Groups

The Host Manager allows user to assign hosts from the environment to one or more *host groups*. Host groups are provided to simplify orientation within larger benchmarking environments.

Host group is a named set of hosts from the benchmarking environment. Host Manager provides one *default group* which always contains all hosts in the environment. The default group is automatically maintained by the Host Manager and always reflects current state of the database.

Each host will be member of at least one group (the default group).

All groups except the default group are managed by the user. Host Manager does not enforce relations between hosts in the group in any way and therefore groups are not automatically updated to reflect changes in hardware and software configuration.

Groups in the database are identified by their name which has to be unique. Each group can also contain textual description of the group and a metadata. Description is simple string which should describe purpose and contents of the group. It is not used by the Host Manager and it is provided only for user's convenience. Metadata are provided for other components of the BEEN as a storage for custom data about the group. The metadata cannot be edited by user.

7.5 Software Aliases

Software aliases are designed to simplify orientation in the list of software packages installed on the host. Typically, applications or packages may have slightly different names across various operating systems and it may be difficult and error-prone to write database queries that match all supported versions of the application. Each software alias acts as a common representative for given application across different versions of an application or operating system. Since software aliases are stored in database and have own set of objects and properties, database queries that work with aliases can be created.

Each alias has its own name (not necessarily unique) and contains data about application it represents (name, version and vendor). Aliases are automatically generated by the database when host is added to the database or when host's data in the database is updated.

For each software alias there is an *alias definition* that contains all data needed for the database

engine to create representative aliases. Each alias definition contains two restrictions (written in RSL query language) and four fields that define values of the properties of the resulting alias. First restriction in the alias definition specifies conditions on the operating system. In this restriction, all properties and child objects of the `os` object can be used. Second restriction specifies conditions on application resulting alias will represent. All properties of the application object can be used in this restriction.

Properties of the resulting alias can be specified either as constant values or they may contain references to the properties of the application that matched conditions specified via restrictions. Property references are specified with syntax similar to the syntax of the property names in Ant. That is, `${name}`, `${vendor}` and `${version}` variables can be used in any of the four fields that define properties of the resulting alias. Multiple variables can be used in one field and database engine will substitute all with the actual values of properties from the matched application.

For example, following alias definition

```
Alias name: Visual Studio (${version})
Result name: ${name}
Result vendor: Microsoft
Result version: ${version}
OS restriction: family == "Windows"
Application restriction: name=~/*Visual Studio.*/i
```

will result in alias

```
Alias name: Visual Studio (8.0.50727.42)
Application name: Microsoft Visual Studio 2005 Professional Edition
- ENU
Application vendor: Microsoft
Application version: 8.0.50727.42
```

when matched with the following application

```
Name: Microsoft Visual Studio 2005 Professional Edition - ENU
Vendor: Microsoft Corporation
Version: 8.0.50727.42
```

7.6 Host Manager Configuration

Several aspects of the Host Manager's operation can be configured by the user. Most of the settings are automatically propagated to all hosts in the environment.

Following options can be configured:

- **Host detection timeout** – specifies total amount of time the Host Manager will wait for the data from the detector scheduled on the host. If the detector does not upload data within specified interval, it is assumed that the detection failed. Host Manager will ignore all data received from the host after the timeout expired. Timeout is measured in milliseconds (but not necessarily with the millisecond precision), minimum value is therefore 1 ms, maximum value is not restricted.
- **Host detection timeout check interval** – specifies precision of the host detection timeout. Host Manager will check if the data have been received only once in a specified interval. Interval is measured in milliseconds. Minimum value is 1 ms, maximum is not restricted. Note that lower value means that the checks are done more often which can impact performance of the Host Manager.
- **Activity monitor refresh interval** – specifies interval between consecutive updates of

the status data for hosts registered with the Host Manager. Host Manager checks if some of the hosts on the network did not crash in specified interval. Interval is measured in milliseconds. Minimum value is 1 ms, maximum is not restricted. Lower interval means that more up-to-date data about the host status will be available, but again, too low interval may negatively impact performance of the Host Manager.

- **Host crash timeout** – specifies total amount of time that the Host Manager will wait before marking host as crashed. Timeout for given host is reset every time the Host Manager receives data from the Load Monitor running on the host. Timeout is measured in milliseconds. Value of the timeout has to be greater than the sampling interval of the Load Monitor in brief mode (otherwise host will be always marked as crashed). Maximum value is not restricted.
- **Brief mode sampling interval** – specifies sampling interval for the Load Monitor in brief mode. This interval is set globally and if the setting is changed, new value is automatically propagated to all connected hosts. Interval is specified in milliseconds, minimum is 1 ms. Note that setting sampling interval to small values can severely degrade performance of the whole environment, since each host will send data to the Host Manager at given rate.
- **Default detailed mode sampling interval** – specifies default sampling interval of the Load Monitor for tasks that request detailed load and do not specify custom value. Value is set globally and change of the value is automatically propagated to all connected hosts. Note that change of this value only affects tasks that start after the change is made. Value is measured in milliseconds, minimum value is therefore 1 ms. Maximum is not restricted.

8 Monitoring Computer Utilization

Since BEEN runs in distributed environment it is important to provide user with feedback about progress of various processes running in the environment. BEEN provides two independent views on the host utilization. First, higher-level view shows details about the tasks that are currently running in the environment. Second, lower-level view provides user with the details about utilization of hardware resources on hosts in the environment. Both views are accessible from the web interface (see sections *Tasks Module (Part II, 3.4)* and *Working with Hosts (Part II, 3.3.1)*).

8.1 Monitoring Running Tasks

The Task Manager maintains information about all contexts and tasks that have been submitted by various components of the BEEN.

For context it stores context name, context ID, description and reference to the experiment associated with the context. For each task in each context the Task Manager stores details about the task package (e.g. name, description), current state of the task and checkpoints reached by the task.

The Task Manager keeps data about tasks and contexts in its storage indefinitely and user has to clear the data manually.

8.2 Host Utilization Monitoring

The utilization data on each host is collected by the Load Monitor which is a part of the Host Runtime. The data from all Load Monitors running in the environment is collected and stored centrally by the Load Server. The Load Server is a part of the Host Manager

The Load Server maintains list of all hosts in the environment and automatically determines status of the host from the data received from the Load Monitor. Each host can be either *offline*, *online* or *crashed*. Since the Load Server stores all data received from the Load Monitors it provides complete record of the utilization of the host when the host was on-line.

To collect utilization data the Load Monitor requires low-level access to the underlying operating system. Since this is not possible directly from Java, native libraries that collect data from the operating system have to be written for each supported platform. Currently Load Monitors fully support Windows and Linux operating systems.

The Load Monitor starts automatically with the Host Runtime and starts collecting data as soon as the Host Manager updates host's configuration in the database. Utilization data is automatically sent to the Load Server which stores all data in the host database. Load Monitors collect data about processor usage, disk reads and writes, incoming and outgoing network traffic, process count and thread scheduler queue length. Based on the data collected Load Server is able to calculate simple statistics about the utilization of the host.

Load Monitors send data to the Load Server in the form of events. The Load Server contains simple event loop which processes data only when needed. This architecture provides for great extensibility of the Load Server since it allows registration of custom event listeners which can process data. Load Monitors generates events of several types – for example events which notify the Load Server about hardware configuration change, Host Runtime start-up or shut-down, or change of the monitoring mode.

Load Monitors collect data in two modes: *brief mode* and *detailed mode*. In brief mode Load Monitor takes samples with relatively long sampling interval and immediately sends all data to the Load Server. This mode is enabled by default since it provides enough data for detection of the crashed hosts and does not stress network by sending large amounts of data to the Load Server.

Load Monitor is switched to the detailed mode only on when requested by the running task. All data collected when monitor is in detailed mode is stored locally and can later be requested by the data collection tasks to provide more details about the host utilization during task's runtime. Load Monitor will select load samples in regular intervals and send them to the Load Server as a notification that the host is still running.

When the task that requested detailed mode ends, the Load Monitor will automatically switch back to the brief mode.

9 Benchmark Manager

The Benchmark Manager together with the Results Repository form the benchmarking framework which is build over the execution framework.

As its name suggests, the Benchmark Manager manages the execution of benchmarks. It enables the user to set up a benchmark, configure its parameters and execute it either manually or schedule its automatic execution in regular intervals. When the benchmark is running, the Benchmark Manager monitors its execution and reports progress and status information to user.

The Benchmark Manager is extensible. Is uses a system of *plugins* which handle all benchmark-specific tasks. The support for new benchmark can be added by writing a new plugin. The plugins can be managed by user from the web interface. For detailed information about plugin structure and responsibilities, see section *Extending Benchmark Manager (Part III, 3)*.

When executing the benchmarks, the Benchmark Manager cooperates closely with the Results Repository. In fact, it requires the Results Repository to be running for most of its operations (e.g. conducting a new benchmark).

9.1 Benchmark Entities

While the execution framework works with task as a basic operational unit, the benchmarking framework introduces a hierarchy of abstractions of processes in the environment – *analysis*, *experiment*, *binary* and *run*. The abstractions used in the Benchmark Manager reflect common use-cases of the benchmarking environment and can be conveniently represented with following hierarchical structure:

- analysis 1
 - experiment 1
 - binary 1
 - run 1
 - run 2
 - ...
 - run n
 - binary 2
 - ...
 - binary m
 - experiment 2
 - ...
 - experiment k
- analysis 2
- ...
- analysis l

We refer to these abstractions as benchmark *entities*. Each entity can be in one of the following

states: *configured*, *dispatched*, *submitted*, *running* and *finished*. Benchmark Manager also provides aggregated view of the processes that run on different levels of the hierarchy. For example, Benchmark Manager aggregates logs generated by all tasks that run in an experiment.

9.1.1 Experiment

An *experiment* is a basic unit of benchmarking. A typical experiment covers downloading of software sources, compilation, deployment, execution, measurement, statistical evaluation and visualization of the results.

9.1.2 Analysis

An *analysis* is a container for several benchmarking experiments. The benchmarking framework distinguishes between two types of analyses:

- A *Comparison analysis* contains experiments aimed to compare performance of different versions of the tested software, different software products with similar functionality (e.g. JBoss vs. Jonas application servers), or the impact of hardware configuration on performance. The experiments in the comparison analysis are created explicitly by the user and run immediately. The number of experiments is not limited.
- A *Regression analysis* serves to find performance regressions during the development of the tested software. After creating the analysis, user creates and configures a *model experiment* and an *execution schedule*. The Benchmark Manager will then use the model experiment as a template for creating “real” experiments automatically in regular intervals defined by the schedule. Generated experiments correspond to versions of the tested software.

9.1.3 Binary

Multiple compilations of the same software under the same configuration may produce binaries with different performance. This is caused by non-deterministic results of compilation, which may have noticeable impact on performance (e.g. because of different cache-miss rate). Software susceptible to the random effects caused by the compilation should therefore be benchmarked so that multiple compiled binaries are tested under the same circumstances in the context of one benchmarking experiment. The statistical analysis of the data produced by the experiment should take all of the binaries into account.

9.1.4 Run

Benchmark experiments are subjects to random effects during execution and measurement, resulting in slightly different performance results even on same hardware and software configuration and the same binary. Thus, several executions of the benchmark in succession are needed in a single benchmarking experiment with the same binary. We call a single execution and measurement of a benchmark a benchmarking *run*. The statistical analysis of the experiment should take all the runs into account.

9.2 Benchmark Plugins

Benchmark Manager is designed with extensibility in mind and therefore allows for adding

support for new benchmarking suites. Each benchmarking suite is defined in its own *benchmark plugin*. Benchmark Manager allows deployment of the new plugins at any time via the web user interface. Each plugin contains several key components – *Configurator*, *Generator* and *Version Provider*.

For information about creating new benchmark plugins, see section *Extending Benchmark Manager (Part III, 3)*.

9.2.1 Configurator

The configurator is a component of the plugin which provides user with the interface to the functionality provided by the plugin. Interface provided by the configurator seamlessly integrates into the web interface and does not require any additional libraries. User interacts with the configurator via a series of wizard-like screens which allow user to go back and forth between various settings before the experiment is created.

9.2.2 Generator

The generator is run by the Benchmark Manager and generates a sequence of tasks that are required for the experiment created by the user during configuration phase. The generator must support both types of the analyses supported by the BEEN (that is, it should be able to generate comparison experiments as well as regression experiments).

9.2.3 Version Provider

The version provider is used only during the regression analyses. It's role is to detect all versions of the software being benchmarked that had not been tested yet. It should be able to search repositories located outside of the BEEN environment. The Benchmark Manager runs the version provider in regular intervals according to the schedule specified during configuration phase. When new version of the software is detected, the Benchmark Manager will automatically call generator with new data to create new experiment based on the template specified by the user.

Since the user of the benchmark may need to test only specific versions of the software, the version provider allows user to specify which versions should be taken into account. For example, user may specify interval in history and only versions released during that interval will be accepted.

9.3 Experiment-level Scheduling and Monitoring

The Benchmark Manager is responsible for experiment-level scheduling and status monitoring. Scheduling performed by the Benchmark Manager is based on the information about experiments and analyses.

New experiments may be added by the users of BEEN or automatically by the Benchmark Manager during regression analyses. Following rules are taken into account by the Benchmark Manager when scheduling experiments:

1. Comparison experiments created by the user are started immediately.
2. Regression experiments created by version providers are processed from the most recent version of benchmarked software to the oldest one.

3. At any moment, only two regression experiments from a single analysis can be running. By following aforementioned rules the Benchmark Manager is able to guarantee that:

1. User will get results of interactively created experiment as soon as possible.
2. Experiments created by the Version Provider are processed from the most recent one – probably being the most important one – to the oldest one.
3. Regression experiments which, by definition, have to run benchmarking software on the same hosts every time, are not interfering with each other since only two will run simultaneously. This prevents collisions between various tasks requiring exclusive access to the specific host. When two benchmarks are run simultaneously we can take advantage of the fact that one experiment may be doing measurements while the other one is compiling software on a different host.

9.4 Experiment Lifecycle

Experiment execution involves several steps that are automatically performed by the Benchmark Manager in conjunction with other components of the BEEN. In following section we describe steps performed when executing comparison analysis and regression analysis.

Comparison analysis:

1. **Creation of a comparison analysis** requires the Benchmark Manager to store basic metadata for the analysis, and report them to the Results Repository. The Results Repository needs the metadata to be able to receive results of subsequently created experiments in the analysis.
2. When an **experiment is created** in the analysis, user selects benchmark plugin from the list of available plugins. Benchmark Manager activates Configurator of the selected plugin which guides user through the experiment creation.
3. Configurator collects all data required by the Generator and benchmarking framework. Configuration depends on the benchmark being run, but always contains host selection and number of runs and binaries.

Running a distributed benchmark requires multiple machines performing a specific activity (for example database server, clients). We refer to the action performed by the hosts as a *host role*. There are two basic types of roles:

- *compilation roles* – these roles represent a host not directly involved in the process of benchmarking, but required for activities related to the benchmarking.
- *benchmarking roles* – these roles represent a host which will be used directly for a benchmarking purposes.

Restriction on the hosts which belong to the specific role are described as RSL expressions. If there are multiple hosts matching expression for the compilation role, they are all valid candidates and the host that will perform action required by the role will be chosen automatically by the Task Manager. On the other hand, the benchmarking role must provide an exact number of hosts that is required for the role. Experiment will fail if the number of hosts that match restriction specified in the configuration is smaller than the number required by the role. The experiment enters *configured* state when the configuration is finished.

4. After the configuration of the experiment is finished, the Benchmark Manager activates the **generator from the selected plugin. Generator will process all data collected during the configuration phase and generate list of task to be run.** Each task is assigned to the specific level of an entity hierarchy. This allows for easy monitoring of

the progress of the experiments.

5. Once the generator finishes, the created experiment enters *dispatched* state. Just before the experiment is started, the Results Repository receives data about the hierarchical structure of the entities in the experiment. This data is later used to verify data generated by the experiment.
6. Dispatched experiments are periodically processed by the Benchmark's Manager scheduling algorithm and tasks ready to run are *submitted* to the Task Manager for execution. All tasks of an experiment run in a single context, i.e. Benchmark Manager's experiment entity corresponds to the Task Manager's context.
7. After submitting the tasks, the Benchmark Manager monitors status of the experiment and waits for experiment to finish. During this phase, status of the experiment is set to *running*.
8. After all tasks in the experiment terminate, status of the experiment is set to finished. By this time all data generated by the task running in the experiment are already uploaded in the Results Repository.

The regression analysis uses a slightly modified process since experiments are created automatically by the Benchmark Manager in conjunction with the version provider from the plugin selected by the user. Following steps are executed during the regression analysis:

1. Regression analysis is created in a way similar to the comparison analysis. All data about the analysis are reported to the Results Repository.
2. When creating regression analysis, user is required to specify model experiment and execution schedule of the version provider.
3. The model experiment is created in the same way as a regular comparison experiment. However, instead of starting the generator, Benchmark Manager stores the model experiment and executes the version provider with settings selected by the user.
4. Execution schedule defined by the user specification when the version provider is run. Every time the version provider is executed, it may create new experiment. All new experiments will use the same layout (same hosts and configuration) as the model experiment defined by the user.
5. Experiments created by the version provider are then processed in the same way as experiments created in the comparison analyses.

10 Results Repository

The Results Repository as a persistent archive of benchmark results and other data obtained when running the experiments (such as utilization of hosts participating on the experiment and log of run tasks). It automatically performs statistic calculations on the results and generates statistic tables and graphs of the results, which can be viewed through the web interface.

Statistical evaluation is performed using the R language¹⁵, a standard tool for statistical processing.

10.1 Results Collection

The tasks in the benchmark store their results on hosts where they run in a format specific for each benchmark. These results need to be checked for correctness, converted into a common format and uploaded to the Results Repository. This work is done by results collection tasks, which are scheduled at the end of each run, binary or experiment (as specified by the generator).

The results collection consists of two phases:

1. The benchmark results are checked for correctness and converted from a format used by the benchmark into a common text-file format. The task for checking and converting the results is specific for each benchmark.
2. Converted results, logs and host utilization information from participating hosts are uploaded to the Results Repository. This operation is generic for all benchmarks.

After the benchmark results are uploaded, the R scripts are executed. They typically invoke computation of various statistics and graphs of the results. For details, see section *Statistics Calculation (Part I, 10.4)*. The graphs for the computer utilization are generated on-demand, because of their large size and low probability that the user will want to see them.

10.2 Results Repository Database

The Results Repository stores all data in a hierarchical database. Each piece of data is associated with an analysis, experiment, binary or a run – we will call them *entities* in the text. We refer to an entity in a lower level in hierarchy than current entity as a *subentity* (i.e. experiment is subentity of analysis).

The entity is *complete* if all results for all its subentities have been uploaded to the Results Repository.

The results of the benchmarks are stored in the run level. Typically, the Results Repository computes a statistics from the data in the run, all runs in the binary etc. up to the analysis level. The graphs are generated only at the analysis and experiment levels. The database would grow to unmanageable size if they were generated at the lower levels too (even now, is not uncommon for the database to grow to the size of tens of GBs).

The load information is stored for all tasks which specified that they want to measure the detailed load when executed by the Task Manager.

¹⁵ <http://www.r-project.com/>

The logs of the tasks are stored separately from the entity hierarchy, as they are primarily tied to tasks, not entities. However, each entity carries information about tasks associated with the entity.

Each entity contains metadata, which are key-value pairs containing various information about the entity. Some of the metadata is used by the Results Repository, some are meant only as information for the user.

10.3 Failed Runs

The benchmark may fail from various reasons, e.g. when the benchmarked application crashes. This case is recognized in the results collection phase and runs which failed will be marked as *invalid*. The invalidity will propagate to the upper levels of the entity hierarchy (binary is marked invalid if all contains only invalid runs, and similarly for the upper levels).

The data of the failed runs is not processed by the R scripts (thus no statistics are computed and no graphs are generated). The user can download run results in raw format, for possible examination of the run failure.

10.4 Statistics Calculation

After the run data is uploaded to the Results Repository, R scripts can be called. Those scripts are called *R scripts callbacks*. The scripts are responsible for computing statistics and graph generation.

There are several types of callback scripts:

- **Complete run** – called after the run upload. It operates on a data of a given run only.
- **Incomplete binary** – called after the run upload. It operates on a data of the whole binary, even if it is still incomplete.
- **Complete binary** – called after uploading of all runs in the binary. It operates on a data of the whole binary, which is complete at the time of running.
- **Incomplete experiment** – called after the run upload. It operates on a data of the whole experiment, even if it is still incomplete.
- **Complete experiment** – called after uploading of all runs in the experiment. It operates on a data of the whole experiment, which is complete at the time of running.

Each benchmarking plugin comes with default setting of the called scripts. This setting can be overridden by the user when creating the experiment.

The entities in the database can be deleted by the user. This operation invalidates the statistics and graphs of all entities at the upper levels and the Results Repository needs to recalculate them. This is accomplished using the *invalidation scripts* mechanism. Anytime some script generates a statistics table or a graph, it can register the invalidation script on a processed entity. These scripts are then automatically called each time something is changed in contents of an entity and they can regenerate statistics tables and graphs. For more details, see section *Writing R Functions (Part III, 4.3)*.

