EverBEEN

Martin Sixta, Tadeáš Palusga, Radek Mácha, Jakub Břečka

`www.everbeen.cz`

Contents

Chapter 1

Introduction to EverBEEN

1.1 Foreword

Automatic testing of software has become an integral part of software development and software engineering today heavily relies on two levels of testing and verification to ensure the quality of programs:

- **Unit testing** which refers to the process of testing whether a single isolated component behaves according to the specification. Unit tests are usually conducted in a white-box manner and nowadays, software is often engineered in a *test-driven development* environment, where the developers first write unit tests before actually implementing a component.
- **Integration testing** which tries to verify the interaction and integration of multiple components within a system. It is usual to perform this using black-box testing and there are several patterns and approaches for integration testing.

Such testing is not only done in order to hunt bugs and discover non-functional code, but it has found its use in performance testing and evaluation as well. The requirement to focus development on performance is becoming a standard part of software engineering. Performance evaluation (benchmarking) can have different goals, e.g.:

- **Regression testing**, which helps the developer determine whether a newly implemented feature has any impact on the performance of the system.
- **Scalability measures**, where the system is measured under an increasing load.
- **Comparison with competing software**
- **Determining the bottleneck**

Despite the fact that performance measuring is definitely useful, it still has quite a low popularity among development teams. Regression benchmarking is uncommon and the implementation of individual benchmarks is usually project-specific. The need for a generic framework for regression benchmarking is obvious for many reasons:

- Benchmarking middleware is hard because of the need for a complex environment. A generic framework could simplify the tasks of deployment, configuration and management of a networked benchmarking environment.
- Performing a long-running benchmark on multiple machines requires a significant amount of work to ensure the benchmark will continue even after a failure of one machine. The framework could provide such facilities easily.
- A benchmarking framework could easily automate evaluation and integrate it into the development process.

- The environment can offer facilities, such as synchronization, logging and communication mechanisms, that would make the task of creating a benchmark easier.
- Statistics, analysis and visualization are good candidates for having a helpful library instead of writing a custom one.

1.1.1 Related works

- Kalibera, T., Lehotsky J., Majda D., Repcek B., Tomcanyi M., Tomecek A., Tuma, P., Urban J.: Automated Benchmarking and Analysis Tool[1] (PDF, 149 kB), VALUETOOLS 2006
- Kalibera, T., Bulej, L., Tuma, P.: Generic Environment for Full Automation of Benchmarking[2] (PDF, 84 kB), SOQUA 2004
- Täuber, J.: Deployment of Performance Evaluation Tools in Industrial Use Case[3] (PDF, 614 kB), MFF 2013

1.2 Case study

During the development of EverBEEN, several use cases were considered and this section describes the ones that BEEN was specifically designed for.

1.2.1 Regression benchmarking

Regression benchmarking is a technique mostly aimed at discovering negative performance impact of a newly added feature, an upgrade or a single patch in the source code. This usually involves performing the same set of benchmarks against various versions of the same software. Often, even individual revisions of the source tree are tested. When the benchmark is stable enough to result in consistent data, it is easy to immediately see which commits had an impact on performance.

While performance degradation is obviously an undesired effect, unplanned performance increase can also be an indicative of a problem. This can easily happen when an expensive check (e.g. for security purposes) is unintentionally removed or bypassed. Whether performance fluctuations be positive or negative, regression benchmarking provides the development team with crucial information which not only denotes the actutal performance change, but also points to the exact code modification that caused it.

1.2.2 Pull-oriented benchmarking

Consider the following case: We have access to a source code repository with version history of a software product, whose developers don't do any regular benchmarking. Suddenly, they realize that their software behaves much slower than a year ago. Although the performance degradation is probably caused by several factors, the developers would like to determine major slowdown culprits and eliminate them.

If this software were a standalone desktop application, the obvious solution would be to build all the revisions of the software since last year and benchmark them. Writing a script automating this task would be straightforward. When the software in question is distributed middleware, even setting up a benchmarking environment could be a costly task.