10.5 Cooperation with Benchmark Manager

The Results Repository closely cooperates with Benchmark Manager in creating analyses, experiments and binaries. In experiment generation phase, information about analysis,

experiments and binaries created by the Benchmark Manager is stored in the Results Repository immediately. This implies that the Results Repository must be running at this time. Information provided by the Benchmark Manager contains number of binaries and runs to be executed and all metadata needed for results upload. This includes a list of hosts that will be uploading results and a list of tasks, from which Results Repository must collect logs for each entity. Each entity that is active in the Benchmark Manager must exist in the Results Repository.

Entities deleted in the Results Repository are deleted in the Benchmark Manager too. The entities which are active (i.e. some tasks associated with the entity are running) in the Benchmark Manager cannot be deleted from the Results Repository. If the Results Repository cannot contact the Benchmark Manager, deletion of analyses or incomplete entities is not possible.

10.6 Database Contents

The Results Repository stores following information:

- Registered R scripts and packages – see section *R Script Packages (Part III, 4.2)*.
- Logs from benchmarking tasks for reference during results browsing.
- Data for the all entities.

The entity data is stored in a hierarchical structure, where data for each entity (except runs) is stored in a separate directory (named by the entity ID), which contains directories for the subentities.

The benchmark results are stored in NetCDF¹⁶ format – a cross-platform format designed for storing scientific data.

10.6.1 Analysis Directory

The analysis directory contains:

- Metadata files describing the analysis (name, plugin, experiment count, generated graphs information, etc.).
- Text files with statistics tables (if generated).
- PNG files with graphs (if generated).
- Information about registered invalidation scripts (if generated).

10.6.2 Experiment Directory

The experiment directory contains:

- Metadata files describing the experiment (name, succesfull/complete binary count, generated graphs information, etc.)
- Table containing information about hosts involved in experiment and benchmarking roles executed on them.
- List of tasks associated with the experiment and information about log upload status.
- Text files with statistics tables (if generated).

¹⁶<http://www.unidata.ucar.edu/software/netcdf/>

- PNG files with graphs (if generated).
- Information about registered invalidation scripts (if generated).

10.6.3 Binary Directory

The binary directory contains:

- Metadata files describing the binary (name, successful run count, valid runs, etc.)
- Validity map, which contains index of all tables from all hosts from each run and validity information about each run. This index is used by Java, so run list can be displayed according to standard run identification. Internally only valid runs are numbered from 0 and invalid runs are stored separately. This is due to R computation and allows it to perform some needed operations much faster (i.e. random selections)
- List of tasks associated with the binary.
- Table with list of tasks for each run in the binary.
- Text files with statistics tables (if generated).
- PNG files with graphs (if generated).
- Information about registered invalidation scripts (if generated).
- Directory with uploaded load information for a binary.

10.6.4 Load Directory

The load directory contains one directory for each run, which contains:

- Table with information about load graphs (if generated).
- PNG files with load graphs (if generated).
- Directory for each host, from which load was uploaded for this run. This directory contains text files with load data from a host for a given run.

10.7 Results Repository Export Format

The Results Repository allows exporting of entity data in a format called “BEEN archive database format”. The exported file may contain data one or more analyses, experiments, binaries or runs.

The exported file is essentially a ZIP file, which contains entity data in the same format as the database itself, i.e. hierarchical structure of directories.

When exporting entities from the lower levels of the hierarchy, the directories for upper levels of the hierarchy will not be present in the file (i.e. there will be no directories for analyses and experiments, when the user will export only selected experiments in one analysis). For details about the Results Repository database format, see section *Database Contents (Part I, 10.6)*.

The directory structure with entity data is accompanied with `export.meta` file which contains information about the type of exported entities and their count.

11 Restriction Specification Language (RSL)

Restriction Specification Language (RSL) is a simple, but general language for selecting objects matching specified conditions from a set. In BEEN, it is used in several places:

- in the task descriptor
 - to select a package for a task from packages stored in the Software Repository
 - to select hosts where the task can run
- in experiment configuration, to specify condition on hosts in various roles
- in Host Manager's software alias definitions, to specify restrictions for the operating system and for the application

For generality, we will call queried objects *items* and their associated metadata *properties*.

11.1 Language Description

We introduce the RSL syntax and semantics in the informal way. For the formal grammar of RSL, see *RSL Grammar (Part IV, Appendix C)*.

11.1.1 Conditions

Basic unit of the RSL is a simple condition. It has a form: *propertyPath operator value*.

Example:

```
hostname == "localhost"
processor.speed > 500
java.version >= 1.5.0
```

Simple condition restricts the query result to those items, whose properties match given conditions. Available operators and allowed property and value types are described below.

Simple conditions can be combined using operators `&&` and `||`, like boolean expressions in Java. Parenthesis can be used to denote precedence.

Example:

```
(name == "foo" || name == "bar") && version >= 3.0
```

In case of hosts, properties have hierarchical structure and more items of same name exist on one level – the property is in fact array (for example, a host usually has many applications installed). Sometimes, it is necessary to specify more restrictions on concrete property in this array. This is done using qualified condition.

Example:

```
application { name == "ssh" && version >= 2.0 }
```

The example could be read as “select those items, which has at least one property ‘application’, which has name equal to ‘ssh’ and version greater or equal to 2.0”.

Note the difference against:

```
application.name == "ssh" && application.version >= 2.0
```


This example means “select those items, which has at least one property ‘application’, which has name equal to ‘ssh’, and at least one property ‘application’ with version greater or equal to 2.0”.

11.1.2 Types

Properties can be of following types:

Long

This type can be written also as a literal on the right-hand side of the operator using usual syntax – sequence of digits – with optional unit appended. Following units are supported:

Unit	Size
b	2^0
k, kB	2^{10}
M, MB	2^{20}
G, GB	2^{30}
T, TB	2^{40}
P, PB	2^{50}

All units are case-sensitive and must be appended right after the last digit (no whitespace or other separator is allowed).

Examples:

```
1
1kB
100M
55TB
```

Version

This type can be written also as a literal on the right-hand side of the operator. Versions are composed from at least two parts, separated by dots (“.”). Those parts can contain letters, digits, dash (“-”) and underscore (“_”). First part of the version must begin with digit.

Examples:

```
1.0
100.100.100
1.2beta5
3.0_R3-3
```

Date

This type can be written also as a literal on the right-hand side of the operator. Used syntax is taken from ISO 8601. Allowed formats are:

Year and month:

YYYY-MM (e.g. 1997-07)

Complete date:

YYYY-MM-DD (e.g. 1997-07-16)

Complete date plus hours and minutes:

YYYY-MM-DDThh:mmTZD (e.g. 1997-07-16T19:20+01:00)

Complete date plus hours, minutes and seconds:

YYYY-MM-DDThh:mm:ssTZD (e.g. 1997-07-16T19:20:30+01:00)

Complete date plus hours, minutes, seconds and a decimal fraction of a second:

YYYY-MM-DDThh:mm:ss.sTZD (e.g. 1997-07-16T19:20:30.45+01:00)

String

This type can be written also as a literal on the right-hand side of the operator using usual syntax – enclosing in quotes. Quote character in the string can be escaped using `\`.

PackageType

This type can be written also as a literal on the right-hand side of the operator using reserved words `source`, `binary`, `task` and `data`.

Example:

```
type == source
type != data
```

List (of strings)

This type can't be written as a literal on the right-hand side of the operator.

11.1.3 Operators

Equality Operators (==, !=)

This operator can be applied to properties of types `Long`, `Version`, `Date`, `String` and `PackageType`. Values on the right-hand side of the operator must be of the same type as the restricted property.

Comparison Operators (>, >=, <, <=)

This operator can be applied to properties of types `Long`, `Version`, `Date` and `String`. Values on the right-hand side of the operator must be of the same type as the restricted property.

Containment Operator (contains)

This operator can be applied to properties of type `List`. Value on the right-hand side of the operator must be `String`.

Example:

```
hardwarePlatforms contains "linux"
```

Regular Expression Operators (=~, !~)

This operator can be applied to properties of type `String`. Value on the right side of the

operator must be a Java regular expression, enclosed in slashes ("/"). Slash characters in the regular expression can be escaped using \/. Optional flag "i" can be added at the end of the regular expression to denote case-insensitivity.

The =~ operator means "matches", the !~ operator means "does not match".

Examples:

```
name =~ /mff\.cuni\.cz/  
name =~ /MFF\.CUNI\.CZ/i  
name !~ /\.com$/
```

12 Related projects

Among the related projects are tools for automated performance monitoring during software development, generic tools for automated distributed testing and generic tools for automated distributed benchmarking.

The tools for automated performance monitoring during software development include TAO Performance Scoreboard¹⁷, A Real-Time Java Benchmarking Framework, Lockheed Martin ATL Benchmarking Tools and Mono Regression Benchmarking Project¹⁸. These tools were all created for use in a particular software project. Porting the tools for use in other software projects would require a significant additional effort.

The Skoll Project started as a tool for distributed software testing that used computing resources provided by outside volunteers. One of many challenges of the project is finding a minimal set of tested software configurations that would still discover potential problems in any configuration. Currently, the project also covers regression benchmarking, focusing on finding benchmarks and configurations that are most sensible to performance problems present in any configuration. Such benchmarks and configurations are first found using the computing resources provided by outside volunteers, and then precisely evaluated on dedicated computers. Within this context, BEEN is a tool for the precise performance evaluation.

The CLIF Tool¹⁹ is a load-injection²⁰ framework targeted primarily at Java middleware. It covers deployment, monitoring and storing of results. The tool is capable of a highly configurable distributed load injection, emulating for example clients accessing a web site. BEEN does not aspire to provide the load injection support for general benchmarks, but is able to run benchmarks that use load injection, adding runtime monitoring, results repository and automated evaluation of results. The results repository of CLIF is limited in comparison.

The NIST Automated Benchmarking Toolset is a generic tool for automated benchmarking in a grid environment. The tool uses a common format for storing results in a relational database. It relies on the Distributed Queuing system as its execution environment and shell scripts as its task implementation language, therefore, the support for Windows platforms is limited. The tool is no longer being developed and the source code is not available.

¹⁷ <http://www.dre.vanderbilt.edu/stats/performance.shtml>

¹⁸ <http://dsrg.mff.cuni.cz/projects/mono/>

¹⁹ <http://clif.objectweb.org/>

²⁰ Load injection is artificial generation of load for benchmarking purposes.

Part II

Using BEEN

1 Installing BEEN

1.1 Requirements

1.1.1 Supported Hardware/Software Platforms

Theoretically, BEEN can be run on any hardware/software platform which supports Java 5.0. However, the hardware detector and the utilization monitor use native libraries, which are available only on several selected platforms. If you try to run BEEN on a platform for which the native libraries are not supplied, only basic detection and monitoring facilities will be available.

Also, we could not test BEEN on every platform available, so it is possible that some unforeseen issues could arise if you try to run BEEN on a platform which was not tested.

Tested and officially supported hardware/software platforms are:

- Windows XP on x86
- Fedora Core 5 Linux distribution on x86
- Gentoo 2006.1 Linux distribution on x86

Tested, but not officially supported hardware/software platforms are:

- Windows 2000, Windows .NET Server 2003 on x86

Results Repository component works only on Gentoo and Fedora Core, as the SJava library, which the Results Repository uses, is only available on Linux platform.

1.1.2 Required Software

For running Host Runtime:

- Java Runtime Environment 5.0

For running Results Repository:

- Java Development Kit 5.0
- R 2.1.x or 2.2.x

For compilation of BEEN:

- Java Development Kit 5.0
- Ant 1.6.x

For generating the Doxygen documentation of the C++ files:

- Doxygen 1.4.7 or higher

For running the web interface:

- Apache Tomcat 5.5.x

For using the web interface:

- Microsoft Internet Explorer 6.0

- Mozilla Firefox 2.0.x
- Opera 9.x

For compiling the detectors and Load Monitor on Windows:

- Visual Studio 2005

For compiling the detectors and Load Monitor on Linux:

- gcc 3.4.x

Notes: BEEN was tested only with Java environments from Sun. While it would probably work correctly with Java environments from other vendors, they are not supported.

Other versions of software than those noted above may work, but are not supported.

Benchmark plugins supplied with BEEN may have additional requirements on the hardware or the installed software; see sections *Benchmarking with Xampler (Part II, 4)* and *Benchmarking with RUBiS (Part II, 5)*.

1.2 Installing BEEN Execution Environment

1.2.1 Windows

1. Make sure the computer meets specified requirements.
2. Extract the `been.zip` package to target directory using Windows Explorer or other suitable application.

1.2.2 Linux

1. Make sure the computer meets specified requirements.
2. Extract the `been.tar.gz` package to target directory using a following command:

```
tar xzf been.tar.gz
```

1.3 Installing Results Repository Prerequisites

Note: On several places, the instructions require you to copy some files from the BEEN DVD. If you don't have the DVD available, you can also download the files from the BEEN web (<http://been.objectweb.org/>).

1.3.1 Fedora Core 5

1. Install the correct version of the R statistical package by downloading the package `R-2.1.0-1.bio.fc3.i586.rpm` from its URL²¹ and running following command as root:

```
yum install R-2.1.0-1.bio.fc3.i586.rpm
```

2. Install NetCDF support for R using packages `netcdf` and `udunits`:

```
yum install netcdf
```

²¹ <ftp://ftp.pbone.net/mirror/apt.bea.ki.se/biorpms/fedora/linux/3/i386/RPMS/biorpms/R-2.1.0-1.bio.fc3.i586.rpm>

```

yum install netcdf-devel
yum install udunits
yum install udunits-devel
ln -s /usr/lib/netcdf-3/libnetcdf.a
    /usr/lib/libnetcdf.a
ln -s /usr/include/netcdf-3/netcdf.h
    /usr/include/netcdf.h

```

3. Copy the RNetCDF R extension package (file RNetCDF-_1.1-3.tar.gz) from the Libraries directory on the BEEN DVD and in the directory with the package file run as root:

```
R CMD INSTALL RNetCDF_1.1-3.tar.gz
```

4. Copy the SJava package (file SJava_0.69-0_fixed.tar.gz) from the Libraries directory on the BEEN DVD and in the directory with the package file run as root:

```
R CMD INSTALL -c SJava_0.69-0_fixed.tar.gz
```

5. Create the R_HOME environment variable that points to the R installation directory:

```
export R_HOME=/usr/lib/R
```

6. Add SJava libraries to the LD_LIBRARY_PATH environment variable:

```

export LD_LIBRARY_PATH
    =${LD_LIBRARY_PATH}:${R_HOME}/lib
    :${R_HOME}/library/SJava/libs

```

7. You may want to add the R_HOME and LD_LIBRARY_PATH variable settings to your ~/.bashrc file.

1.3.2 Gentoo 2006.1

1. Install the correct version of the R statistical package using following command as root:

```
emerge =R-2.2.1
```

2. Install NetCDF support for R using packages netcdf and udunits:

```
emerge netcdf udunits
```

3. Copy the RNetCDF R extension package (file RNetCDF-_1.1-3.tar.gz) from the Libraries directory on the BEEN DVD and in the directory with the package file run as root:

```
R CMD INSTALL RNetCDF_1.1-3.tar.gz
```

4. Copy the SJava package (file SJava_0.69-0_fixed.tar.gz) from the Libraries directory on the BEEN DVD and in the directory with the package file run as root:

```
R CMD INSTALL -c SJava_0.69-0_fixed.tar.gz
```

5. Create the R_HOME environment variable that points to the R installation directory:

```
export R_HOME=/usr/lib/R
```

6. Add SJava libraries to the LD_LIBRARY_PATH environment variable:

```

export LD_LIBRARY_PATH
    =${LD_LIBRARY_PATH}:${R_HOME}/lib
    :${R_HOME}/library/SJava/libs

```

7. You may want to add the R_HOME and LD_LIBRARY_PATH variable settings to your ~/.bashrc file.

1.4 Installing Web User Interface

1. Make sure the computer meets specified requirements. Note that for correct functionality, Tomcat should not be installed in a directory which contains spaces in its path.
2. Make sure Tomcat is not running.
3. Copy all files from the `webinterface` directory in the BEEN installation directory to the `webapps/been` directory in Tomcat installation directory (this directory does not exist and must be created).
4. Copy following files from the BEEN installation directory to `webapps/been/WEB-INF/lib` directory in Tomcat installation directory (this directory does not exist and must be created):
 - `dist/been.jar`
 - `lib/webinterface/commons-fileupload-1.0.jar`
 - `lib/log4j/log4j-1.2.12.jar`
5. Because the BEEN web user interface is written in UTF-8 encoding, you need to adjust Tomcat's settings to respect this, if you want to use other than standard ANSI characters in entered data:
 1. Open the `conf/server.xml` file in Tomcat installation directory in a text editor.
 2. Find the `<Connector>` element with attribute `port="8080"` and add attribute `URIEncoding="UTF-8"` to this element.
 3. Save the `conf/server.xml` file.

2 Running BEEN

For successful running of benchmarks, you need to:

- Run Host Runtime on all participating hosts.
- Run Task Manager on one host.
- Run web interface on one host.
- Configure the web interface – set the host of Task Manager.
- Run all BEEN services.

All those components could be run on one host if needed. For running of benchmarks supplied with BEEN, additional hosts are necessary.

2.1 Running Task Manager

Task Manager needs to be run before all Host Runtimes – each Host Runtime connects to the Task manager on startup and would fail, if the Task Manager was not started at that time.

2.1.1 Windows

1. Open `bin` directory in the BEEN installation directory with Windows Explorer or other suitable application.
2. Run `taskmanager.bat` batch file.
3. Command Shell should appear with following messages:

```
Log level: DEBUG
Could not load configuration file. New configuration file
will be created (with default values set).
Task Manager started...
```

2.1.2 Linux

1. Open a terminal and go to `bin` directory in the BEEN installation directory.
2. Run `taskmanager.sh` shell script.
3. Following messages should appear on the terminal:

```
Log level: DEBUG
Could not load configuration file. New configuration file
will be created (with default values set).
Task Manager started...
```

2.2 Running Host Runtime

2.2.1 Windows

1. Open a command shell and go to `bin` directory in the BEEN installation directory.
2. Run `hostruntime.bat` batch file with a parameter specifying host, where the Task Manager is running.

Example:

```
hostruntime.bat aiya.ms.mff.cuni.cz
```

3. Following messages should appear in the command shell:

```
Note: Can't start the RMI registry - another instance is
probably running.
2006/12/04 16:35:06.640 INFO Initializing Load Monitor.
2006/12/04 16:35:06.687 INFO Load Monitor initialized
successfully.
Host Runtime started...
```

2.2.2 Linux

1. Open a terminal and go to `bin` directory in the BEEN installation directory.
2. Run `hostruntime.sh` shell script with a parameter specifying host, where the Task Manager is running.

Example:

```
./hostruntime.sh aiya.ms.mff.cuni.cz
```

3. Following messages should appear on the terminal:

```
Note: Can't start the RMI registry - another instance is
probably running.
2006/12/04 16:35:06.640 INFO Initializing Load Monitor.
2006/12/04 16:35:06.687 INFO Load Monitor initialized
successfully.
Host Runtime started...
```

2.3 Running and Configuring Web Interface

1. Open a command shell (on Windows) or a terminal (on Linux) and go to the BEEN installation directory.
2. Make sure Tomcat is running. You can start it by running the `start-tomcat` target in the `build.xml` build file.

```
ant start-tomcat
```

3. Run the web browser and go to URL `http://localhost:8080/been/`. The web interface of BEEN should appear.
4. Click on the **Configuration** tab in the web interface, fill in the **Task Manager host name** field and click **Save** button. A green bar with text “Configuration saved successfully.” should appear. If the Task Manager could not be found on entered host,

orange error message bar will appear with an explanation of error.

2.4 Running Services

1. Click on the **Services** tab in the web interface. A list of BEEN services will appear.
2. For each service, fill-in a host where the service should be run and click the **Start** button. The green bar with text “Service started successfully.” should appear. If the service could not be run on entered host, orange error message bar will appear with an explanation of error.

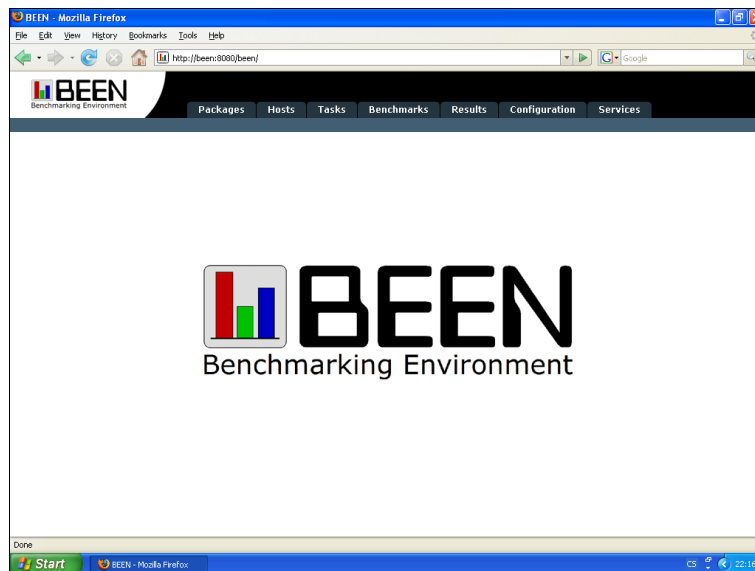
3 Using Web User Interface

Using the web interface, users can control all BEEN components, view their status, run benchmarks, inspect benchmark results, etc. The main advantage of the web interface is that there is no need to install a specialized application just to control BEEN – it can be controlled from any computer provided with a web browser.

For requirements of the server and client parts of the web interface, see section *Required Software (Part II, 1.1.2)*. Instructions for running the web interface can be found in section *Running and Configuring Web Interface (Part II, 2.3)*.

When the web interface is running, its URL is `http://hostname:8080/been/` where *hostname* is a host running the server part of the web interface.

In this section, we describe a structure and capabilities of the web interface and we present instructions how to accomplish various tasks.



Screenshot 1: BEEN web interface main page.

3.1 Web Interface Structure

The web interface functionality is divided into modules. Each module is presented to the user as a tab on the navigation bar. Most modules serve as an interface to specific BEEN services. The modules are:

- **Packages** – interface to the Software Repository
- **Hosts** – interface to the Host Manager
- **Tasks** – interface to the Task Manager
- **Benchmarks** – interface to the Benchmark Manager
- **Results** – interface to the Results Repository
- **Configuration** – allows user to configure the web interface, the Host Runtimes, and

the Host Manager

- **Services** – manages execution of all BEEN services

Detailed description of the web interface modules follows.

3.2 Packages Module

The Packages module serves as an interface to the Software Repository. It allows listing, viewing, uploading and downloading of the stored packages.

3.2.1 Working with Packages

Listing Packages

When you click on the **Packages** tab, a list of packages stored in the Software Repository will appear. The packages are divided into source, binary, task and data packages. For each package its human-readable name, package name and version are displayed.

To filter the list, click the **Add condition** link in the bar on the top of the page and specify condition on package metadata attributes using a select box for an attribute name, a condition operator and an attribute value. By clicking on the **Add condition** link again, you can add more conditions to the filter; by clicking on the **Delete condition** link, you can delete a specific condition. When you are done, click the **Filter** button and the list will be filtered only to packages matching all specified conditions.

Viewing Package Details

You can display package details by clicking on the package name. A page listing all package metadata attributes will appear.

Downloading Packages

You can download a package by clicking the **Download** button corresponding to the package you want to download. The web browser will probably show a dialog, where you can choose the location of the downloaded package. After confirmation, the download should begin.

The format of the downloaded packages is described in section *Packages (Part I, 6.1)*.

Deleting Packages

You can delete a package by clicking on the **Delete** button corresponding to the the package you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the package will be deleted; otherwise it will be kept.

Uploading Packages

You can upload a package to the Software Repository by clicking on the **Upload package** link in the navigation bar. A package upload form will appear. Enter the package file into the **Package file** box and click the **Upload** button. If the file entered is a valid Software Repository package file and is uploaded correctly, the green bar with text “Package uploaded successfully.” should appear. Otherwise, orange error message bar with an explanation of the error will appear.

3.3 Hosts Module

The Hosts module serves as an interface to the Host Manager. It allows manipulation with hosts, groups and software aliases.

3.3.1 Working with Hosts

Listing Hosts

When you click on the **Hosts** tab, a list of hosts in the Host Manager's database will appear. For each host, its hostname and status is displayed. Host status can be one of these values:

- **Online** – the host was online during the last check
- **Offline** – the host was shut down normally
- **Crashed** – the host does not send data to the Load Server, so it is assumed it is not responsive or was shut down abnormally
- **Unknown** – the host status is unknown

Displaying Detailed Information about Hosts

You can view detailed information about particular host by clicking on its host name. A page with several tabs will appear:

- **Configuration.** Displays a configuration of the host, as reported by the detector. By clicking on tabs in the **Configuration** tab, you can view information about the used detector, operating system, Java runtime, processors, memory, disk drives, network adapters, applications and user-defined properties.

The user-defined properties can be added, edited and deleted using controls on the **User-defined properties** tab.

By default, the most recent configuration detected is displayed in the **Configuration** tab. To display older detected configurations, select a date from the select box on the bar at the top of the **Configuration** tab and click the **Select** button.

To run a detector on the host and obtain fresh configuration information, click **Refresh** button on the bar at the top of the **Configuration** tab. The page indicating that new detection is in progress will appear. If the detection is successful, the fresh configuration will be displayed after a while; otherwise an error message with explanation of the error would appear.

- **Load.** Displays information about host utilization – the most current data obtained from the Load Monitor and averages computed from data obtained in last 10 and 60 minutes. Reported data include:
 - amount of free memory on the host
 - number of processes
 - process queue length
 - utilization of all host's processors
 - number of bytes read or written per second on all host's drives and network interfaces
- **Tasks.** Lists all tasks running on the host as reported by the Task Manager. You can

display detailed information about the task (by clicking on its ID) and context the task is running in (by clicking on the context ID). To kill a task, click the **Kill** button corresponding to the task you want to kill.

- **Logs.** Displays the logs produced by the tasks running on the host, as reported by the log storage component of the Task Manager. You can sort the logs by all columns by clicking on the column headers.

3.3.2 Working with Groups

For more information about host groups, see section *Host Groups (Part I, 7.4)*.

Listing Groups

When you click on the **Groups** link in the navigation bar, a list of host groups in the Host Manager's database will appear. For each group, its name is displayed. There is always one group present – “All hosts”. This group is special – it contains all hosts in the Host Manager's database and cannot be deleted.

Editing Groups

You can edit a group by clicking on the group name. A form will appear where you can change the group name, description and list of hosts in the group. The group name must be non-empty and unique.

The list of hosts can also be selected using a RSL expression. To activate the box for entering the RSL expression, click the **Select hosts using RSL expression** link. You can now enter the RSL expression. After clicking on the **Select matching hosts** button, the expression is evaluated in the background (i.e. without reloading the page) and hosts matching the expression are selected. If there is any syntactical or semantical error in RSL expression, the error message with an error description will appear. For more information about RSL, see section *Restriction Specification Language (RSL) (Part I, 11)*.

When you are done with editing of the group, click the **Edit** button to save your changes, or the **Cancel** button to discard your changes.

The “All hosts” group can be edited too, but only allowed operation is changing of the group description.

Adding Groups

You can add a group by clicking on the **Add group** link on the navigation bar. A form will appear with similar options as when editing a group.

When you are done with adding of the group, click the **Add** button to save your changes, or the **Cancel** button to discard your changes.

Deleting Groups

You can delete a group by clicking on the **Delete** button corresponding to the group you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the group will be deleted; otherwise it will be kept.

3.3.3 Working with Software Aliases

For more information about software aliases, see section *Software Aliases (Part I, 7.5)*.

Listing Alias Definitions

When you click on the **Aliases** link in the navigation bar, a list of software alias definitions defined at the Host Manager will appear. For each alias definition, its name is displayed.

Editing Alias Definitions

You can edit an alias definition by clicking on the alias definition name. A form will appear, where you can change the alias name, result name, result vendor, result version and restrictions for the operating system and the application. The alias name, result name and restriction for the application must be non-empty; the alias name must be unique.

In the **Result name**, **Result vendor** and **Result version** boxes, you can use special variables `${name}`, `${vendor}` and `${version}`. When matching the alias with installed applications, they will be replaced with real values of the matching application.

The restrictions are specified using RSL query language – see section *Restriction Specification Language (RSL) (Part I, 11)*.

When you are done with editing of the alias definition, click the **Edit** button to save your changes, or the **Cancel** button to discard your changes.

Adding Alias Definitions

You can add an alias definition by clicking on the **Add alias** link on the navigation bar. A form will appear with similar options as when editing an alias definition.

When you are done with adding of the alias definition, click the **Add** button to save your changes, or the **Cancel** button to discard your changes.

Deleting Alias Definitions

You can delete an alias definition by clicking on the **Delete** button corresponding to the the alias definition you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the alias definition will be deleted; otherwise it will be kept.

3.4 Tasks Module

The Tasks module serves as an interface to the Task Manager. It allows listing and viewing of contexts and tasks present in the system and executing of tasks.

3.4.1 Displaying Context and Task Information

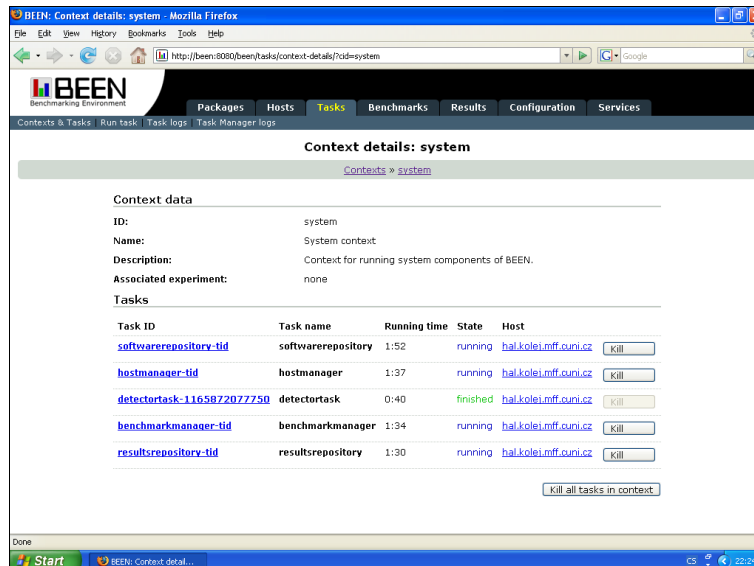
Listing Contexts

When you click on the **Tasks** tab, a list of existing contexts will appear. For each context, its ID and name are displayed.

Displaying Context Details

You can display context details and list of tasks in the context by clicking on the context name. A page will appear where you can view context ID, name, description and link to associated benchmarking experiment, if the context has one.

A list of tasks in the context is also present on the page. For each task, its ID, name, running time, status and host where the task is running are displayed. For list of possible task states, see section *Task Status (Part I, 3.3)*.



Screenshot 2: Context details.

Displaying Task Details

You can display task details by clicking on the task name in the context details page. A page with two tabs will appear:

- **Information.** Displays detailed information about the task, such as package name, host where the task is running, task status, restarting and timeout settings, paths to task directories, reached checkpoints and task properties.
- **Logs.** Displays the logs produced by the tasks running on the host, as reported by the log storage component of the Task Manager. You can sort the logs by all columns by clicking on the column headers.

Killing Single Tasks

Killing a single task can be accomplished in two ways:

1. In the context details page, click the **Kill** button corresponding to the task you want to kill.
2. In the task details page, click the **Kill** button at the bottom of the page.

In both cases, a confirmation will appear. If you click the **OK** button, the task will be killed; otherwise it will be left untouched.

Killing All Tasks in a Context

Killing all tasks in selected context can be accomplished in two ways:

1. In the context list page, click the **Kill all tasks in context** button corresponding to the context you want to kill all tasks in.
2. In the context details page, click the **Kill all tasks in context** button at the bottom of the page.

In both cases, a confirmation will appear. If you click the **OK** button, all tasks in the context will be killed; otherwise they will be left untouched.

Deleting Contexts

Deleting a context means killing all tasks in the context and deleting all the context data stored at the Task Manager. A context can be deleted in two ways:

3. In the context list page, click the **Delete context** button corresponding to the context you want to kill all tasks in.
4. In the context details page, click the **Delete context** button at the bottom of the page.

In both cases, a confirmation will appear. If you click the **OK** button, all tasks in the context will be killed; otherwise they will be left untouched.

Note that the `system` context is special – it cannot be deleted and it lasts forever.

3.4.2 Running Tasks

You can run a task by clicking on the **Run task** link on the navigation bar. A page with two tabs will appear:

- **Describe task using form.** In this tab, you can run a task by filling a form. You have to:
 - Select a task name from a list of names of task packages in the Software Repository.
 - Select a host on which the task should run from the list of hosts in the Host Manager database.
 - Select a context in which the task should run from the list of active contexts in the Task Manager.
 - Optionally enter task properties. Only string properties can be entered. You must enter one property on each line of the box in the format *name = value*. Whitespace around the name and value is trimmed.

When you are done with filling a form, click the **Run** button to run the task.

- **Describe task using task descriptor XML.** In this tab, you can run a task by directly entering the XML representation of the task descriptor into the **Task descriptor** form field. This means of running tasks serves mainly for debugging purposes. For the description of the task descriptor XML format, see section *Task Descriptor (Part I, 3.2)*.

When you are done with entering a task descriptor, click the **Run** button to run the task.

3.4.3 Displaying Logs

Displaying Logs of All Tasks

You can display logs of all tasks in the system by clicking on the **Task logs** link on the navigation bar. A page will appear which displays the logs produced by the all tasks in the system, as reported by the log storage component of the Task Manager. You can sort the logs by all columns by clicking on the column headers.

Displaying Task Manager Logs

You can display the Task Manager logs by clicking on the **Task Manager logs** link on the navigation bar. A page will appear which displays the Task Manager logs, as reported by its log storage component. You can sort the logs by all columns by clicking on the column headers.

3.5 Benchmarks Module

The Benchmarks module serves as an interface to the Benchmark Manager. It allows creating and monitoring of benchmark analyses and experiments, as well as listing and installing of the benchmarking plugins.

3.5.1 Working with the Entity Hierarchy

Listing Analyses

When you click on the **Analyses** link in the navigation bar, a list of analyses in the Benchmark Manager will appear. The analyses are divided into comparison analyses and regression analyses. For each analysis, its name is displayed.

Displaying Analysis Details

You can display analysis details and a list of experiments in the analysis by clicking on the analysis name in the analysis list page. A page with two tabs will appear:

- **General.** Displays basic analysis information (its name, description and type). In the bottom part, a list of experiments in the analysis is displayed. In case of regression analysis with created model experiment, the roles and callback R scripts set up when creating the experiment are displayed here too.
- **Logs.** Displays the logs produced by the tasks which were running in the analysis. You can sort the logs by all columns by clicking on the column headers.

In case of regression analysis with created model experiment, an additional tab will appear:

- **Scheduling information.** Allows entering of the scheduling information for running of the experiments. For description of the options, see *Entering scheduling information* in section *Creating Experiments (Part II, 3.5.2)*.

Displaying Experiment Details

You can display experiment details by clicking on the experiment name in the analysis details page. A page with two tabs will appear:

- **General.** Displays basic experiment information (its status, progress, number of binaries, runs and tasks), roles and callback R scripts set up when creating the experiment. In the bottom part, tasks of the experiment are displayed, divided hierarchically into the binaries and runs. For each task, its ID, context ID, name, running time, status and host where the task is running are displayed
- **Logs.** Displays a logs produced by the tasks which were running in the experiment. You can sort the logs by all columns by clicking on the column headers.

Task ID	Task name	Running time	State	Host	Context
34	log-upload	N/A	submitted	(unknown)	0-0
Binary 0					
0	local-download	0:30	finished	hal.koleimff.uni.cz	0-0
1	jboss-build	N/A	submitted	(unknown)	0-0
2	create-package-metadata	0:21	finished	hal.koleimff.uni.cz	0-0
3	localupload	N/A	submitted	(unknown)	0-0
4	local-download	0:24	finished	hal.koleimff.uni.cz	0-0
5	mysql-initialize	N/A	submitted	(unknown)	0-0
6	local-download	N/A	submitted	(unknown)	0-0
7	rubis-mysql-restore	N/A	submitted	(unknown)	0-0
8	mysql-run	N/A	submitted	(unknown)	0-0
9	mysql-runcommand	N/A	submitted	(unknown)	0-0
10	mysql-runcommand	N/A	submitted	(unknown)	0-0
11	local-download	0:26	finished	hal.koleimff.uni.cz	0-0

Screenshot 3: Tasks in running experiment.

3.5.2 Creating Experiments

After creating the analysis, experiments can be added into it.

In case of comparison analysis, the unlimited amount of experiments can be added into the analysis; each of them will be run once.

In case of regression analysis, only one experiment can be added into the analysis, along with scheduling information. Added experiment will serve as a “model experiment” for regular experiments created automatically by the Benchmark Manager according to the defined schedule.

Both experiments in comparison analysis and the model experiment in regression analysis are set up using a wizard. The wizard has following steps:

1. Plugin selection
2. Choosing experiment name and description
3. Entering scheduling information (regression analysis only)
4. Configuration of the experiment
5. Selection of hosts into benchmarking roles
6. Entering R callback scripts for statistical processing
7. Confirmation of entered data
8. Task generation (comparison analysis only)

You can activate the wizard by clicking on the **Create experiment** button in the analyses list page, which is corresponding to the analysis where you want to create the experiment.

In the wizard, you can move forward and backward using **Previous** and **Next** buttons. At the end of the wizard, the **Finish** button will be displayed instead of the **Next** button. Clicking on the **Finish** button will end the wizard and add created experiment to the analysis. The running of the experiment can be delayed a bit, as the Benchmark Manager processes experiments in background in regular intervals.

You can cancel the experiment creation at any time by clicking on the **Cancel** button in the wizard.

Detailed description of the wizard screens follows.

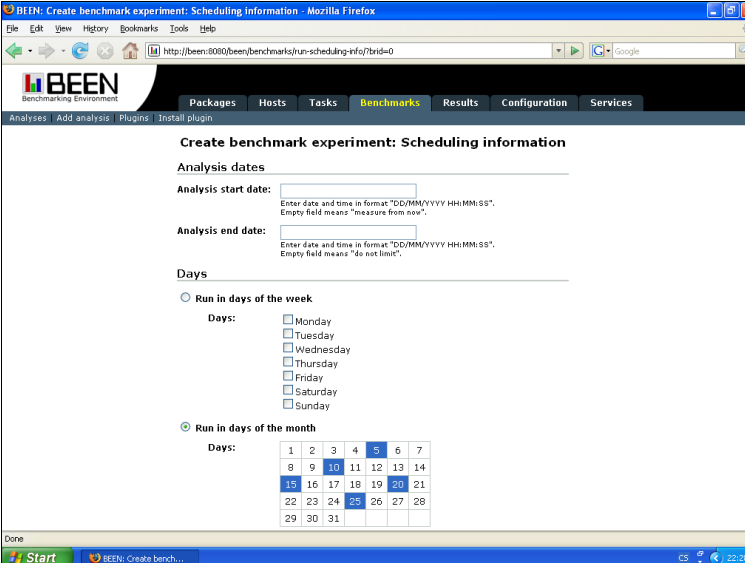
Plugin Selection

On this screen, you can select which benchmarking plugin will be used for the experiment. A list of plugins is displayed in the top part of the screen. Description of the selected plugin is displayed in the yellow box in the bottom part of the screen.

Choosing Experiment Name and Description

On this screen, you can enter experiment name (into the **Name** box) and description (into the **Description** box). The name must be non-empty, but does not have to be unique (it is only informational – internally, the experiments are uniquely identified within the analysis by automatically generated ID). The description is optional.

Entering Scheduling Information



Screenshot 4: Entering scheduling information.

On this screen, you can enter scheduling information for running of the experiments. This screen is displayed only when creating model experiment in the regression analysis.

In the **Analysis dates** section, you can enter start and end date of the analysis. Format of both dates is “DD/MM/YYYY HH:MM:SS”. You can enter none, one, or both dates. Empty start date is substituted by “now”, empty end date means the measurement will continue forever. If you enter both dates, they must be correctly ordered (start date must not be after the end date). If

you enter only end date, it must be set in the future.

In the **Days** section, you can choose days when the analysis is run. You can select either days of the week (by clicking on the **Run in days of the week** radio button and selecting days using checkboxes), or days of the month (by clicking on the **Run in days of the month** radio button and selecting days using the calendar, **Select all** link or **Select none** link).

In the **Time of the day** section, you can choose hours and minutes in the day when the analysis is run. The hours and minutes are both entered as a comma-separated list of numbers or number ranges (two numbers separated by dash). This format is similar to the format used by the `cron` Unix utility. Empty **Hours** field means “repeat analysis every hour”, similarly for minutes.

Configuration of the Experiment

Configuration of the experiment usually isn't single screen, but more screens generated by the experiment configurator. Those screens are plugin-dependent and contain various controls used to configure the experiment.

Usually, benchmarking plugin configurator will present a screen, where you can enter restrictions for used host roles in RSL query language. For more information about RSL, see section *Restriction Specification Language (RSL) (Part I, 11)*.

For description of options configurable in plugins supplied with BEEN, see sections *Benchmarking with Xampler (Part II, 4)* and *Benchmarking with RUBiS (Part II, 5)*. For description of controls available to the experiment configurator, see section *Configurator (Part III, 3.3)*.

Selection of Hosts into Benchmarking Roles

On this screen, you can select which host will belong to what roles. The screen may be skipped if the benchmarking plugin has no roles to set. For each role, a select box is displayed with hosts that the plugin determined to be suitable for the role (based on the experiment configuration and other factors, specific for each plugin). Required host count for each role is displayed too. The web interface will pre-select hosts to the role for you. You can select different hosts for the role, but you must respect the required host count.

If the required host count for some role can't be achieved, the error message will appear. You have to add hosts to BEEN (i.e. run the Host Runtime on them, with correct Task Manager host parameter) and refresh the screen – new hosts will be added dynamically.

Entering R Callback Scripts for Statistical Processing

On this screen, you can specify R callback scripts for the experiment. The scripts are initially provided by the benchmarking plugin. Following scripts can be edited:

- Complete run script
- Complete binary script
- Complete experiment script
- Incomplete binary script
- Incomplete experiment script

The scripts must be written in R language, using its standard functions or functions defined in script plugins, installed in the Results Repository. You can display help for the functions defined in the plugins by clicking on the **Help for BEEN R functions**.

For detailed information about the R scripts, see section *Statistics Calculation (Part I, 10.4)*.