This task can be generalized into a problem of running a benchmark over a set of parameters. The parameters are known in advance and the benchmark is by necessity a user-written code. A generic benchmarking framework should therefore simplify both parameter specification *and* the process of writing benchmark code. The user's options must be flexible enough to support many possible configurations of the benchmark — one might want to benchmark a single piece of software with various configurations.

[1]http://d3s.mff.cuni.cz/publications/download/Submitted_1404_BEEN.pdf
[2]http://d3s.mff.cuni.cz/publications/download/KaliberaBulejTuma-FullAutomationOfBenchmarking.pdf
[3]https://is.cuni.cz/webapps/zzp/detail/78663/4417375

A benchmark iterating over a predefined set of parameters is called a **pull-oriented benchmark**. The details of EverBEEN's support for this use case are discussed later along with the project goals.

1.2.3 Push-oriented benchmarking

Another practical use case is the incorporation of benchmarking into a **continuous integration environment**. Such environments usually perform a large suite of unit tests whenever a commit into the repository is made. The results (especially failed tests) are then shown either on a project status web page or sent via email to the developers.

Deploying a *continuous regression testing* suite into such a system would then be a matter of integrating a benchmarking framework in such a way that a suite of prepared benchmarks would be run every time a new commit is made. We call this case **push-oriented benchmarking**, because there is no predefined set of items to benchmark. Instead, a *push event* should be dispatched that would cause the newly created revision to be tested.

1.3 Target audience

EverBEEN is a project for developers, testers, software project leaders and researchers who are looking for a way to automate benchmarking and testing. To these people, EverBEEN can provide an environment that can be easily deployed into a heterogeneous network and that can ease the task of creating, debugging, running and managing benchmarks, especially those aimed at evaluating distributed systems and middleware.

What BEEN is *not*:

- BEEN is not a benchmark, nor will it by itself perform any actual benchmarking.
- BEEN is not a standalone desktop application, you will have to provide or write your own tasks and benchmarks.

To be able to use BEEN, users are expected to be experienced in the area of software benchmarking and performance testing. Users should already know what exactly they want to benchmark, how they are going to benchmark it and what the outcome of their benchmarks should be. They should know how to interpret and evaluate the resulting data and be able to understand what the benchmark measured and whether the output makes sense.

Users should also have a decent knowledge about general benchmarking practices, possible problems, and various factors that can influence the validity of results. This is especially required for regression benchmarking, where the consequences of a wrong choice of metrics or data misinterpretation are amplified by the direct projection of benchmarking conclusions on the development process.

1.4 Project history

1.4.1 BEEN

The original BEEN project was started in Fall 2004 and finished at the turn of 2006 and 2007 It was supervised by Tomáš Kalibera and developed by Jakub Lehotský, David Majda, Branislav Repček, Michal Tomčányi, Antonín Tomeček and Jaroslav Urban. This project's assignment was:

> *"The aim of the project will be to create a highly configurable and modular environment for benchmarking of applications, with special focus on middleware benchmarks."*

The team that worked on the project created the whole architecture and individual components of the framework and eventually implemented a functional benchmarking environment in Java, using *RMI* as the main mean of communication among its individual parts.

1.4.2 WillBEEN

The second incarnation of the framework was called *WillBEEN* and it mainly continued development of the original project. Its goal was to extend BEEN, mainly focusing on adding support for non-Java user tasks (scripts), creating a modular results repository component and devising a fast and reliable command-line user interface for the framework.

This project was supervised by Petr Tůma and developed by Andrej Podzimek, Jan Tattermusch and Jiří Täuber. The team started working in 2009 and finished the project in March 2010. During the development, several components were redesigned and reimplemented and the project integrated several new technologies, such as JAXB[4] and Apache Derby[5].

1.4.3 State of WillBEEN in 2012

In 2011, the faculty decided to create another assignment for BEEN as its state was still far from ideal. Since the original team started working on the project more that 7 years before, its codebase used obsolete technologies and the legacy of the initial architecture was causing issues with both stability and performance. The choice of *RMI* for component communication was deemed to be the main culprit. WillBEEN also had many *single points of failure*, e.g., disconnecting a single component rendered the whole environment unusable.