Confirmation of Entered Data

On this screen, you can see a summary of data entered in the wizard so far (except the experiment configuration, which is not displayed for technical reasons). You can check it and if you see any error, you can go back and correct entered values.

In case of regression analysis, the wizard ends on this screen. After clicking on the **Finish** button, the model experiment is created and you will be returned to the analysis list page.

In case of comparison analysis, you can continue by clicking on the **Next** button, as on previous wizard screens.

Task Generation

On this screen, you can see a list of tasks generated for the experiment. For each task, its name and task ID are displayed. This screen is displayed only when creating experiment in the comparison analysis.

The wizard ends on this screen. After clicking on the **Finish** button, the experiment is created and you will be returned to the analysis list page.

3.5.3 Managing Plugins

For more information about Benchmark Manager plugins, see *Extending Benchmark Manager (Part III, 3)*.

Listing Installed Plugins

When you click on the **Plugins** link in the navigation bar, a list of plugins installed in the Benchmark Manager will appear. For each plugin, its name and status “OK” or “error”) are displayed.

Installing Plugins

You can install a plugin to the Benchmark Manager by clicking on the **Install plugin** link in the navigation bar. A package upload form will appear. Enter the plugin file into the **Plugin file** box and click the **Upload** button. If the file entered is a valid Benchmark Manager plugin file and is uploaded correctly, the green bar with text “Package uploaded successfully.” should appear. Otherwise, orange error message bar with an explanation of the error will appear.

3.6 Results Module

The Results module serves as an interface to the Results Repository. It allows viewing of the results' hierarchical structure, displaying computed statistics, graphs, associated metadata and logs. You can also export or delete the results and upload R script plugins.

If the Results Repository service could not initialize the R environment on the start-up, a yellow warning box will be displayed on all pages in the module to notify user about this fact. Results Repository cannot perform any statistical computation when R is disabled.

3.6.1 Browsing the Entity Hierarchy

Listing Analyses

When you click on the **Analyses** link in the navigation bar, a list of analyses, for which the Results Repository stores data, will appear. For each analysis, its name is displayed.

Exporting Analyses Data

The tab **Export** is present on the right side of the analysis list page. It allows export of data for one or more analysis from the Results Repository. To export the data, select analyses to export by clicking on checkboxes next to their name and click on the **Export selected** button. The web browser will probably show a dialog, where you can choose the location of the downloaded data. After confirmation, the download should begin.

Currently, the only supported export format is “BEEN archive database format”. Its structure is documented in section *Results Repository Export Format (Part I, 10.7)*.

Deleting Analyses Data

You can delete analysis data by clicking on the **Delete** button on the analysis list page, corresponding to the analysis you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the analysis data will be deleted; otherwise it will be kept.

If the **Delete** button is disabled, the analysis data is locked (probably because some computations are performed on it at the time) and can't be deleted. If you need to delete the analysis data, wait for a while and try reloading the page.

The analysis data can also become locked in the time between displaying the analysis list page and clicking on the **Delete** button. If you try to delete the analysis data in this situation, an error message with explanation will appear.

Displaying Analysis Details

You can display analysis details and a list of experiments in the analysis by clicking on the analysis name in the analysis list page. A page with five tabs will appear:

- **Statistics.** Displays statistics computed from the results of experiments in the analyses. The statistics are generated by the R scripts set up when creating the experiments.
- **Graphs.** Displays visual representation of the results of experiments in the analyses. The graphs are generated by the R scripts set up when creating the experiments.
- **Metadata.** Displays information about the analysis – see *Results Repository Metadata (Part IV, Appendix B)* for details.

All the metadata is read-only, except the comment, which can be changed. You can change the comment by modifying text in the **Comment** box on the **Metadata** tab and clicking on the **Change** button. There is no limitation on the comment length and it can be empty.

- **Logs.** Displays the logs produced by the tasks, which were running in the analysis. You can sort the logs by all columns by clicking on the column headers.
- **Export.** Allows exporting of the experiments' results. See section *Exporting Experiment Data*.

A list of experiments in the analysis is also present on the page. For each experiment, its name

and completeness status are displayed. By default, incomplete experiments are invisible – to display them, click on the **Show incomplete experiments** link.

Exporting Experiment Data

The tab **Export** is present on the right side of the analysis details page. It allows export of data for one or more experiments in the analysis from the Results Repository. To export the data, select experiments to export by clicking on checkboxes next to their name and click on the **Export selected** button. The web browser will probably show a dialog, where you can choose the location of the downloaded data. After confirmation, the download should begin.

Currently, the only supported export format is “BEEN archive database format”. Its structure is documented in section *Results Repository Export Format (Part I, 10.7)*.

Deleting Experiment Data

You can delete experiment data by clicking on the **Delete** button on the analysis details page, corresponding to the experiment you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the experiment data will be deleted; otherwise it will be kept.

If the **Delete** button is disabled, the experiment data is locked (probably because some computations are performed on it at the time) and can't be deleted. If you need to delete the experiment data, wait for a while and try reloading the page.

The experiment data can also become locked in the time between displaying the analysis details page and clicking on the **Delete** button. If you try to delete the experiment data in this situation, an error message with explanation will appear.

Displaying Experiment Details

You can display experiment details and a list of binaries in the experiment by clicking on the experiment name in the analysis details page. A page with five tabs will appear:

- **Statistics.** Displays statistics computed from the results of binaries in the experiment. The statistics are generated by the R scripts set up when creating the experiment.
- **Graphs.** Displays visual representation of the results of binaries in the experiment. The graphs are generated by the R scripts set up when creating the experiment.
- **Metadata.** Displays information about the experiment– see *Results Repository Metadata (Part IV, Appendix B)* for details.

All the metadata is read-only, except the comment, which can be changed. You can change the comment by modifying text in the **Comment** box on the **Metadata** tab and clicking on the **Change** button. There is no limitation on the comment length and it can be empty.

- **Logs.** Displays the logs produced by the tasks, which were running in the experiment. You can sort the logs by all columns by clicking on the column headers.
- **Export.** Allows exporting of the binaries' results. See section *Exporting Binary Data*.

A list of binaries in the experiment is also present on the page. For each binary, its name and completeness status are displayed. By default, incomplete binaries are invisible – to display them, click on the **Show incomplete binaries** link.

Exporting Binary Data

The tab **Export** is present on the right side of the experiment details page. It allows export of

data for one or more binaries in the experiment from the Results Repository. To export the data, select binaries to export by clicking on checkboxes next to their name and click on the **Export selected** button. The web browser will probably show a dialog, where you can choose the location of the downloaded data. After confirmation, the download should begin.

Currently, the only supported export format is “BEEN archive database format”. Its structure is documented in section *Results Repository Export Format (Part I, 10.7)*.

Deleting Binary Data

You can delete binary data by clicking on the **Delete** button on the experiment details page, corresponding to the binary you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the binary data will be deleted; otherwise it will be kept.

If the **Delete** button is disabled, the binary data is locked (probably because some computations are performed on it at the time) and can't be deleted. If you need to delete the binary data, wait for a while and try reloading the page.

The binary data can also become locked in the time between displaying the experiment details page and clicking on the **Delete** button. If you try to delete the binary data in this situation, an error message with explanation will appear.

Displaying Binary Details

You can display binary details and a list of runs in the binary by clicking on the binary name in the experiment details page. A page with five tabs will appear:

- **Statistics.** Displays statistics computed from the results of runs in the binary. The statistics are generated by the R scripts set up when creating the experiment.
- **Metadata.** Displays information about the binary– see *Results Repository Metadata (Part IV, Appendix B)* for details.

All the metadata is read-only, except the comment, which can be changed. You can change the comment by modifying text in the **Comment** box on the **Metadata** tab and clicking on the **Change** button. There is no limitation on the comment length and it can be empty.

- **Logs.** Displays the logs produced by the tasks, which were running in the binary. You can sort the logs by all columns by clicking on the column headers.
- **Export.** Allows exporting of the runs' results. See section *Exporting Run Data*.

A list of runs in the binary is also present on the page. For each run, its name and validity status are displayed. By default, invalid runs are invisible – to display them, click on the **Show invalid runs** link.

Exporting Run Data

The tab **Export** is present on the right side of the binary details page. It allows export of data for one or more runs in the binary from the Results Repository. To export the data, select runs to export by clicking on checkboxes next to their name and click on the **Export selected** button. The web browser will probably show a dialog, where you can choose the location of the downloaded data. After confirmation, the download should begin.

Currently, the only supported export format is “BEEN archive database format”. Its structure is documented in section *Results Repository Export Format (Part I, 10.7)*.

Deleting Run Data

You can delete run data by clicking on the **Delete** button on the binary details page, corresponding to the run you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the run data will be deleted; otherwise it will be kept.

If the **Delete** button is disabled, the run data is locked (probably because some computations are performed on it at the time) and can't be deleted. If you need to delete the run data, wait for a while and try reloading the page.

The run data can also become locked in the time between displaying the binary details page and clicking on the **Delete** button. If you try to delete the run data in this situation, an error message with explanation will appear.

Displaying Run Details

You can display run details by clicking on the run name in the binary details page. A page with several tabs will appear:

- **Statistics.** This tab is displayed only for valid runs. It displays statistics computed from the results of the run. The statistics are generated by the R scripts set up when creating the experiment.
- **Raw data.** This tab is displayed only for invalid runs. It displays a text informing that the run is invalid and allows downloading of the raw run data for possible analysis and debugging.

To download the raw data, click on the **download raw data** link on the **Raw data** tab. The web browser will probably show a dialog, where you can choose the location of the downloaded data. After confirmation, the download should begin.

- **Load.** Displays visual representation of the utilization of the computer where the run was run.
- **Metadata.** This tab is displayed only for valid runs. It displays information about the run – see *Results Repository Metadata (Part IV, Appendix B)* for details. All the metadata is read-only.
- **Logs.** Displays the logs produced by the tasks, which were running in the run. You can sort the logs by all columns by clicking on the column headers.

3.6.2 Managing R Script Packages

For more information about R script packages, see section *R Script Packages (Part III, 4.2)*.

Listing Installed R Script Packages

When you click on the **R script packages** link in the navigation bar, a list of R script packages installed in the Results Repository will appear. For each package, its name and description are displayed.

Deleting R Script Packages

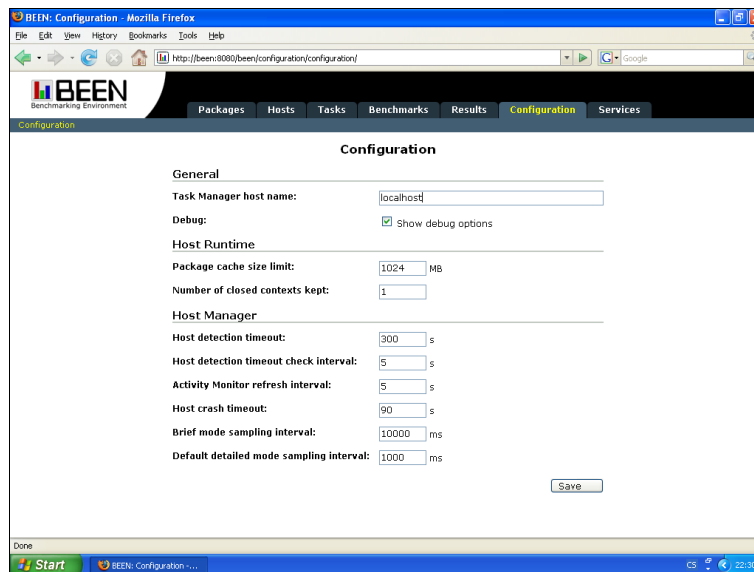
You can delete a R script package by clicking on the **Delete** button corresponding to the package you want to delete. After clicking, a confirmation will appear. If you click the **OK** button, the package will be deleted; otherwise it will be kept.

Uploading R Script Packages

You can upload a R script package to the Results Repository by clicking on the **Upload R script package** link in the navigation bar. A package upload form will appear. Enter the package file into the **Package file** box and click the **Upload** button. If the file entered is a valid Results Repository package file and is uploaded correctly, the green bar with text “Package uploaded successfully.” should appear. Otherwise, orange error message bar with an explanation of the error will appear.

3.7 Configuration Module

The Configuration module allows you to configure the web interface, the Host Runtimes and the Host Manager.



Screenshot 5: BEEN configuration.

To display the configuration options, click on the **Configuration** tab. The options are divided into three groups:

- **General.** General options are stored at the web interface and are always displayed. Following options are listed in the **General** group:
 - **Task manager host name** – the host where the Task Manager is running. Correct setting of this option is necessary for functionality of all web interface modules, as the Task Manager manages tasks and services the web interface communicates with. Entered host must exist and the Task Manager must be running on it.
 - **Show debug options** – if checked, the Services module shows controls allowing to run all services on the same host as the Task Manager at once, to stop all services at once, and to run them in remote debugging mode. For detailed description, see section *Starting, restarting and stopping services*.
- **Host Runtime.** Host Runtime options are stored at the Task Manager and distributed to Host Runtimes after their registration. The options are displayed only when the Task Manager is running and the web interface is connected to it. Following options are listed in the **Host Runtime** group:
 - **Package cache size limit** – the size limit of the Host Runtime package cache, in megabytes. The entered number must be positive.

- **Number of closed contexts kept** – number of closed contexts, for which the Host Runtime should keep data on the disk. The entered number must be non-negative.
- **Host Manager.** Host Manager options are stored at the Host Manager and displayed only when the Host Manager is running. Following options are listed in the **Host Manager** group.
 - **Host detection timeout**
 - **Host detection timeout check interval**
 - **Activity Monitor refresh interval**
 - **Host crash timeout**
 - **Brief mode sampling interval**
 - **Default detailed mode sampling interval**

All intervals are specified in seconds, except the sampling intervals, which are specified in milliseconds. All entered numbers must be positive. Additionally, the Host crash timeout must be greater than brief mode sampling interval.

For detailed description of the Host Manager configuration settings, see section Host Manager Configuration (Part I, 7.6).

3.8 Services Module

The Services module displays manages execution of all BEEN services.

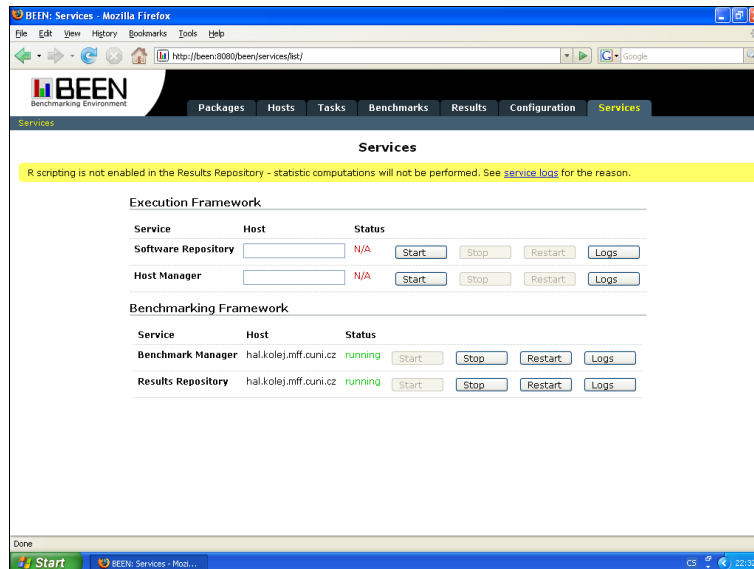
3.8.1 Working with Services

Listing Services

You can display a list of BEEN services by clicking on the **Services** tab. Services are divided into execution framework and benchmarking framework. For each service, its name and status are displayed. If the service is running, the host where it is running is displayed too. Service status can be of the following values:

- Starting
- Running
- Stopping
- Restarting
- N/A

If the Results Repository is started, but could not initialize the R environment on the start, a yellow warning box will be displayed on all pages in the module to notify user about this fact. Results Repository cannot perform any statistical computation when R is disabled.



Screenshot 6: Service list with warning about disabled R scripting displayed.

Starting, Restarting and Stopping Services

You can start a service by entering a host, where you want to run the service, into a box corresponding to the services you want to run, and clicking on the **Start** button. To start the service successfully, the host must exist and the Host Runtime must be running on it.

You can restart a running service by clicking on the **Restart** button corresponding to the service you want to restart.

You can stop a running service by clicking on the **Stop** button corresponding to the service you want to stop.

If the debug options are enabled, several new options are available:

- You can start all non-running services by clicking on the **Start all services** button.
- You can stop all running services by clicking on the **Stop all services** button.
- You can start a service with parameters for remote debugging by clicking on the **Start (remote debug)** button. Following parameters are passed to the Java Virtual Machine which is running the service:

```
-Xdebug -Xrunjdpw:transport=dt_socket,address=8000,
suspend=y,server=y
```

Displaying Service Logs

You can display service logs by clicking on the Logs button corresponding to the service whose logs you want to display. A page will appear which displays the service logs, as reported by the log storage component of the Task Manager. You can sort the logs by all columns by clicking on the column headers.

4 Benchmarking with Xampler

Xampler is a benchmarking suite for CORBA brokers. It allows performance measurements of the most major CORBA brokers including JacORB, Jonathan, omniORB, OpenORB, ORBacus, OrbRiver, Orbix, TAO and VisiBroker. There are thirteen separate benchmarking suites covering various scalability aspects and ranging from a simple ping to monitoring a number of servant instances.

Xampler plugin currently supports benchmarking of omniORB broker on Linux platform. Xampler – omniORB combination was selected as proof of concept for running platform dependent benchmarks in BEEN environment; on the contrary to the Rubis benchmark which is a Java-based multiplatform application.

4.1 Introduction to Xampler

Xampler's benchmarking suites are formed by the client and server binaries. When the server starts, it stores its IOR reference in a file on local disk. Clients then read the reference from the file and use it to locate the server. Afterwards benchmark-specific measurement is initiated and results are written on standard output of both client and server processes.

To compile benchmark binaries, the broker's binary must be present on the compilation host running because the IDL compiler is required by the compilation.

By default, results generated by the Xampler are stored in the format which is hard to parse and process automatically. Hence the benchmarks are run in the *raw* mode. In the raw mode, only clients produce output which is stored in the more parser-friendly format.

In a common benchmarking setup the server and client applications are expected to run on different hosts so the IOR reference stored on server's hosts must be transferred to clients. This is accomplished by a dedicated task which waits for server's startup, reads the IOR reference and sets a checkpoint with a value of the reference. Client's execution task waits for the checkpoint and then stores the reference and executes the client process. Standard output of client is redirected to a file, so that results can be later collected and processed by the Results Repository.

4.2 Comparison Analyses with Xampler

Xampler plugin requires specific information to be able to setup a benchmarking experiment. The configuration is performed on four screens in following order:

- **omniORB CVS Repository screen** – configures access to a CVS repository with omniORB versions. Since omniORB is a SourceForge project, anonymous access to the CVS is available. OmniORB broker is developed in branches which correspond to its major versions. The **branch** option allows user to select a branch to checkout source from. Since Xampler provides suites for both 4.0.X and 4.1.X versions, the branches of interest should be `omni4_1_develop` and `omni4_0_develop`. The other development versions are not supported
- **Software selection screen** – collects version information for the experiment. Xampler and omniORB version fields correspond to the versions of packages available in the software repository. Branch option should correspond to the option specified on the CVS access screen - being 4.0 for `omni4_0_develop` and 4.1 for `omni4_1_develop`

branch. Number of binaries and runs determines how many binaries will be created for each source package of omniORB and how many runs of a suite will be benchmarked for each binary

- **Suite selection screen** – allows user to select one of available suites
- **Suite subselection screen** – is optional and appears for the most of marshalling suites. It allows selection of a data type which will be marshaled and one of in/out/inout directions. See Xampler documentation for details
- **Settings screen** – specifies compile-time and run-time options for benchmarking. For the concurrent client connections suite, the runtime settings specify a number of clients that the server should wait for before starting measurements. While runtime options are passed to benchmark executables on command-line, the compile-time options are hacked into C++/all_defs.h file
- **Client hosts specification screen** – allows user to select hosts on which clients will be started
- **Server host specification screen** – allows user to input RSL restrictions on server hosts
- **Compilation host screen** – allows user to specify conditions for a hosts on which Xampler and omniORB are compiled

4.3 Regression Analyses with Xampler

The process of configuring the model experiment for a regression analysis is very similar to the configuration process of a comparison experiment. Version provider for Xampler takes advantage of omniORB being a SourceForge.net project (<http://omniorb.sourceforge.net/>) with public CVS access and generates timestamps used when downloading specific history revisions. For each newly created source package new omniORB and Xampler binaries are created to maximize consistency of the results.

4.4 Required Packages

The plugin requires only Xampler source package to be present in the Software Repository, omniORB sources will be downloaded from the CVS repository for both types of analyses. Following tasks are required when working with the Xampler:

- **Generic tasks:**
 - local-download task, version 1.0
 - localupload task, version 1.0
 - create-package-metadata task, version 1.0
 - log-collection task, version 1.0
- **omniORB:**
 - native-cvs-download task, version 1.0
 - omniorb-source-package task, version 1.0
 - omniorb-compile-linux task, version 1.0
- **Xampler:**
 - xampler-compile-linux task, version 1.0
 - xampler-execute task, version 1.0
 - check-and-convert-xampler task, version 1.0

- `xampler` source package, recommended version 1.11

4.5 Xampler Role Requirements

As stated before, plugin supports only benchmarking on Linux platforms. Following are the recommended specifications for various roles in the benchmark:

Server/Client:

- 512 MB of RAM
- 200 MB of disk space
- Python 2.4 or newer

Compilation:

- CVS
- Python 2.4 or newer
- gcc 3.3.6 or newer

4.6 Notes

- `xampler-execute` tasks are designated as context-exclusive. They will not start on a machine where one of BEEN components is running because the host can't be "locked".
- Client and server may run on a single machine, or in a distributed manner.
- The benchmark generates a decent load on machine and it may become unresponsive resulting in Host Manager warnings about unreachable Host Runtime.
- The CVS uses pserver connection, sending password in plain-text only, so watch your credentials.

5 Benchmarking with RUBiS

RUBiS is an application server scalability benchmark. It implements a web auction site, modeled after eBay.com, and its basic functionality: selling, browsing and bidding. Three kinds of user sessions are distinguished: visitor, buyer and seller. For a visitor session, users don't need to register, but are only allowed to browse the site. Buyer and seller session require registration. Buyers can additionally bid on items, browse their current bids, and read comments and ratings of other users. Seller sessions require a fee before the user is allowed to put an item for sale. The seller can specify a minimum price of his item.

5.1 Client Emulator

RUBiS measures the application servers' performance by emulating client interaction via a web browser. One *client emulator* can concurrently simulate several clients. There are 26 defined interactions that can be performed from the client's web browser. The most important are:

- browsing items by category or region
- bidding, buying or selling items
- leaving comments on other users and consulting one's own user page.

Browsing items also includes consulting the bid history and the seller's information.

Sequence of interactions for the same customer is called a *session*. For each customer session, the client emulator opens a persistent HTTP connection to the web server (provided by the application server) and closes it at the end of the session.

Client's workload is defined by a *transition table*. The table specifies a matrix of probabilities of transitions between every interaction. There are two transition tables, one with browsing transitions that imply read-only interactions, and one with bidding transitions that include 15% of read-write interactions. The bidding transitions are the most representative of a real auction site workload. The client waits for some time before transitioning to the next interaction. The waiting time is called the *think time*. The think time and the total session time are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively.

5.2 Server

RUBiS provides several implementations of the auction site, which can be considered as different benchmarks. There are several implementations in EJB (Enterprise Java Beans), a PHP implementation and a Java servlets implementation. EJB implementations are of the biggest interest to us, and they are the only ones supported in the RUBiS plugin for BEEN.

The purpose of an EJB server is to abstract the application business logic from the underlying middleware. There are two types of beans: *entity beans* that map data stored in the database to objects (usually one entity bean instance per database table row), and *session beans* that are used to perform temporary operations (stateless session beans) or represent temporary objects (stateful session beans). The EJB server is responsible for providing services such as database access (JDBC), transactions (JTA), naming (JNDI) or management support (JMX).

The RUBiS auction site has several EJB implementations, which can be used as different benchmarks of EJB servers:

- **Session Beans** – session beans are used to implement the business logic, leaving only the presentation logic in the servlets. This implementation uses the fewest services from the EJB container. The session beans benefit from the connection pooling and the transaction management provided by the EJB server. It greatly simplifies the servlets-only code, in which the connection pooling would have to be implemented by hand.
- **DAO separation with Entity Beans CMP** – in this implementation, the data access code is extracted from the servlets, and moved into Data Access Objects (DAO) that are implemented using entity beans. The business logic embedded in the servlets directly invokes methods on the entity beans that map the data stored in the database. With container-managed persistence (CMP), the vast majority of the SQL queries is generated by the EJB container. EJB 1.1 CMP, however, requires stateless session beans to execute complex queries involving joins on multiple tables. To avoid fine-grained access of getter/setter methods of the beans, functions are provided that return results populated with the values of the bean instance attributes. With this implementation, the impact of fine-grained accesses between the Web and EJB containers is evaluated.
- **DAO separation with Entity Beans BMP** – this implementation is the same as the DAO separation with entity beans CMP version, except that bean-managed persistence (BMP) is used. With BMP, the SQL queries have to be hand-coded in the beans. Exactly the same queries as the CMP version are implemented, including the use of a stateless bean to execute complex queries. The goal of this implementation is to evaluate the cost of the container's persistence service by comparing it with the Entity Beans CMP version.
- **Session facade** – the session facade pattern uses stateless session beans as a facade to abstract the entity components. This method reduces the number of business objects that are exposed to the clients over the network, thus providing a uniform coarse-grained service access layer. Calls between facade and entity beans are local to the EJB server and can be optimized to reduce the overhead of multiple network calls. Container-managed persistence is used for the entity beans. This implementation involves a larger number of beans, and thus stresses the component pooling of the container. It also exploits the database connection pooling, transaction manager and persistence services.
- **Session facade with local interfaces** – although the session facade beans and the entity beans execute inside the same JVM, with RMI (Remote Method Invocation) the communication between them has to go through all the communication layers, as if they were on different machines. The EJB 2.0 specification introduces local interfaces to optimize intra-JVM calls by bypassing the communication layers. Beans with a local interface cannot be called remotely, i.e. from another JVM even if the JVM runs on the same machine. This implementation takes advantage of these local interfaces. This implementation uses the session facade pattern, and container-managed persistence. Only session facade beans have a remote interface that is exposed to the servlets. The entity beans only have a local interface that is used by the session facade beans. Therefore, interactions between session and entity beans bypass the communication layers. This implementation requires EJB 2.0 compliant containers.
- **MDB** – in this implementation, the data access code is extracted from the servlets, and moved into Message Driven Beans (MDB). The business logic embedded in the servlets directly invokes methods on the MDB that query the database.
- **EJB CMP2.0** – this implementation uses the session facade pattern and local interfaces as described above in the “Session facade with local interfaces” section. It also takes advantage of container-managed persistence 2.0 (CMP2.0). EJB-QL is used in most of the queries, however a stateless session bean is required to execute some complex

queries. This implementation requires EJB 2.0 compliant containers.

5.3 Database

RUBiS uses a MySQL database with 7 tables: `users`, `items`, `categories`, `regions`, `bids`, `buy_now` and `comments`. The `users` table records contain the user's name, nickname, password, region, rating and balance. Besides the category and the seller's nickname, the `items` table contains the name that briefly describes the item and a more extensive description, usually an HTML file. Every bid is stored in the `bids` table, which includes the seller, the bid, and a `max_bid` value used by the proxy bidder (a tool that bids automatically on behalf of a user). Items that are directly bought without any auction are stored in the `buy_now` table. The `comments` table records user comments. As an optimization, the number of bids and the amount of the current maximum bid are stored with each item to prevent many expensive lookups of the `bids` table. This redundant information is necessary to keep an acceptable response time for browsing requests. As users only browse and bid on items that are currently for sale, the `items` table is split in a new and an old items table. The very vast majority of the requests access the new items table, thus considerably reducing the working set used by the database.

The database is sized according to some observations found on the eBay Web site. There are about 33,000 items for sale, distributed among eBay's 20 categories and 62 regions. A history of 500,000 auctions is kept in the old-items table. There is an average of 10 bids per item, or 330,000 entries in the `bids` table. The `buy_now` table is small, because less than 10 % of the items are sold without auction. The `users` table has 1 million entries. It's assumed that users give feedback (comments) for 95 % of the transactions. The `comments` table contains about 506,500 comments referring either to items or old items. The total size of the database, including indices, is 1.4 GB.

5.4 Using the RUBiS Plugin in BEEN

The RUBiS plugin for BEEN supports benchmarking of the Jonas and JBoss application servers. When creating an experiment, you will be presented with several configuration screens:

- **Software selection** – select the EJB server that will be benchmarked, its version and platform where it will run. If you enter a version which has a binary package in the Software Repository, then the binary will be used for the benchmarking. Otherwise, a source package of the same version is required in the Software Repository. Select the version and platform of the MySQL server similarly.
- **Server implementation** – select which EJB implementation of the auction site will be deployed. See their descriptions in section *Server (Part II, 5.2)*.
- **Compilation roles** – set requirements on the host that will compile the EJB server using the RSL query language.
- **Benchmarking roles** – set requirements on the hosts that will figure in the benchmarking roles using the RSL query language. Different hosts should be used in different roles. The benchmarking roles are:
 - **Clients** – the client emulators will be run on hosts of the Clients role. Several client emulators can be run simultaneously, thus several hosts can figure in the Clients role. Their number is specified in a later configuration screen.
 - **EJB server** – the EJB server (JBoss or Jonas) will run on the host of this role. One host will be used for this role.

- **Database** – the MySQL database will run on the host of this role. One host will be used for this role.
- **Benchmark and RUBiS properties** – on this screen you can configure the experiment properties and RUBiS runtime properties:
 - **Run count** – number of runs in this experiment.
 - **Client emulator count** – number of hosts in the Clients role. The client emulator will be run on each of these hosts. Higher number of hosts in the Clients role implies higher workload on the EJB server. On each client emulator, a specified number of clients is emulated (their number is set on this screen too, see below). If the total number of emulated clients is too high, the clients will encounter errors (timeouts, etc.). This situation should be avoided.
 - **Emulated clients per host** – number of clients emulated by each client emulator. Be careful not to set the number too high, like in client emulator count.
 - **Up ramp time (in minutes)** – the duration of the “up ramp” phase, in minutes. The “up ramp” phase is used to “warm up” the EJB server and the database to stabilize the results.
 - **Up ramp slowdown factor** – the think time is multiplied by this number during the up ramp phase.
 - **Session time (in minutes)** – the duration of the main benchmarking phase.
 - **Down ramp time (in minutes)** – the duration of the “down ramp” phase, in minutes. The down ramp phase is used to “slow down” the EJB server and the database to stabilize the results.
 - **Down ramp slowdown factor** – the think time is multiplied by this number during the down ramp phase.
 - **Maximum transitions per client session** – maximum number of transitions between interactions during one client session. When this number is reached, the session of one emulated client is finished, and another client will be emulated (thus there is always the required number of emulated clients per client emulator).
 - **Debug level** – debug level of the client emulator. The levels are:
 - **0** – no debug messages
 - **1** – error messages are written to the standard output
 - **2** – error messages and their HTML pages are written to the error output
 - **3** – all HTML pages received from the server are written to the standard output; error messages and their HTML pages are written to the error output
 - **Database dump type** – select the type of the database dump of the RUBiS database. The contents of the RUBiS database will be initialized from the dump. There are two dump types:
 - **Binary** – a copy of MySQL's binary files will the database content. This might not be compatible with a different version of MySQL than the one which was used to create it. Filling the RUBiS database from this dump is faster than the text dump.
 - **Text** – a textual form of the RUBiS database. This dump type has better compatibility with different MySQL versions. Filling the RUBiS database

from this dump is slower than from the binary dump.

- **Transition table** – choose a browsing (read-only) transition table, or a bidding (read-write) one.

5.4.1 Required Packages

- **MySQL:**
 - MySQL binary package, version at least 5, for the required platform (Linux, Win32)
 - `mysql-initialize` task, version 1.0
 - `mysql-run` task, version 1.0
 - `mysql-runcommand` task, version 1.0
 - `mysql-shutdown` task, version 1.0
- **RUBiS:**
 - RUBiS source package, version 1.5.1
 - `rubis-mysql-dump` data package, version 1.0, required if the text database dump will be used
 - `rubis-mysql-dump-binary` data package, version 1.0, required if the binary database dump will be used
 - `rubis-build-prepare` task, version 1.0
 - `rubis-deployer` task, version 1.0
 - `rubis-mysql-backup` task, version 1.0
 - `rubis-mysql-initialize` task, version 1.0
 - `rubis-mysql-restore` task, version 1.0
 - `rubis-run`, task, version 1.0
- **For benchmarking JBoss:**
 - JBoss source or binary packages, version at least 4, required for comparison analyses
 - `svn-checkout` task, version 1.0, required for regression analyses
 - `jboss-build` task, version 1.0, required if JBoss is built from source
 - `jboss-configure` task, version 1.0
 - `jboss-run` task, version 1.0
 - `jboss-shutdown` task, version 1.0
 - `local-download` task, version 1.0
 - `create-package-metadata` task, version 1.0, required if JBoss is built from source
- **For benchmarking Jonas:**
 - Jonas source or binary packages, version at least 4.8, required for comparison analyses
 - `svn-checkout` task, version 1.0, required for regression analyses
 - `create-package-metadata` task, version 1.0, required if Jonas is built from

source

- `jonas-build` task, version 1.0, required if Jonas is built from source
- `jonas-configure` task, version 1.0
- `jonas-run` task, version 1.0
- `jonas-shutdown` task, version 1.0
- **Results Collection:**
 - `check-and-convert-rubis` task, version 1.0
 - `results-collect` task, version 1.0
 - `log-upload` task, version 1.0

5.4.2 RUBiS Role Requirements

All roles can be run on Windows and Linux platforms. The roles have following requirements on the hosts where they are run:

Jonas or JBoss compilation:

- Java Development Kit 5.0
- Ant 1.6.x
- 500 MB of free disk space
- at least 512 MB RAM recommended

Clients:

- on Linux, `libstdc++.so.5` installed

EJB Server:

- Java Development Kit 5.0
- Ant 1.6.x
- 500 MB of free disk space
- at least 1 GB RAM recommended
- fast CPU (~2 GHz or faster) recommended

Database:

- 4 GB of free disk space
- at least 512 MB RAM recommended
- fast CPU (~2 GHz or faster) recommended

5.4.3 Notes

- The RUBiS plugin uses context exclusive tasks for running the EJB server, MySQL and client emulators. Thus you cannot run it on the same host as BEEN's services. This avoids corruption of results.
- The `jboss-run` and `jonas-run` tasks start their respective EJB servers, and wait until they finish. To see the output of an EJB server, view the output of these tasks. Similarly, for the output of MySQL consult the output of the `mysql-run` task.
- Don't run the database role on a host which has MySQL already installed (even if it's not running). The MySQL server used by RUBiS would use the other server's system

wide configuration file. Unfortunately, this behavior cannot be stopped.

- 100 total emulated clients (i.e. number of client emulators multiplied by the number of clients emulated by every client emulator) create a decent load on the EJB servers.
- Avoid running other web services on the host in the EJB server role. They might occupy ports required by JBoss or Jonas (e.g. 8080). In particular, avoid running BEEN web user interface on the host.

6 Executing a Simple Benchmarking Analysis

In this section, we present a step-by-step guide how to run a simple RUBiS comparison analysis.

6.1 Executing RUBiS Comparison Analysis

For executing the example RUBiS comparison analysis, you will need four computers (hosts). We will refer to them as *A*, *B*, *C* and *D*. Substitute the letters with real hostnames when trying to run the analysis by this guide.

All hosts must satisfy general requirements on running BEEN – see section *Required Software* (Part II, 1.1.2). The hosts may run Windows or Linux system (except host *A*, which must run Linux). Additionally:

- Host *A* must satisfy requirements for running the Results Repository and BEEN web interface. For the description of the requirements, see section *Required Software* (Part II, 1.1.2).
- Host *B* must satisfy requirements for running RUBiS “EJB server” role.
- Host *C* must satisfy requirements for running RUBiS “Clients” role.
- Host *D* must satisfy requirements for running RUBiS “Database” role.

For description of the requirements of the RUBiS roles, see section *RUBiS Role Requirements* (Part II, 5.4.2).

6.1.1 Execution Steps

1. Install BEEN on all hosts. For installation instructions, see section *Installing BEEN Execution Environment* (Part II, 1.2).
2. Install all prerequisites for running the Results Repository on host *A*. For installation instructions, see section *Installing Results Repository Prerequisites* (Part II, 1.3).
3. Run the Task Manager on host *A* using `taskmanager.bat` batch file (on Windows) or `taskmanager.sh` script (on Linux) in the `bin` directory.
4. Run the Host Runtime on all hosts using `hostruntime.bat` batch file (on Windows) or `hostruntime.sh` script (on Linux) in the `bin` directory.
5. Run the BEEN web interface on host *A* by executing `ant redeploy-with-restart` command in the BEEN installation directory. Verify that the web interface is running by browsing on the URL `http://A:8080/been/`. The main screen of the BEEN web interface should appear. For detailed information about running the BEEN web interface, see section *Running and Configuring Web Interface* (Part II, 2.3).
6. In the web interface, click on the **Configuration** tab, enter “*A*” into the **Task Manager host name** box and click **Save**.
7. In the web interface, click on the **Services** tab. A list of BEEN services should appear. No service should be running (their state should be “N/A”). For each service, enter “*A*”

into the **Host** box and click the **Start** button. This will start all services on host A.

After starting the services, make sure there is no yellow bar with information about unavailable R scripting displayed. If the bar appears, host A probably does not meet the requirements for running the Results Repository (it cannot initialize the R environment). In this case, you have two options:

1. Stop the Results Repository service, install and configure R correctly on host A and start the Results Repository again.
 2. Choose another host which meets the requirements as A and try the guide again from the beginning.
8. In the web interface, click on the **Packages** tab and then on the **Upload packages** link in the navigation bar. You now have to upload several packages required to run RUBiS comparison analysis from the BEEN DVD to the Software Repository, or download them from the project web.

The required packages are listed in section *Required Packages (Part II, 5.4.1)*. You need to upload the packages listed in “MySQL”, “RUBiS” and “For benchmarking JBoss” parts (i.e. Jonas packages need not to be uploaded).

All the packages are placed in the `/Packages` directory on the BEEN DVD, or available from the BEEN web (<http://been.objectweb.org>).

To upload the package, enter its filename into the **Package file** box and click the **Upload** button. A green bar with text “Package uploaded successfully.” should appear after the package is uploaded.

9. Click on the **Benchmarks** tab. The (empty) list of benchmarks analyses should appear. Click on the **Add analysis** link in the **Comparison analyses** section. A form will appear where you can enter the name and description of the new analysis. Enter “Test analysis” as an analysis name and leave the description empty. When you are finished, click on the **Add** button.
10. In the analysis list, click on the **Create experiment** button corresponding to the created analysis. The plugin selection screen will appear.
11. Select “RUBiS” from the list of plugins and click on the **Next** button. The screen where you can enter experiment name and description will appear.
12. Enter “Test experiment” as an experiment name and leave the description empty. When you are finished, click on the **Next** button. The RUBiS plugin configuration screen will appear.
13. You must select which software will be benchmarked by RUBiS. Leave the versions at default values. The **Platform** selection for the JBoss should match the platform of host B and the **Platform** selection for the MySQL should match the platform of host D. When you are finished, click on the **Next** button. The RUBiS plugin configuration screen with choice of the EJB implementation will appear.
14. Select any value from the **Implementation** list. For more information about the implementations, see section *Server (Part II, 5.2)*. When you are finished, click on the **Next** button. The RUBiS plugin configuration screen for setting the compilation role conditions will appear.
15. Enter string name = “B” into the **JBoss** box and click on the **Next** button. The RUBiS plugin configuration screen for setting the benchmarking roles conditions will appear.
16. Enter string name = “C” into the **Clients** box, name = “B” into the **EJB Server** box and name = “D” into the **Database** box. When you are finished, click on the **Next** button. The RUBiS plugin configuration screen for setting miscellaneous options will

appear.

17. Leave all settings on default values and click on the **Next** button. The screen for selection of hosts into benchmarking roles will appear.
18. Host B should be selected in the **EJB Server** role, host C should be selected in the **Clients** role and host D should be selected in the **Database** role. Just click on the **Next** button. The callback R scripts screen will appear.
19. Leave the callback R scripts settings on default values and click on the **Next** button. The confirmation screen will appear.
20. Check that all displayed values are correct and match the instructions above. If so, click on the **Next** button and the screen with list of tasks will appear. If the values are incorrect, go back through the wizard and correct them.
21. Check that the list of tasks is not empty and click on the **Finish** button. The analysis list page will appear and the experiment will be created and scheduled to run by the Benchmark Manager. The running of the experiment can be delayed a bit, as the Benchmark Manager processes experiments in background in regular intervals.
22. In the analysis list, click on the **Test analysis** link. The analysis details page will appear.
23. In the analysis details page, click on the **Test experiment** link. The experiment details page will appear. At the top of the page, you can watch the experiment progress; in the bottom part you can see the experiment's tasks. If you want to see updated information, reload the page.
24. After the experiment finishes (the progress reaches 100 %), you can view its results. Click on the **Results** tab. A list of analyses in the Results Repository will appear.
25. In the analysis list page, click on the **Test analysis** link. The analysis details page will appear.
26. In the analysis details page, click on the **Test experiment** link. The experiment details page will appear. You can view the experiment statistics, graphs, metadata, logs, etc. You can also descend in the entity hierarchy and view the results of particular binaries and runs.

7 Compiling BEEN

BEEN is compiled using the Ant build tool. Compilation is done by running Ant's *targets*, which are defined in the `build.xml` build file in the root of the BEEN source code directory.