WillBEEN's development team had to cope with a large, old and fragile codebase. While changes introduced during the development were of good to high quality, the team lacked necessary resources to radically change or rewrite all parts of the framework.

WillBEEN deployment was yet another problematic part. Installing and configuring the environment took a tremendous amount of effort. Last but not least, the user API for writing benchmarks was very complicated, and user benchmark code was almost impossible to debug.

The new team was therefore supposed to eliminate some of these shortcomings, while stabilizing the framework even further. Thus the goals set were to rewrite the oldest parts of the framework, while maintaining the rest, along with finding a better approach to component communication based on asynchronous message passing. The work load was estimated to +20,000 LOC.

1.4.4 EverBEEN

EverBEEN is a complete rewrite of the BEEN framework from scratch. It took into account previous experience[6] with WillBEEN deployment and exploited current technologies and software development standards.

EverBEEN has a fundamentally different, decentralized architecture. Many aspects of the project were simplified by virtue of popular 3rd party Java libraries, which makes the whole framework more stable and compliant to modern development techniques. However, the naming of individual BEEN components and work units was preserved. Therefore, users familiar with previous BEEN implementations should have no trouble adapting to the new system implementation.

The decision to do a complete rewrite was made after careful consideration of all options. The incompatibility of project goals with the state of WillBEEN's codebase was the key piece that tipped the odds in favor of restarting from scratch.

EverBEEN is supervised by Andrej Podzimek and Petr Tůma, and developed by Martin Sixta, Tadeáš Palusga, Radek Mácha and Jakub Břečka. The work on the project started in Fall 2012 and its first stage is aimed to finish in September 2013.

[4]http://jaxb.java.net/
[5]http://db.apache.org/derby/
[6]https://is.cuni.cz/webapps/zzp/detail/78663/4417375/

1.5 Project Goals

This section contains text copied directly from the Project Committee's web site.

http://ksvi.mff.cuni.cz/~holan/SWP/zadani/ebeen.txt

1.5.1 Overview

> *"The Been framework automatically executes software performance measurements in a heterogeneous networked environment. The basic architecture of the Been framework consists of a host runtime capable of executing arbitrary tasks, a task manager that relies on the host runtime to distribute and execute scheduled sequences of tasks, and a benchmark manager that creates the sequences of tasks to execute and measure benchmarks. Other components include a software repository, a results repository, and a graphical user interface."*

The Been framework has been developed as a part of a student project between 2004-2006, and substantially modified as a part of another student project between 2009-2010.

1.5.2 Goals

The overall goal of this project is to modify the Been framework to facilitate truly continuous execution. In particular, this means:

- Reviewing the code responsible for communication between hosts, setting up rules that prevent the communication from creating orphan references (and therefore memory leaks), and rules that make the communication robust in face of network and host failures.
- Reviewing the code responsible for logging, setting up rules that govern all log storage (and prevent uncontrolled growth of logs).
- Reviewing the code responsible for temporary data storage, setting up rules that enforce reliable temporary data storage cleanup while preserving enough data for post mortem inspection of failed tasks and hosts.
- Reviewing the code responsible for measurement result storage, setting up rules for archival and cleanup that would make it possible to store recent results in detail and older results for overview purposes.
- Generally clean up any reliability related bugs.

1.5.3 How we met the goals

The following overview takes into account goals set for the project as submitted to the Project Committee. The overall changes were much more substantial than anticipated.

Reviewing the code responsible for communication between hosts . . .
A completely new architecture and communication protocol was introduced based on scalable, redundant data distribution.

Reviewing the code responsible for logging . . .
Both user code API and framework code were ported under a unified logging system compliant to latest Java development standards.

Reviewing the code responsible for temporary data storage . . .
A deletion policy was set up for all leftover user task data (working directories, logs, results), enforcing automatic cleanup after a configurable expiration period or possibility of easy manual deletion.