To run an Ant target, use following command:

```
ant target-name
```

To view help of all Ant targets, use following command:

```
ant -p
```

The most important targets are:

- `all` – compiles BEEN, creates the distribution files and task packages, populates the services' directories in the `data` directory and creates the Javadoc API documentation.
If you want to compile BEEN from scratch, this target is the one you want.
After running this target, the Task Manager, the Host Runtime and services are ready to run. The web interface must be additionally deployed to Tomcat servlet container (see target `deploy`).
- `install` – compiles BEEN, creates the distribution files and task packages and populates the services' directories in the `data` directory.
After running this target, the Task Manager, the Host Runtime and services are ready to run. The web interface must be additionally deployed to Tomcat servlet container (see target `deploy`).
- `build` – compiles Java sources. The compiled classes are placed into the `build` directory.
- `dist` – creates the distribution files, i.e. the `been.jar` file with BEEN Java classes and the task packages. The distribution files are created in the `dist` directory.
- `compile-monitor` – compiles the Load Monitor for the current operating system, if it is supported. See section *Required Software (Part II, 1.1.2)* for compiler requirements.
Compilation of the Load Monitor is needed very rarely, as the binaries for all supported operated systems are distributed with BEEN.
- `compile-detector` – compiles the Detector for the current operating system, if it is supported. See section *Required Software (Part II, 1.1.2)* for compiler requirements.
Compilation of the detectors is needed very rarely, as the binaries for all supported operated systems are distributed with BEEN.
- `clean` – deletes the compiled sources and data used by BEEN's components.
- `dataclean` – deletes the `data` directory used by BEEN components. To populate the directory again with default data of the components, run the `install` target.
- `distclean` – deletes the `dist` directory containing the distribution files and task packages. To create the distribution packages files and task packages again, run the `dist` target.
- `deploy` – deploys the BEEN web user interface to the Tomcat servlet container. Tomcat must be stopped when running this target (see also the `redeploy-with-restart` target). For correct functionality, the `tomcat.dir` property in the `user.properties` file must point to the Tomcat installation directory.

- `redeploy-with-restart` – stops the Tomcat servlet container, deploys the BEEN's web user interface to the Tomcat and starts Tomcat again. For correct functionality, the `tomcat.dir` property in the `user.properties` file must point to the Tomcat installation directory.
- `start-tomcat` – starts Tomcat. For correct functionality, the `tomcat.dir` property in the `user.properties` file must point to the Tomcat installation directory.
- `stop-tomcat` – stops Tomcat. For correct functionality, the `tomcat.dir` property in the `user.properties` file must point to the Tomcat installation directory.
- `javadoc` – generates Javadoc API documentation from the Java sources of BEEN.
- `docs-native` – generates Doxygen API documentation from the C++ sources of detectors and load monitors.
- `test` – runs JUnit²² tests. For correct functionality, the `junit.jar` property in the `user.properties` file must point to a recent JAR file of the JUnit unit testing tool.

The compilation of BEEN can be controlled by setting properties in the `user.properties` file. This file does not exist by default and must be created by the user.

Documentation and default values of supported properties can be found in the `build.properties` file. Do not change this file, but use `user.properties` file to override the settings.

The build properties are:

- `tomcat.dir` – path to the Tomcat installation directory.
- `junit.jar` – path to a JAR file with a recent version of JUnit.
- `use.examples` – this property is defined (with any value), example data will be copied to the services' data directories.
- `copy.task.packages` – if this property is defined (with any value), task packages created during compilation will be automatically copied to the Software Repository data directory.
- `extra.packages` – if you need to automatically copy additional packages to the Software Repository, set this property to a path to the directory with the packages. This is useful when developing benchmarks.

²² <http://junit.sourceforge.net/>

Part III

Extending BEEN

1 Introduction

BEEN architecture has been created with extensibility on mind. BEEN provides support for execution of various tasks in distributed heterogeneous environment and supports two benchmarks (Xampler and RUBiS) out-of-the box.

Support for both Xampler and RUBiS is written using exactly the same facilities that are available to the end users of the environment. Users can write support for new types of benchmarks or add functionality to already existing benchmarking plugins.

Functionality of BEEN can be extended in several ways:

- **Writing custom tasks.** Custom tasks can add support for compilation and execution of additional software in context of presently supported benchmarks.
- **Adding support for new benchmarks.** New benchmarks can be added by writing custom Benchmark Manager plugins.
- **Writing new R scripts for the Results Repository.** By providing new scripts, user can perform better analysis of existing data or add capabilities which allow gathering of data in new format.

To ease development of the extensions, BEEN provides several facilities to aid developer when writing custom tasks and plugins (for example running services with Java remote debugging enabled or extensive logging facilities).

2 Writing Custom Tasks

A *task* is the smallest unit of execution in BEEN. It is similar to an ordinary program or process.

Task's code and data are stored in a BEEN package file. Packages are managed by the Software Repository. When the task is started, its package is downloaded from the Software Repository to the correct host and unpacked by the Host Runtime. Host Runtime automatically creates following directories when executing a task:

- **Task directory** – contains the unpacked contents of the task's package. This directory is deleted after the task finishes.
- **Working directory** – task should use this directory to store data that must be accessible even after it is finished. This directory is deleted only when the context of the task is deleted.
- **Temporary directory** – directory which can be used to store temporary data which are not needed after the task is finished. This directory is deleted after the task finishes.

Task Manager provides checkpoints as a synchronization primitive which can be used by the tasks. Checkpoint can also store data and therefore can be used as a means of communication between tasks.

Checkpoints are name-value pairs, where the name is a `String` and the value is any `Serializable` object. Tasks can set checkpoints to indicate their state (e.g. checkpoint with the name `running` and value `null` indicating that a database has started) and to send information to other tasks (e.g. checkpoint with the name `server-ref` with a value containing an IOR CORBA reference of a remote object). On the other hand, tasks can query values of checkpoints which were set by other tasks. The query can be either non-blocking, or blocking. The task can wait for a blocking checkpoint for a specified amount of time or block indefinitely.

Several checkpoints are automatically set by the Task Manager when task enters specific state (for example when the task starts or terminates).

For more information about tasks and checkpoints see Javadoc documentation for the `Task` and `CheckPoint` classes.

2.1 Packaging

Data belonging to a task are stored in the BEEN package. The package of a task must contain the task's code, any 3rd party libraries used by the task and other data required by the task.

The package is a ZIP file which has to contain metadata files and a `files` directory.

The `files` directory must contain the task's code, libraries and data. When the task starts, contents of this directory is unpacked to the task's `task` directory.

In the root directory of the package, two metadata files must be present:

- **metadata.xml** – metadata file describing a BEEN package. The information stored in this file is used by the Software Repository. The package name is used as the name of the task (e.g. in a task descriptor). Author of the task should also provide a version and a human readable name. The type of the package must be set to `task`.

Example of metadata.xml file:

```

<package>
  <name>example1task</name>
  <version>1.0</version>
  <type>task</type>
  <humanName>Example task</humanName>
</package>

```

For detailed information about contents of the `metadata.xml` file, see section *Packages (Part I, 6.1)*.

- **config.xml** – information from this file is used by the Host Runtime when executing the task. It specifies classpath required by the task and the class which contains `run` method of the task. The classpath entries are relative to the `files` directory of the task package.

Example of config.xml file:

```

<packageConfiguration>
  <java
    classPath=".:example1task.jar:examples/hello.jar"
    mainClass=
      "cz.cuni.mff.been.task.example.task1.Example1Task"
  />
</packageConfiguration>

```

In this example, the class representing the task is the `Example1Task` class (which must extend the `Job` or `Service` class, see *Jobs and Services*). The `example1task.jar` file contains code of the task, and the `hello.jar` is a 3rd party library used by the task.

2.2 Task Class

All tasks are descendants (though not direct – see below) of the `Task` class. The `Task` class provides basic features needed by all tasks:

- **Logging** – tasks can log messages with various levels of importance. All log messages are stored in the central storage, and can later be examined by the user from the web interface. The standard output and error output of the task are automatically captured and stored in the log storage.
- **Task parameters** – parameters for the tasks are provided in the form of task properties. Task properties are name-value pairs, where value can be any `Serializable` object (for example `String`). Since the task's properties are not passed as a command-line arguments, they are not restricted by limitations of command-line on various platforms.
- **Support for checkpoints** – the `Task` class provides several methods to query and set checkpoints and their values.
- **Communication with other components of BEEN** – task can access the `TasksPortInterface` interface via the `Task` class. This interface provides additional means of communication with the Host Runtime and other BEEN components. Queries to the BEEN naming service are also forwarded through this interface (see *Services* for more information about the naming service).

Apache Ant²³ tool is integrated to BEEN and classes from Ant are on the classpath of each task. Tasks can therefore use all of the facilities provided by Ant to perform various build-related

²³ <http://ant.apache.org/>

actions. Special wrappers for some Ant tasks are provided in the `cz.cuni.mff.been.common.anttasks` package. For more details about wrappers provided, see Javadoc documentation of that package.

2.3 Jobs and Services

Each task in the BEEN is either a job or a service. *Job* tasks start, usually perform one specific action and then finish. On the other hand, *services* run for longer time – after start, they register remote interfaces in the naming service provided by BEEN. Other tasks and services than use this remote interface to access information provided by the service and invoke actions. Services have to be stopped explicitly by the user. Services and jobs must extend the `Service` and `Job` class, respectively.

Tasks and services are loaded automatically by BEEN and do not require `main` method.

2.3.1 Jobs

A job is similar to a batch script – that is, it should perform an action and then terminate. Author of the job task needs to extend the `Job` class and override `run` method. Normal return of the `run` method indicates successful finish of the job. To indicate an error, task should throw a `TaskException` exception from the `run` method. The exception message will be automatically logged, its stack trace will be printed and job will be terminated with an exit value indicating an error.

You also have to override the `checkRequiredProperties` method which checks whether all properties required by the task are set.

For an example task, see source code and documentation of the `Example1Task` class.

2.3.2 Services

Services are special tasks that serve as a core components of BEEN. They provide remote interfaces which can be registered in the naming service provided by BEEN. Each service has to extend the `Service` class and has to be included on the BEEN's classpath.

The service has to override the `getName`, `start` and `stop` methods of the `Service` class. The `getName` method must return the name of the service, which is used for lookup of the service in the naming service. The `start` method is called automatically during the service startup. It should initialize the service and register all remote interfaces required by the service. The `stop` method is called when the service is being shut down. It should perform all steps necessary for a clean shutdown.

Registration of the remote interfaces of the `Service` should be done in its constructor. Remote interfaces can be registered with the `addRemoteInterface` method. The remote interface will then be automatically registered with the BEEN naming service and with RMI registry running on the same host as the service. The registered remote interface will be available in the RMI registry on the URL `rmi://hostname:1099/been/service-name/interface-name`.

Every service automatically creates a *control interface*. This remote interface can be used to stop or restart the service.

For more information about the services, see source code and Javadoc documentation of the `ExampleService1` service.

2.4 Existing Tasks

BEEN is distributed with a set of jobs that provide various functionality required for benchmarking. Some jobs are more general (e.g. downloading a package from the Software Repository, or downloading source tree of a software from the Subversion repository), some are more specialized (e.g. support for the Xampler and RUBiS benchmarks).

2.4.1 ant-build

Runs the Apache Ant build tool. For the task to work, Ant has to be installed on the target host.

Task Properties

Name	Required	Description
directory	yes	working directory in which Ant will be executed
target	no	name of the target in the build file which will be called; if omitted, the default target is used
parameters	no	additional parameters that will be passed to Ant

Checkpoints Set

None.

2.4.2 check-and-convert-rubis

Validates results generated by the RUBiS benchmark and converts them to a format suitable for results-collect task.

Task Properties

Name	Required	Description
suite	Yes	Benchmarking suite used
results.role	Yes	String identifying benchmarking role for results upload, e.g. "client 1"
results.paths	Yes	Table containing information about the location of results. For each line, three tab-separated items must be present: binary ID, run ID and the path to the file with raw data produced by the binary and run.

Checkpoints Set

None.

2.4.3 check-and-convert-xampler

Validates results generated by the Xampler benchmark and converts them to a format suitable for results-collect task.

Task Properties

Name	Required	Description
suite	Yes	Benchmarking suite used
results.role	Yes	String identifying benchmarking role for results upload, e.g. "client 1"
results.paths	Yes	Table containing information about the location of results. For each line, three tab-separated items must be present: binary ID, run ID and the path to the file with raw data produced by the binary and run.

Checkpoints Set

None.

2.4.4 cvs-download

Performs checkout from a CVS repository.

Task Properties

Name	Required	Description
repository	yes	URL of the CVS repository
module	yes	name of a CVS module
password	no	password to use for the checkout
revision	no	name of a branch to checkout
data	no	if set, version from this date will be checked out

Checkpoints Set

None.

2.4.5 regex-substitute

Substitutes all matches of a regular expression in a file with a string.

Task Properties

Name	Required	Description
file	yes	file to edit
regex	yes	regular expression
substitution	yes	substitution of the regular expression matches

Checkpoints Set

None.

2.4.6 example1task

Example task which shows how to write custom tasks, use Ant to perform various operations and use 3rd party libraries.

Task Properties

Name	Required	Description
src.file	yes	path to the file which will be copied
dest.file	yes	name of the file's copy

Checkpoints Set

None.

2.4.7 jboss-build

Compiles JBoss from its source.

Task Properties

Name	Required	Description
jboss.dir	yes	path to the JBoss source code

Checkpoints Set

None.

2.4.8 jboss-configure

Configures various settings of JBoss. It must be run before starting JBoss. It edits configuration files of the default server. It supports MySQL datasource configuration only, and it copies the MySQL JDBC connector to the JBoss deployment directory.

Task Properties

Name	Required	Description
jboss.root	yes	path to the JBoss installation directory
jboss.jnp.port	no	port the the JNP service
jboss.webservice.port	no	port the the web service service
jaws.datasource	no	datasource name for JAWS
jaws.typemapping	no	datasource type mapping for JAWS
cmp-jdbc.datasouce	no	datasource name for CMP JDBC
cmp-jdbc.typemapping	no	datasource type mapping for CMP JDBC
naming.call-by-value	no	value of the CallByValue option of the JBoss naming service

Name	Required	Description
<code>use.jboss.web.loader</code>	no	value of the UseJBossWebLoader attribute of the JBoss naming service

Checkpoints Set

None.

2.4.9 `jboss-run`

Starts JBoss.

Task Properties

Name	Required	Description
<code>jboss.dir</code>	yes	path to the JBoss installation directory

Checkpoints Set

Name	Description
<code>running</code>	set when JBoss has successfully started

2.4.10 `jboss-shutdown`

Stops JBoss.

Task Properties

Name	Required	Description
<code>jboss.dir</code>	yes	path to the JBoss installation directory
<code>jboss.jnp.port</code>	no	port number of the JBoss' JNP service; if omitted, default JNP port is used

Checkpoints Set

None.

2.4.11 `jonas-build`

Compile Jonas from its source. Binary of the Tomcat is required for the compilation.

Task Properties

Name	Required	Description
<code>jonas.dir</code>	yes	path to the Jonas source code
<code>tomcat.dir</code>	yes	path to the Tomcat installation directory

Checkpoints Set

None.

2.4.12 jonas-configure

Configures Jonas. It must be run before Jonas starts. Supports the MySQL datasource only.

Task Properties

Name	Required	Description
jonas.root	yes	path to the Jonas installation directory
jrmport	no	port number of the Jonas JRMP service
mysql.datasource.name	no	datasource name in the MySQL.properties file
mysql.datasource.user	no	user name in the MySQL.properties file
mysql.datasource.url	no	URL in the MySQL.properties file
mysql.minconnpool	no	minimum connection pool size in the MySQL.properties file
mysql.maxconnpool	no	maximum connection pool size in the MySQL.properties file
mysql.maxwaittime	no	maximum wait time for threads that didn't fit into the connection pool, set in the MySQL.properties file
http.port	no	port for the Jonas web server

Checkpoints Set

None.

2.4.13 jonas-run

Starts Jonas.

Task Properties

Name	Required	Description
jonas.dir	yes	path to the Jonas installation directory
tomcat.dir	yes	path to the Tomcat installation directory

Checkpoints Set

Name	Description
running	set when Jonas has successfully started

2.4.14 jonas-shutdown

Stops Jonas.

Task Properties

Name	Required	Description
<code>jonas.dir</code>	yes	path to the Jonas installation directory

Checkpoints Set

None.

2.4.15 local-download

Downloads a package from the Software Repository.

Task Properties

Name	Required	Description
<code>rsl</code>	yes	RSL expression identifying the package to download

Checkpoints Set

None.

2.4.16 log-upload

Task Properties

Notifies the Results Repository to retrieve the logs of all remaining tasks of an experiment.

Name	Required	Description
<code>analysis.id</code>	yes	ID of the analysis, for which to upload logs
<code>experiment.id</code>	yes	ID of the experiment, for which to upload logs

Checkpoints Set

None.

2.4.17 mysql-initialize

Initializes a fresh MySQL installation. This initialization is needed before the first start of MySQL.

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory

Checkpoints Set

None.

2.4.18 `mysql-run`

Starts the MySQL database server.

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory
<code>connections.max</code>	no	maximum number of concurrent connections to MySQL

Checkpoints Set

Name	Description
running	set when MySQL has successfully started

2.4.19 `mysql-runcommand`

Runs a command on the MySQL server (e.g. an SQL statement).

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory
<code>user.name</code>	yes	user name under which will the command be called
<code>command</code>	yes	MySQL command that will be executed
<code>database.name</code>	no	name of the database which is the context of the command

Checkpoints Set

None.

2.4.20 `mysql-shutdown`

Shuts down MySQL.

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory

Checkpoints Set

None.

2.4.21 native-cvs-download

Performs checkout from a CVS repository via the CVS command. The CVS client must be installed in the system.

Task Properties

Name	Required	Description
repository	yes	URL of the CVS repository
module	yes	name of a CVS module
password	no	password to use for the checkout
revision	no	name of a branch to checkout
data	no	if set, version from this date will be checked out

Checkpoints Set

None.

2.4.22 omniORB-compile-linux

Compiles omniORB on the Linux platform.

Task Properties

Name	Required	Description
source.dir	yes	path to the directory which contains a subdirectory with the omniORB source
omniORB.root	yes	path to the root directory of omniORB source, relative to the path set in the source.dir task property

Checkpoints Set

None.

2.4.23 omniORB-source-package

Creates a source package of omniORB.

Task Properties

Name	Required	Description
omniORB.source.root	yes	path to the directory with omniORB source

Checkpoints Set

None.

2.4.24 results-collect

Collects results, logs and load information and sends them to the Results Repository. Must be called after the check-and-convert task, which is benchmark specific.

Task Properties

Name	Required	Description
analysis.id	yes	ID of analysis, which is uploading results
experiment.id	yes	ID of experiment, which is uploading results
results.paths	yes	list of directories which contain converted results. Each directory has to be specified on a new line

Checkpoints Set

None.

2.4.25 rubis-build-prepare

Prepares the RUBiS source code for compilation. It sets the correct values of properties in the `build.properties` file and edits the required source files.

Task Properties

Name	Required	Description
rubis.root	yes	path to the RUBiS source code
ejb.server	yes	type of the EJB server to use by RUBiS. Valid values are “jboss” and “jonas”
j2ee	yes	path to a directory that contains J2EE API jar files

Checkpoints Set

None.

2.4.26 rubis-deployer

Deploys RUBiS to an EJB server, i.e. copies the right files to right places. It does not start the EJB server.

Task Properties

Name	Required	Description
rubis.root	yes	path to the RUBiS source code
ejb.server	yes	type of the EJB server to use by RUBiS. Valid values are “jboss” and “jonas”
jboss.root	no ²⁴	path to the JBoss installation directory

²⁴ Required if the `ejb.server` task property is set to “jboss”.

<code>jonas.root</code>	no ²⁵	path to the Jonas installation directory
<code>rubis.benchmark</code>	yes	specifies which one of the several RUBiS benchmark suites will be used. Valid values are “EJB_CMP2.o”, “EJB_EntityBean_id”, “EJB_EntityBean_id_BMP”, “EJB_local_remote”, “MDB”, “EJB_Session_facade” and “EJB_SessionBean”.
<code>database.hostname</code>	no ²⁶	name of the host on which database to be used by RUBiS is running

Checkpoints Set

None.

2.4.27 `rubis-mysql-backup`

Creates backup of the RUBiS database in MySQL to it's working directory. The backup can later be used to quickly restore the default state of the RUBiS database.

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory

Checkpoints Set

None.

2.4.28 `rubis-mysql-initialize`

Initializes the MySQL database for usage with RUBiS. It creates required databases and tables, and fills them with data from a MySQL dump.

Task Properties

Name	Required	Description
<code>mysql.root</code>	yes	path to the MySQL installation directory
<code>dump.root</code>	yes	path to the directory with the MySQL database dump of the RUBiS database

Checkpoints Set

None.

2.4.29 `rubis-mysql-restore`

Restores the RUBiS database in MySQL from a backup.

²⁵ Required if the `ejb.server` task property is set to “jonas”.

²⁶ Required if the `ejb.server` task property is set to “jboss”, because it has to be set in a datasource descriptor that is deployed with RUBiS.

Task Properties

Name	Required	Description
mysql.root	yes	path to the MySQL installation directory
backup.path	yes	path to the directory with the backup of the RUBiS MySQL database

Checkpoints Set

None.

2.4.30 rubis-run

Runs the RUBiS client emulator. This task can work in 2 roles: main and slave. When all slave clients are ready to start, the main client starts them by setting a checkpoint.

Task Properties

Name	Required	Description
rubis.root	yes	path to the RUBiS installation directory
ejb.hostname	yes	hostname of the EJB server
database.hostname	yes	hostname of the database server
run.index	yes	index of the benchmarking run for this execution of the client emulator
main.client	no ²⁷	task ID of the main client. If this task property is set, this client will be in the slave role
clients	no ²⁸	String array of task IDs of the slave clients. If this task property is set, this client will be in the main role
transition.table	no	name of the transition table which will be used
transitions.max	no	maximum number of transitions during one client session
upramp.time	no	length of the up ramp, in milliseconds
upramp.slowdown.factor	no	slowdown factor used during the up ramp
session.time	no	length of the benchmarking session, in milliseconds
downramp.time	no	length of the down ramp, in milliseconds
downramp.slowdown.factor	no	slowdown factor used during the down ramp
client.count	no	number of concurrently emulated clients
debug.level	no	debug level of the client emulator; valid values are 0, 1, 2 or 3

²⁷ Required if the `clients` task property is not set.

²⁸ Required if the `main.client` task property is not set.

Checkpoints Set

Name	Description
<code>ready</code>	client in slave role sets this checkpoint to indicate readiness to start the client emulator. The main client waits for every slave client to set this checkpoint and then it sets the <code>start</code> checkpoint
<code>start</code>	task in master role sets this checkpoint to tell all slave clients to start the client emulator. Every slave client sets the <code>ready</code> when it's ready to start the client emulator, and then waits for the main client to set this checkpoint

2.4.31 `svn-checkout`

Performs a checkout from a Subversion repository. If no revision is specified, then it will checkout the HEAD.

Task Properties

Name	Required	Description
<code>url</code>	yes	URL of the SVN repository
<code>revision.number</code>	no	if this task property is set, then the revision number in it's value will be checked out from the SVN repository
<code>revision.time</code>	no	if this task property is set, then the revision from the time specified in it's value will be downloaded from the SVN repository. The time must be specified in a correct format for your locale

Checkpoints Set

None.

2.4.32 `xampler-execute`

Runs the Xampler benchmark. It can run the server or the client component of the benchmark.

Task Properties

Name	Required	Description
<code>omniorb.root</code>	yes	path to the installation directory of omniORB
<code>omniorb.root</code>	yes	path to the installation directory of Xampler
<code>xampler.suite.path</code>	yes	path to a suite of Xampler. The path is relative to the Xampler installation directory
<code>xampler.role</code>	yes	if set to “server”, the server component of Xampler will be run. If set to “client”, the client component of Xampler will be run
<code>server.tid</code>	no ²⁹	the task ID of the <code>xampler-execute</code> task that runs the server component of Xampler

²⁹ Required if the the `xampler.role` task property is set to “client”.

Name	Required	Description
xampler.server.params	no	runtime parameters of the server component of Xampler
xampler.client.params	no	runtime parameters of the client component of Xampler

Checkpoints Set

Name	Description
server.started	set by the server component of Xampler when the server successfully starts. The clients wait until this checkpoint is set by the server

2.4.33 xampler-compile-linux

Compiles Xampler with omniORB on the Linux platform.

Task Properties

Name	Required	Description
xampler.dir	yes	path to the directory which contains an archive with the sources of Xampler
omniorb.root	yes	path to the installation directory of omniORB
omniorb.version	yes	version of omniORB which is used for the compilation.

Checkpoints Set

None.

2.4.34 log-tester

Tests the log storage by generating log messages or standard and error output.

Task Properties

Name	Required	Description
message.count	yes	how many messages will be generated
message.delay	yes	time delay between messages, in milliseconds
action	yes	if value is "log", log messages will be generated. If the value is "output", standard and error output will be generated

Checkpoints Set

None.

2.4.35 testworker

This task can be set to do various example works. It is used for testing purposes. You have to choose which actions it should perform. It's designed to cooperate with other running testworker tasks.

Task Properties

Name	Required	Description
<code>do.checkpoint.value</code>	no	if set to “true” or “yes”, it retrieves a value of a checkpoint. The value must be of the <code>String[2]</code> type. The name of the checkpoint must be specified in the <code>checkpoint.name</code> task property. Task ID of a task that sets the checkpoint must be specified in the <code>checkpoint.task</code> task property
<code>do.checkpoint.set</code>	no	if set to “true” or “yes”, it sets a checkpoint with a value. The value will be { "BEEN", "DSRG" }. The name of the checkpoint must be specified in the <code>checkpoint.name</code> task property
<code>do.checkpoint.block</code>	no	if set to “true” or “yes”, it retrieves a value of a checkpoint in a similar way as when the <code>do.checkpoint.value</code> task property is set to “true” or “yes”, but the query for the checkpoint value is blocking. The <code>checkpoint.name</code> and <code>checkpoint.task</code> task properties must be also set
<code>do.checkpoint.value</code>	no	if set to “true” or “yes”, it retrieves the value of a task property. The value must be of the <code>String[2]</code> type. Name of the task property must be specified in the <code>property.name</code> task property
<code>do.wait</code>	no	if set to “true” or “yes”, the task sleeps for the amount of seconds set in the <code>wait.time</code> task property
<code>wait.time</code>	no ³⁰	number of seconds to sleep

Checkpoints Set

None.

2.4.36 create-package-metadata

Creates metadata file for a BEEN package in its working directory.

Task Properties

Name	Required	Description
<code>package-name</code>	no	name of the package
<code>package-human-name</code>	no	human readable, or a more descriptive package name
<code>version</code>	no	version of the package
<code>hardware-platforms</code>	no	hardware platforms supported by the package, separated by spaces
<code>software-platforms</code>	no	software platforms supported by the package, separated by spaces
<code>type</code>	no	type of the package, valid values are “binary”, “source”, “data” and “task”

³⁰ Required if the `do.wait` task property is set to “true” or “yes”.

Checkpoints Set

None.

2.4.37 localupload

Creates a BEEN package from a directory, and uploads it to the Software Repository.

Task Properties

Name	Required	Description
dir	yes	path to the directory whose contents will be packed in a BEEN package
metadata.file	yes	path to the metadata file describing the BEEN package
use.config	no	if set to “true” or “yes”, a config.xml file describing a BEEN task will be a part of the BEEN package. The path to the config.xml file must be set in the config.file task property
config.file	no ³¹	path to the config.xml file for a task package

Checkpoints Set

None.

³¹ Required if the use.config task property is set to “true” or “yes”.

3 Extending Benchmark Manager

The Benchmark Manager has a plugin-based architecture. It allows extending the Benchmark Manager, so it can support any benchmark, for which a plugin is available.

The Benchmark Manager plugin is a set of Java classes, accompanied by metadata in XML-based format. The plugin has three different responsibilities:

1. **Creating experiment configuration** – different benchmarks operate in distinct ways which have to be properly configured. It's impossible to create a common configuration mechanism that would be sufficient for every benchmark. Thus, the plugin must create a user interface suitable for configuration of the benchmark. The *configurator* specifies several user interface screens which are presented to the user in sequence, and processes the input in the displayed screens. User's input in a screen can influence the structure of following screens. The output of the configurator is the configuration of the experiment, which is stored in experiment metadata.
2. **Scheduling of tasks** – benchmarks require a sequence of tasks to perform low level operations – compile the benchmarked software, deploy it, run the benchmark, etc. The benchmark manager plugin must ensure that the tasks in the sequence will be executed in the correct order, on specified hosts, and with the required parameters. The part of the benchmark manager plugin responsible for scheduling of tasks is the *task generator*. The experiment configuration is used to generate the task sequence.
3. **Determining which versions of software will be benchmarked during a regression analysis** – a regression analysis benchmarks software in regular intervals. The part of the benchmark plugin, which finds versions of the benchmarked software available at the scheduled benchmarks times is called a *version provider*. It must deal with problems like that the benchmarked software might not have changed between some benchmarked times, or BEEN may not be running in some of those times.

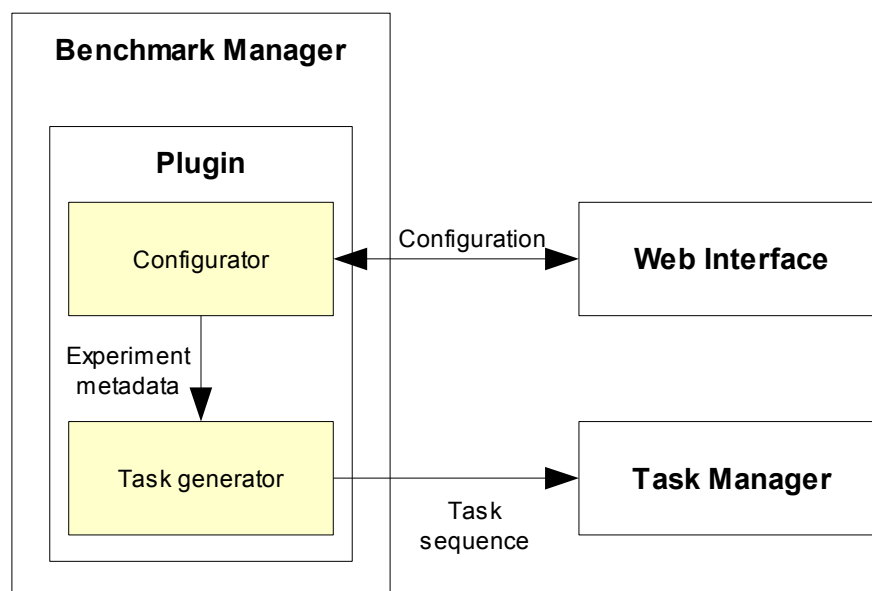


Diagram 2: Benchmark plugin workflow.

3.1 Packaging

Benchmark manager plugins are distributed as JAR files. The JAR file must contain an XML plugin descriptor and Java code implementing the configurator, task generator and version provider.

The name of plugin descriptor file must be `plugin.xml` and it must be located in the root of the JAR file. Benchmark Manager plugins are managed by the *Java Plugin Framework (JPF)*, <http://jpf.sourceforge.net/>) – the `plugin.xml` file is a JPF plugin manifest. BEEN does not use all features available in JPF, and thus not all elements of the plugin manifest are used by BEEN. We will describe the structure of the plugin descriptor on the example (plugin descriptor of a testing plugin, included in BEEN):

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.5"
    "http://jpf.sourceforge.net/plugin_0_5.dtd">

<plugin id="cz.cuni.mff.been.plugins.simpletest" version="0.1">
  <doc>
    <doc-text>
      <![CDATA[
        Simple benchmark plugin test.<br />
        This plugin runs 4 tasks:<br />
        ...
      ]]>
    </doc-text>
  </doc>

  <attributes>
    <attribute id="pluginName" value="Simple test" />
  </attributes>

  <requires>
    <import plugin-id="cz.cuni.mff.been.benchmarkmanager.core" />
  </requires>

  <runtime>
    <library type="code" path="code/" id="core" />
  </runtime>

  <extension id="RUBiS"
    plugin-id="cz.cuni.mff.been.benchmarkmanager.core"
    point-id="benchmark"
  >
    <parameter id="generator"
      value="cz.cuni.mff.been.benchmarkmanager.plugins
        .simpletest.SimpleTestGenerator"
    />
    <parameter id="configurator"
      value="cz.cuni.mff.been.benchmarkmanager.plugins
        .simpletest.SimpleTestConfigurator"/>
    <parameter id="versionprovider"
      value="cz.cuni.mff.been.benchmarkmanager.plugins
        .simpletest.SimpleTestVersionProvider"/>
  </extension>
```

</plugin>

The DOCTYPE of the XML document is of a plugin manifest for JPF. The `plugin` element is the root element. The `id` attribute of the `plugin` element specifies a unique identification of the benchmark plugin. The best way to create the `id` is to use a naming scheme similar to the one used in Java's packages. There may not be two plugins with same `id` installed. The `version` attribute is mandatory, though it is not used by BEEN.

Each plugin can contain a brief description. It is displayed to the user during the selection of a plugin for an experiment. The description should be inside the `doc-text` sub-element of the `doc` element. You can use XHTML tags in the description. Usage of the XHTML tags requires you to put the description into a CDATA section, or escape the tags using XML entities.

The plugin descriptor should contain the name of the plugin. The name is displayed to the user as the name of the plugin during selection of a plugin for an experiment. Use the `attribute` element with the `pluginName` `id` to specify the plugin name. If the plugin name isn't specified, then the plugin `id` will be displayed in the user interface as the name of the plugin.

The `requires` element is a mandatory element in all plugin descriptors, required by JPF. In essence, all Benchmark Manager plugins extend a core plugin, which is distributed with the Benchmark Manager. You don't need to understand the meaning of this element, you can copy & paste it from this example plugin descriptor.

The `library` element specifies location of the plugin code. You must set the `type` attribute of the `library` element to `code`. The `path` attribute specifies the location of the plugin code relative to the root directory of the plugin's JAR file. If the path is a directory name, it must end with a "/" character. In the case of the plugin above, the code is in the `code` subdirectory of the JAR file. The `id` attribute of the `library` element is not used by BEEN, but the suggested value is the `id` of your plugin.

The `extension` element is used to specify which classes implement the configurator, the task generator and the version provider. The `id` attribute of the `extension` element is not used by BEEN, but the suggested value is the `id` of your plugin. Also set the `parameter` elements for configurator, generator and version provider to the classes implementing them in your plugin.

3.2 Experiment Metadata

Configuration of every benchmarking experiment is stored in experiment metadata, implemented by the `ExperimentMetadata` class.

Information stored in the experiment metadata:

- **Benchmark properties** – standard Java Properties, i.e. name-value pairs. They are used to store most of the configuration of the experiment (e.g. property with the name `ejb.server` and value "JBoss").
- **Host roles** – hosts used during benchmarking serve different purposes (e.g. several hosts are clients and one host is the server). Each of these purposes is called a *host role*. Each benchmark requires hosts in distinct roles, with every role containing a specified number of hosts. Hosts in roles are specified using RSL. For more information about RSL, see section *Restriction Specification Language (RSL) (Part I, 11)*.

There are two types of roles, *standard roles* and *benchmarking roles*. The RSL specification of hosts in benchmarking roles is resolved to specific hosts by the Benchmark Manager, and user selects the required number of hosts from those hosts.

Then tasks running in the benchmark roles are run on the exact hosts that were selected by the user. The RSL in the standard roles isn't resolved by the Benchmark Manager, but is stored in the task Descriptors of the tasks of this role. The Task Manager then resolves the RSL when he starts the tasks. The postponing of the RSL resolution in the standard roles is used to allow the Task Manager to assign hosts most suitable or available at the moment of the task's start. This is used for roles that compile software.

- **Run count** – number of benchmark runs that have to be performed.
- **Binary count** – number of software binaries that have to be benchmarked during the experiment.

Consult the Javadoc documentation of the `ExperimentMetadata` class for API to the experiment metadata.

3.3 Configurator

Configurator's task is to configure a benchmarking experiment via user's input in a user interface created by the configurator. The interface of the configuration is specific to the used benchmark. The user is presented a sequence of configuration screens. The configurator is a finite state automaton, which processes the user's input in the previous screen, changes its state accordingly (e.g. by remembering the values set by the user), and creates the next screen which will be shown to the user. The sequence of the screens, or their structure, can change based on user's input (e.g. selection of benchmarked software is followed by a configuration screen for the selected software). The configurator must also create screens when the user is moving backwards in the configuration screen sequence.

Technically, the configurator is a class extending the `Configurator` class. The `Configurator` class provides access to experiment metadata and other utility methods, and abstract methods that must be overridden by the plugin's configurator. These methods return the first configuration screen, the next screen and the previous screen. Returning `null` from any of these methods means that the configuration is finished. The methods for creating the next and previous screen receive the previous screen that was presented to the user as a parameter and the user's input can be retrieved from it.

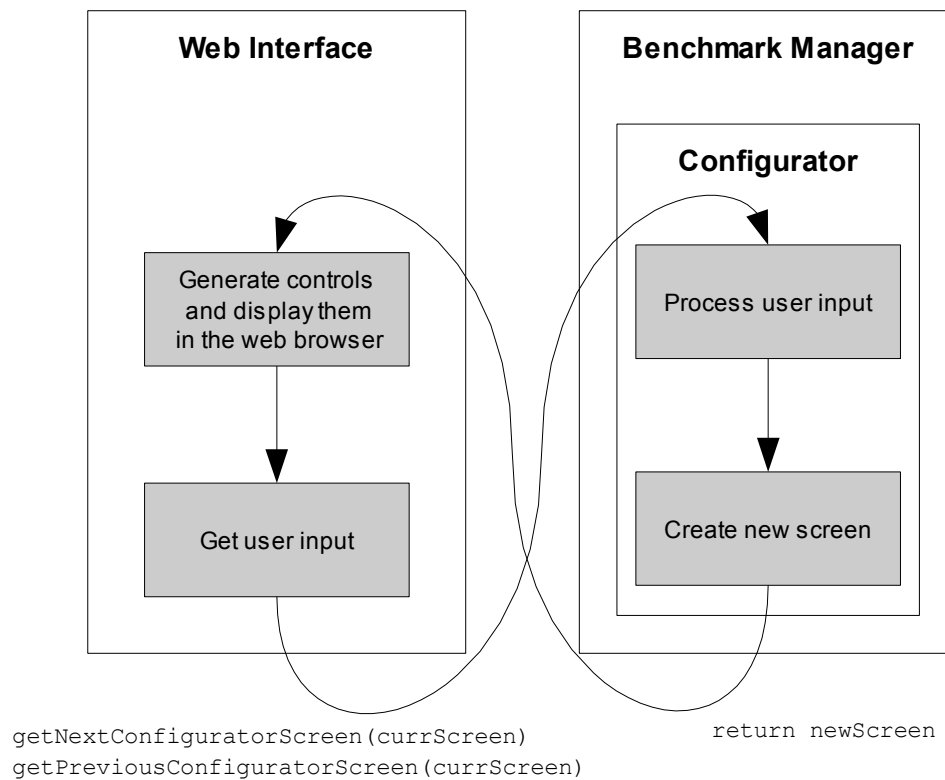


Diagram 3: Cooperation of the web interface and the configurator.

Each configuration screen is composed of various user interface elements (radio buttons, text input boxes, etc.). The screen is defined by the configurator via composition of instances of classes representing the interface elements. The screen's definition is then sent to the user interface component of BEEN, which analyses it and displays the user interface elements. When the user proceeds to the next screen, his input in the previous screen is stored in the same classes which are in turn sent to the configurator for further processing.

The screens are represented by instances of the `Screen` class, and identified by a *Screen ID*, which is assigned by the configurator. The configurator detects the previous screen by its *Screen ID* and thus correctly processes the user's input. The screen is divided into sections – instances of `Section` class, which have a heading, description and may contain various controls. Supported control classes are:

- **RadioWithSections** – represents radio buttons with each of them containing a `Section`.
- **Static Text** – displays static text to the user. The text cannot be modified.
- **Input** – text input. Can be of several sizes, and the user's input can be validated by a custom validator.
- **Select** – a list of options. User can select one item from the list.
- **MultiSelect** – a list of options. User can select several items from the list.
- **Role** – input element for entering RSL restrictions for a host role. It checks the syntactic validity of the entered RSL, and can display RSL syntax help.

Example – creating a simple screen:

```

Input hostCountInput = new Input(
    "Host count",           // text input label
    "1",                   // default value
    Input.Size.SMALL,      // input size

```

```

    intValidator // integer validator
);

Option[] options = new Option[] {
    new Option("1", "Option 1"), // "1" is the option's ID
                                // "Option 1" is the label
                                // of the option
    new Option("2", "Option 2"),
};

Select select = new Select(
    "Misc options", // select label
    options,        // select options
    0,              // index of the option selected
);                // by default

Section section = new Section(
    new Item[] { // controls in the section
        hostCountInput,
        select
    },
    "Sample section", // section label
    "Simple example section" // section description section
);

return new Screen(
    new SID(1), // screen ID
    new Section[] { section } // sections in the screen
);

```

Example – processing input from the screen:

```

Section section = screen.getSections()[0];
Input input = (Input) section.getItems()[0];
int hostCount = Integer.valueOf(input.getValue());
Select select = (Select) section.getItems()[1];
String option = exclusivitySelect.getSelectedId();

```

For further details, see the Javadoc documentation of the `Configurator` class.

3.4 Task Generator

The task generator creates a sequence of tasks for the execution of one benchmark experiment. The sequence is created in accordance to the experiment configuration stored in experiment metadata.

The task generator creates the task descriptors of the required tasks and stores them in a task sequence, which is submitted to the Task Manager after the task generator finishes. The task sequence isn't a flat list of tasks, but a hierarchical data structure. Its structure respects the hierarchy of a benchmark experiment, which can contain several binaries, which in turn can contain several runs. Thus, some tasks are associated with the experiment (e.g. initialization of a database used by all binaries and runs), some tasks are associated with a binary (e.g. compilation of the binary), and some tasks with a run (e.g. execution and measurement of the run).

The task generator of a benchmark plugin must extend the `TaskGenerator` class. The `TaskGenerator` class handles communication with other BEEN components, storage of

metadata and presents the abstract method `generate()` that must be overridden.

A task generator can log messages with several level of importance. These log messages are stored as log messages of the Benchmark Manager. Experiment metadata can be retrieved by the `getExperiment()` method. The task generator can get a reference to the Software Repository, Host Manager and Results Repository. Finding packages in the Software Repository is simplified by several helper methods.

Creation of task descriptors is simplified. The `createTaskDescriptor(String taskName)` method creates a task descriptor which will run a task with the given task name. The task will run on the host belonging to the active host role specified by the last call of the `setActiveRole(String roleName)`. If several hosts belong to the active host role, you can specify the host's index in the `createTaskDescriptor()` method.

The `createTaskDescriptor()` method doesn't automatically add new task descriptor to the task sequence. That is accomplished with `addTask(TaskDescriptor)` method. The tasks need to be associated with their binaries and runs. They are associated with the last opened binary and run. Binaries are opened and closed via the `createNewBinary()` and `closeBinary()` methods, respectively. Runs are opened and closed via the `createNewRun()` and `closeRun()` methods, respectively. If no run is currently open, the task is associated only with the last open binary (e.g. a task that compiles the binary). If no binary is currently open, the task is associated only with the experiment (e.g. initialization of database which will be used by all binaries and runs).

The correct order of execution of the tasks is guaranteed by task dependencies. The dependencies can specify that a task will be started after a certain checkpoint is set by a specific task. Standard checkpoints are set automatically by BEEN when tasks start and finish (see the Javadoc documentation of the `Task` class). Proper usage of dependencies can make some tasks run in parallel (on the same host, or on different hosts), and other tasks in a sequence. The `TaskGenerator` class provides wrapper methods for creating dependencies.

For further details, see the Javadoc documentation of the `TaskGenerator` class.

Example – task generation:

```
// NOTE: tasks used in this example task generator don't exist
public class ExampleTaskGenerator extends TaskGenerator {

    public void generate() throws GeneratorException {
        setActiveRole("client");

        // Get a benchmark property.
        String timeout = getExperiment().getProperty("timeout");

        // Open a new binary.
        CreateNewBinary();
        // Created task will be associated with currently opened
        // binary.
        TaskDescriptor taskForBinary = createTaskDescriptor(
            "init-binary"
        );
        addTask(taskForBinary);

        TaskDescriptor previousRun;
        // Create tasks for every run.
        for (int i = 0; i < getDefaultRunCount(); i++) {
            // Open a new run.
```

```

CreateNewRun();

TaskDescriptor taskForRun = createTaskDescriptor(
    "run-benchmark"
);
// Set a task property.
taskForRun.addTaskProperties("timeout", timeout);

// We want the tasks of all runs to execute in a sequence,
// and after the first binary initialization task is
// finished.
if (i == 0) {
    // Add dependency on the succesfull finish of the
    // binariy's initialization task.
    taskForRun.addDependencyCheckpoint(
        createSuccessDepenency(taskForBinary.getTaskTid())
    );
} else {
    // Add dependency on the succesfull finish of the
    // previous run's task.
    taskForRun.addDependencyCheckpoint(
        createSuccessDepenency(previousRun.getTaskTid())
    );
}

closeRun();
previousRun = taskForRun;
}

closeBinary();
}
}

```

3.5 Version Provider

Regression benchmarks are run in a user-defined schedule. When the time of a new regression benchmark comes, the Benchmark Manager must determine which versions of the software have to be benchmarked. For example, if the software hasn't changed since the last benchmarking, it's unnecessary to benchmark it again. Also, if BEEN has been turned off for some time, several versions might be needed to be benchmarked. Finding which versions have to be benchmarked is the job of the version provider.

The version provider finds the versions using the information about the regression analysis schedule, last run of the regression benchmark of the analysis and from the time and date interval during which the regression benchmarks of the analysis should be done.

4 Extending Results Repository

Results Repository can be extended to support additional benchmarks and statistical processing of their results. As mentioned in section *Results Collection (Part I, 10.1)*, the results collection process consists of data validation, conversion of the data into a common format and upload of the resulting files to the Results Repository. Validation and conversion have to be implemented as a separate tasks for each benchmark. Data upload is generic operation and is common for all benchmarks. To process results of a new benchmark, custom R scripts must be written.

4.1 Validation and Conversion Tasks

After the benchmark run is finished, tasks that validate and convert results must be scheduled to run on the hosts participating in the experiment. Writing such task consists of creating a class derived from abstract class `CheckAndConvertTask`.

The most important method of the class is the `checkAndConvertOneRun` method, which should take the raw data generated by the benchmark, validate its integrity and convert results into one or more tables in the format supported by the Results Repository. Output format is a simple text file which contains one row of the table on each line with columns of the table separated by the tab characters. For each table created, `notifyTableCreation` method has to be called to let the Results Repository know about the table and its validity. If all tables created during the data conversion are valid, `checkAndConvertOneRun` should return `true`, otherwise it should return `false`.

If the data conversion fails (i.e. the `checkAndConvertOneRun` method returns `false`), `getCrashList` method is called automatically. This method should return list of all files that the task was unable to convert. Those files will be uploaded to the Results Repository as raw data.

For more details, see Javadoc documentation for the `CheckAndConvertTask` class.

4.2 R Script Packages

The Results Repository can be further extended via R packages. These packages are ZIP files which contain R source scripts. Once the package is uploaded via the web user interface, it is loaded into the Results Repository and sourced by R for immediate usage. The functions from the scripts contained in the package can be used then in the callback and invalidation scripts. The package is identified by its filename.

Each package contains following files:

- **description** – file containing one line describing the package. This description is displayed in the web user interface.
- ***.r** – all files with the `.r` extension contain source code of the R scripts. These files are all loaded by R and all functions they contain can later be used either by other R scripts or as a callback functions. To avoid conflicts in function names, script authors are encouraged to use naming convention: `packagename.functionname`, e.g. `Xampler.generateAnalysisRegressionGraph`.
- ***.hlp** – contain documentation on the functions provided in the package. Each

function.hlp file is assumed to contain documentation of the function with name *function*. Function names and file names are case-sensitive. Each help file has to contain function signature on the first line. Rest of the file should be valid HTML with the description of the parameters and return value of the function.

4.3 Writing R Functions

By default, the Results Repository contains package called `base`. This package contains functions for reading contents of the results database, generic graph drawing, etc.

Functions that are used as callbacks for scripts have to accept following parameters:

- **Run callbacks:**

```
functionName(aid, eid, bid, rid, valid, ...)
```

- **Complete and incomplete binary callbacks:**

```
functionName(aid, eid, bid, valid, ...)
```

- **Complete and incomplete experiment callbacks:**

```
functionName(aid, eid, valid, ...)
```

All arguments specified above are required; function author may use any arguments in place of the ellipsis. Required arguments are filled automatically by Java. For example, an experiment callback function implemented in R can have following signature:

```
xampler.generateExperimentGraph <- function(aid, eid, valid =  
  TRUE, color = "blue", statistic = "mean")
```

In the user interface, where experiment callback is entered, you can call it using:

```
xampler.generateExperimentGraph("red", "median")
```

Documentation files for the functions should describe format as seen by the user – that is without the implied required parameters.

5 Debugging BEEN

Debugging BEEN is not an easy task because its distributed nature and the fact that almost every part of BEEN runs in separate Java Virtual Machine. However, BEEN contains several function to aid with this task.

5.1 Debugging Host Runtime and Task Manager

The Host Runtime and the Task manager are quite easy to debug, because they are not tasks, but normal Java classes. The Host Runtime is run by executing the `main` method of the `HostRuntimeRunner` class; the Task manager is run by executing the `main` method of the `TaskManagerRunner` class.

You can use conventional debugging facilities of the Java platform of your favorite IDE to attach a debugger to the Host Runtime or the Task Manager and trace the execution, set breakpoints, inspect variables, etc.

5.2 Debugging Tasks

As the tasks run in separate JVM, they are not easy to debug. In fact, the easiest method of debugging them is to use the logging facilities provided by the Host Runtime and the Task Manager.

For the debug messages, use the log level `DEBUG` (i.e. in the task, write the messages using the `Task.logDebug` method). For detailed trace messages, use the log level `TRACE` (i.e. in the task, write the messages using the `Task.logTrace` method)

By default, messages at the `TRACE` and `DEBUG` level will be dropped by the Task Manager. You can override this setting by passing a log level name (i.e. “`DEBUG`”) to the Task Manager start-up script. Messages with log level lower than the set one will be dropped.

Windows example:

```
taskmanager.bat DEBUG
```

Linux example:

```
./taskmanager.sh DEBUG
```

For the description of the available log levels, see section *Task Manager (Part I, 5)*.

5.3 Debugging Services

The services can be executed with parameters which allow connecting of the remote debugger. To enable remote debugging of the Services, follow these steps:

1. In the web interface, click the **Configuration** tab.
2. Check the **Debugging options** checkbox and click **Save**.
3. Click the **Services** tab.
4. Run the service you want to debug remotely using the **Start (remote debug)** button. The Host Runtime will pass additional parameters to the JVM in which the service

executes:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,  
suspend=y,server=y
```

5. Attach a debugger to the service's JVM.

5.4 Debugging Load Monitor

You can enable debug messages from the Load Monitor by setting the environment variable `BEEN_HOSTRUNTIME_DEBUG` to “true”.

Part IV

Appendices

Appendix A: Objects and Properties of the Host

Following table list all objects and properties (except user-defined ones, of course) that can be found in the host database. For each property or object table contains its full path, type (for properties) and short description of the object or property value.

Name	Type	Description
adapters	integer	Number of network adapters installed on the host.
aliases	integer	Number of software aliases present on the host.
applications	integer	Number of the applications detected on the host
detector	string	Identification string of the detector library that collected data for this host.
drives	integer	Number of disk drives installed on the host.
checkdate	string	Date of the data collection (format is YYYY/MM/DD).
checktime	string	Time of the data collection (format is hh:mm:ss).
memberof	list	List all groups this host is member of.
name	string	Canonical name of the host on the network.
processors	integer	Number of processors detected on the host.
adapter(i)		Object which stores details about on network adapter or interface.
adapter(i).mac	string	Hardware address of the network adapter.
adapter(i).name	string	Name of the network adapter or interface.
adapter(i).type	string	Name of the communication protocol used by the adapter.
adapter(i).vendor	string	Name of the vendor of the adapter.
alias(i)		Object which stores details about one software alias.
alias(i).alias	string	Name of the software alias.
alias(i).name	string	Name of the application this alias represents.
alias(i).vendor	string	Name of the vendor of the application this alias represents.
alias(i).version	version	Version of the application this alias represents.
application(i)		Object which stores details about one application.
application(i).name	string	Name of the application.

Name	Type	Description
<code>application(i).vendor</code>	string	Name of the vendor of the application.
<code>application(i).version</code>	version	Version of the application.
<code>beendisk</code>		Object which stores data about the drive on which BEEN is installed.
<code>beendisk.beenhome</code>	string	Full path to the installation directory of the BEEN.
<code>beendisk.freespace</code>	integer	Size of the free space (in bytes) on the drive on which BEEN is installed.
<code>beendisk.size</code>	integer	Total size (in bytes) of the drive on which BEEN is installed.
<code>drive(i)</code>		Object which stores details about one disk drive installed on the host.
<code>drive(i).device</code>	string	Name of the device assigned to the drive by the OS.
<code>drive(i).media</code>	string	Media type this drive accepts (e.g. CD-ROM).
<code>drive(i).model</code>	string	Model name of the drive.
<code>drive(i).partitions</code>	integer	Number of partitions detected for the drive.
<code>drive(i).size</code>	integer	Total size of the drive in bytes.
<code>drive(i).partition(j)</code>		Object which stores details about one partition on the parent drive.
<code>drive(i).partition(j).device</code>	string	Name of the device assigned to the partition by the OS.
<code>drive(i).partition(j).filesystem</code>	string	Name of the file system used on the partition.
<code>drive(i).partition(j).freespace</code>	integer	Free space available on the partition in bytes.
<code>drive(i).partition(j).name</code>	string	Mount point (or disk name on Windows).
<code>drive(i).partition(j).size</code>	integer	Total size of the partition in bytes.
<code>java</code>		Object which stores details about Java installed on the host.
<code>java.runtime</code>	string	Name of the Java runtime installed on the host.
<code>java.runtimever</code>	version	Version of the Java runtime.
<code>java.specification</code>	version	Version of the Java specification JVM conforms to.
<code>java.vendor</code>	string	Name of the vendor of the Java.
<code>java.version</code>	version	Version of the Java installed on the host.
<code>java.vmvendor</code>	string	Vendor of the JVM installed on the host.

Name	Type	Description
java.vmversion	version	Version of the JVM installed on the host.
memory		Object which stores details about memory on the host.
memory.pagefile	integer	Total size of all paging files in bytes.
memory.physical	integer	Total size of RAM in bytes.
memory.swap	integer	Total size of the swap space in use in bytes.
memory.virtual	integer	Size of the virtual memory available to each process.
os		Object which stores details about the operating system.
os.arch	string	Identification of the processor architecture for which OS has been compiled (e.g. <i>x86_64</i>).
os.family	string	Operating system family (can be one of <i>Windows</i> , <i>Linux</i> , <i>Solaris</i> or <i>Other</i>).
os.name	string	Name of the operating system.
os.vendor	string	Name of the system vendor.
os.windows		Object which stores details about the Windows operating system.
os.windows.build	string	Build identification string.
os.windows.encryption	integer	Encryption strength in bits provided by the OS.
os.windows.sp	version	Version of the service pack installed.
os.windows.sysdir	string	Path to the system directory of the OS.
os.windows.version	version	Version of the Windows operating system.
os.windows.windir	string	Path to the installation directory of the Windows.
os.linux		Object which stores details about Linux operating system. Present only if host runs on Linux.
os.linux.distribution	string	Name of the Linux distribution (e.g. <i>Fedora Core</i>).
os.linux.distversion	version	Version of the Linux distribution (e.g. <i>5</i>).
os.linux.kernelversion	version	Version of the kernel.
os.linux.osrelease	string	Linux release string.
os.linux.osversion	string	Linux version string.
os.other		Object which stores details operating system which does not have native detector library support.
os.other.version	version	Version of the OS as reported by the Java.
os.solaris		Object which stores details about the Solaris OS.

Name	Type	Description
<code>processor(i)</code>		Object which stores details about the processor.
<code>processor(i).cache</code>	integer	Size of the built-in L2 cache of the processor.
<code>processor(i).model</code>	string	Model name of the processor.
<code>processor(i).speed</code>	integer	Speed of the processor in Hz.
<code>processor(i).vendor</code>	string	Name of the vendor of the processor.
<code>user</code>		Object which contains all user-defined objects and properties.

Child objects of the `os` object depend on the operating system running on the host. Only one of `linux`, `windows`, `other` or `solaris` is present at any time.

All indexed object (like drives or partitions) use placeholder indices *i* and *j* instead of numbers. These indices have to be replaced by the correct number in the real-world use.

Appendix B: Results Repository Metadata

Following tables describe entity metadata, that can be displayed in the web user interface. All metadata values are strings.

1.1 Analysis Metadata

Name	Description
Name	analysis name
Type	analysis type, either <code>comparison</code> or <code>regression</code>
Comment	human readable comment
Created on	date and time when the analysis was created
Valid	true if at least one contained experiment is valid; false otherwise
Experiments	number of contained experiments
Valid experiments	number of experiments that contain at least one valid binary
Complete experiments	number of complete experiments in analysis

1.2 Experiment Metadata

Name	Description
Name	experiment name
Comment	human readable comment
Created on	date and time when the experiment was created
Plugin id	plugin ID
Plugin name	name of the plugin used when performing the experiment
Context name	name of the context associated with experiment
Valid	true if at least one contained binary is valid; false otherwise
Complete	true if all contained binaries has uploaded results; false otherwise
Binaries	number of contained binaries
Valid binaries	number of valid binaries
Complete binaries	number of complete binaries
Expected binaries	number of binaries that are expected to upload results
Expected runs per binary	number of runs per each binary that are expected to upload results
Samples per run count	number of samples that each run is expected to contain (benchmark-specific, optional)

1.3 Binary Metadata

Name	Description
Name	binary name
Comment	human readable comment
Created on	date and time when the binary was created
Valid	true if at least one uploaded run is valid; false otherwise
Complete	true if all runs have uploaded results; false otherwise
Runs	number of contained runs
Valid runs	number of valid runs
Expected runs	number of runs that are expected to upload results
Last valid run idx	index of next valid run upon results upload, used internally by the Results Repository
Last crashed run idx	index of next crashed run upon results upload, used internally by the Results Repository

1.4 Run Metadata

Name	Description
Valid	true if results of run are valid; false otherwise
Complete	true if run is completely uploaded; false otherwise

Appendix C: RSL Grammar

The grammar presented here is a simplified version of the grammar used to parse RSL and serves for informational purposes only. The “real” grammar with syntactic look-ahead specification and embedded code for construction of the RSL parser can be found in `Parser.jj` file in BEEN sources. The RSL parser is generated in build time using javacc³² tool.

The RSL grammar is written in BNF-like syntax. Nonterminals are printed in italic, terminals are written in quotes. Alternatives are separated by “|” character. Characters “?”, “*” and “+” denote a possible repetition (zero or one time, zero or more times, one or more times). Parentheses are used for grouping. Expressions *digit*, *digits*, *alpha*, *alphas*, *alphanum* and *alphanums*, *not-slash* and *not-quote* are used in obvious sense.

RSL expression is a stream of tokens. Whitespace (sequence of tabs, spaces, CR or LF characters) between tokens is skipped. The tokens are:

```
package-type := "source" | "binary" | "task" | "data"
long          := digits ("K" | "M" | "G" | "T" | "P")? alphas
version       := digit (alphanum | "_" | "-")*
               "." (alphanum | "_" | "-")+
               ("." (alphanum | "_" | "-")+)*
date          := digit digit digit digit      # year
               "-" digit digit                # month
               (
                 "-" digit digit              # day
                 (
                   "T" digit digit            # hours
                   ":" digit digit            # minutes
                   (
                     ":" digit digit          # seconds
                     (
                       "." digit             # decimal fraction
                     )?                      # of a second
                   )?
                 )?
                 (
                   # time zone
                   "Z"                       # designator
                   | ("+" | "-") digit digit ":" digit digit
                 )
               )?
             )?
pattern       := "/" (non-slash | "\"/") "/" "i"?
string        := "\"" (non-quote | "\"") "\""
property-path := (alpha | "_") (alphanum | "_")*
               "." ((alpha | "_") (alphanum | "_")*)*
```

Those are the grammar rules with *condition* as a starting non-terminal:

```
condition      := or-condition
or-condition   := and-condition ("or" and-condition)*
and-condition  := simple-condition
               | qualified-condition
               | sub-condition
```

³² <https://javacc.dev.java.net/>

```

        ("and" (simple-condition
                | qualified-condition
                | sub-condition)
        ) *
simple-condition    := property-path
                    ("==" | "!=" | "<" | "<=" | ">" | ">="
                     | "contains" | "=~" | "!~")
                    (long | version | date | pattern | string)
qualified-condition := property-path "{" condition "}"
sub-condition       := "(" condition ")"

```

Appendix D: Source Code Layout

BEEN source code is divided into several directories. Their layout is based on usual conventions used for the Java projects. The description of the directories follows:

- `bin` – batch files (for Windows) and shell scripts (for Linux) used for running the Task Manager and the Host Runtime.
- `build` – created by the compile target, contains compiled Java classes.
- `data` – various data directories used by BEEN components.
- `dist` – results of the BEEN build – the `been.jar` file with compiled BEEN Java classes and the `tasks` subdirectory, which contains task packages.
- `doc` – documentation files. Directly in the directory, there is a documentation source file (in OpenOffice.org 2.0 .odt format) and generated PDF file. In the subdirectories, the generated documentation of the Java and C++ sources is placed by the `javadoc` and `docs-native` targets.
- `examples` – example data and classes used for BEEN development.
- `lib` – 3rd party libraries used by BEEN.
- `native` – C++ source code of the detectors and load monitors for all supported platforms.
- `resources` – additional resources: DTD files, testing data files used by the unit tests, example task descriptors and files (`metadata.xml`, `config.xml`, etc.) used in the task packages.
- `src` – Java source code of BEEN. The source files are divided into packages, all of which are subpackages of the `cz.cuni.mff.been` package.
- `tests` – Java unit tests of some parts of BEEN, written using JUnit framework. The structure of the subdirectories matches the structure of the `src` directory.
- `webinterface` – code of the web interface which is deployed on the Tomcat servlet container.

Appendix E: Third-party Libraries

BEEN uses several third-party libraries for various purposes (logging, RSL parser generation, benchmark plugin management, interface to source-control systems, etc.).

Most libraries are placed in the form of JAR files in the `lib` directory.

The complete list of the libraries, their files and licenses follows:

Name	File(s)	License
Apache Ant	ant.jar	Apache License 2.0
FileUpload	commons-fileupload-1.0.jar	Apache License 2.0
Jakarta Commons Logging	commons-logging.jar	Apache License 2.0
JavaCC	javacc.jar	BSD License
Java Plug-in Framework	jpf.jar, jpf-boot.jar	LGPL 2.1
JavaSVN	javasvn.jar	custom license
LOG4J	log4j-1.2.12.jar	Apache License 2.0
NetCDF	netcdfAll.jar, pefsAll.jar	LGPL 2.1
SJava	sjava.jar, Environment.jar	GPL 2.0
ANTLR	antlr.jar	BSD License
JavaCVS	cvslib_36.jar	GPL 2.0
Java Assembler	jas.jar ³³	custom
Java Help	jhall.jar ³⁴	Sun's Binary Code License Agreement
JSP API	jsp-api.jar	Apache License 2.0
Servlet API	servlet-api.jar	Apache License 2.0
Selenium Core	webinterface/tests/selenium-core/*	Apache License 2.0

³³ Distributed by Omegahat as a part of `sjava.jar`.

³⁴ Distributed by Omegahat as a part of `sjava.jar`.