Reviewing the code responsible for measurement result storage ...
A complete overhaul of the component responsible for result storage and retrieval was made.

Generally clean up any reliability related bugs.
Adoption of standard development techniques and usage of third party components resulted in a much smaller and compact code base.

1.6 Project Output

The initial assignment of the EverBEEN project mainly focuses on delivering a more usable, stable and scalable product. That being said, it was assumed that the development team will work on existing codebase and refactor it instead of starting from scratch.

There were, however, multiple design flaws refactoring alone could not remedy. The *RMI* library was too deeply embedded into the codebase to be simply replaced. The individual modules of WillBEEN were cross-linked and couldn't be separated by well-defined interfaces. Multiple implementations of the same functionality (e.g. logging) made the codebase scattered and inconsistent. Also, the WillBEEN implemented several custom facilities which are, as of toady, standard issue among external Java libraries.

To meet stability and scalability requirements, the team decided to rewrite BEEN from scratch, only preserving the concept and several design decision, e.g. the choice of most components and their purpose. Subsequently, the team could focus on creating a scalable, usable product from the first moment.

Therefore the project goals were extended to include:

- Preserving the basic concept of the whole environment
- Innovating the code base by use of modern technologies and practices
- Delivering a highly scalable and stable product
- Reducing the number of single points of failure
- Making the framework easy to deploy
- Improving usability by simplifying task and benchmark creation and debugging

1.6.1 Distributed nature of EverBEEN

One of WillBEEN's major issues was reliance on network stability. The framework required that all involved computers be running and available. Disconnecting some of the core services caused the whole framework to hang or crash, and recovery was often impossible. Also, the core EverBEEN components were required to be running for the whole time, which created a lot of single points of failure. That aggravated common situations like short-term network outages to irrecoverable system failures.

Such fragile client-server architecture seemed inappropriate for a framework supposedly tailored for large and heterogeneous networks. That is why EverBEEN is built on *Hazelcast* – a decentralized, highly scalable platform for distributed data sharing. Hazelcast is a Java-based library that implements peer-to-peer communication over TCP/IP, featuring redundant data sharing, transparent replication and automatic peer discovery. This platform provides distributed maps, queues, lists, locks, topics, transactions and synchronization mechanisms using distributed hashing tables.

Hazelcast supports data redundancy and fail-over mechanisms, which EverBEEN uses to provide a decentralized benchmarking environment. Its nodes are mutually equal, and the framework keeps running as long as at least one node is partaking in data sharing. When a node gets disconnected, the cluster is notified and ceases using this node until it reconnects. To fully profit from this fault-tolerant behavior, core EverBEEN components function in a decentralized manner and transparently partition work across many instances.

This architecture makes EverBEEN a fully distributed platform with high availability and scalability, while eliminating most bottlenecks and substantially reducing the number of critical components.

1.6.2 EverBEEN's Support for Regression Benchmarking

EverBEEN was designed to cover both use cases discussed in the Case Study, while keeping the user code API to a minimum. The API for writing benchmarks is a unified means of creating and submitting sets of tasks on every invocation (realized by the framework). Depending on the benchmark's control flow, it can either act like a service to support *push-oriented* benchmarking, or iterate over a pre-defined set of parameters in a *pull-oriented* way.

During development, implementation of a declarative language describing benchmarks was considered. Such language would, however only support the pull-oriented case. Subsequently, EverBEEN would require a different API for push-oriented benchmarking. The unified API offers unlimited flexibility, as the generation of task sets is in full control of the user. Additionally, the running benchmark can take the current (incomplete) results into account and modify the progress of the benchmark. This feature has many uses, for example granularity refinement in reaction to a previously detected anomaly.

The unified API for writing tasks and benchmarks is discussed in detail in section 2.5 (Task and Benchmark API).

1.7 The EverBEEN team

The EverBEEN framework was developed by *Jakub Břečka*, *Radek Mácha*, *Tadeáš Palusga* and *Martin Sixta*, under the supervision of *Andrej Podzimek*.

1.7.1 Contributions

Overview of main contributions to the project by team members:

Jakub Břečka

- Task API
- Benchmark API
- Monitoring
- Web Interface
- `nginx` benchmark

Radek Mácha

- Object Repository
- Software Repository
- `JAXB` internals
- Inter-process communication

Tadeáš Palusga

- Host Runtime
- Web Interface
- Software Repository
- BPK Plugin for Maven

Martin Sixta

- Task Manger
- Host Runtime
- Task API
- Inter-process communication
- `Hazelcast` benchmark

1.8 Glossary

`benchmark`
Special-purpose task designed for *task context* generation.

`Benchmark API`
API assisting EverBEEN users with writing *benchmarks*.

`BPK`
EverBEEN package containing software and metadata necessary for running *tasks*, *benchmarks* and/or *evaluators*.

`BPK Plugin`
Apache Maven plugin capable of generating *BPK* bundles.

`checkpoint`
Inter-task synchronization primitive.

`DATA node`
A `node` instance that participates in distributed data sharing and runs a *Task Manager* service.

`evaluator`
Special-purpose task designed to perform presentable evaluations on results generated by other tasks.

`EverBEEN service`
A software component that adds extra functionality to an EverBEEN node. Services are launched at node boot time. Node service selection is specified by command-line options.

`Host Runtime`
EverBEEN service that executes *tasks* and mediates communication between *tasks* and the rest of the EverBEEN cluster.

`NATIVE node`
A `node` instance that does not participate in distributed data sharing.

`node`
Java application providing clustering functionality and capable of running EverBEEN services.

`Object Repository`
Universal storage component for EverBEEN user data.

`Result`
User type carrying *task* output data.

`Software Repository`
An *EverBEEN service* cabable of distributing *BPK* bundles across the cluster.

`task`
Unit of user-written code executable by the EverBEEN framework.

`Task API`
API assisting EverBEEN users with writing *tasks*.

`task descriptor`
`XML` description of a *task*'s configuration.

`task context`
A container grouping multiple tasks into a logical unit.

`task context descriptor`
A `XML` representation of a *task context*.

`Task Manger`
EverBEEN service in charge of *task* scheduling.

Chapter 2

EverBEEN user guide

2.1 EverBEEN requirements

BEEN is designed from the ground up to be a multi-platform software. Currently supported platforms include:

- Linux – most recent distributions
- Mac OS X 10.8 and later
- Microsoft Windows 7 and later
- FreeBSD

In order to deploy BEEN these software packages need to be installed:

- Java Runtime Environment (JRE) version 1.7

For writing and debugging user-written tasks:

- Apache Maven version 3

For a node that will run the web interface client:

- Java Servlet compatible container (e.g. Tomcat 7, Jetty)

(The container is optional, the Web Interface can be also run in embedded mode.)

For a node that will run the results repository, the machine needs:

- MongoDB version 2.4

The clients that should be able to access the web interface need to have one of the following web browsers:

- Google Chrome version 29 or newer
- Mozilla Firefox version 22 or newer

The project does not have any explicit hardware requirements, any machine that meets the software requirements listed above, should be able to run BEEN. However, the recommended minimum machine hardware configuration is:

- Modern CPU with at least 2.0 GHz
- 100Mbit network interface
- 4 GB of RAM
- 10 GB of HDD free space

2.2 Basic concepts

Before delving into the deployment process a few concepts must be explained. The concepts are explored and further explained in the following chapters.

2.2.1 BEEN services

An EverBEEN service is a component that runs indefinitely and processes requests. Essential services include:

- Host Runtime — executes tasks
- Task Manager — schedules tasks
- Software Repository — serves packages
- Object Repository — provides persistence layer

2.2.2 Tasks

An EverBEEN task is a basic executable unit of the framework. Tasks are user written code which the framework runs on Host Runtimes.

Tasks are distributed in the form of package files called *BPK*s (from `BEEN package`). BPKs are uploaded to the Software Repository and are uniquely identified by *groupId*, *bpkId* and *version*.

Task Descriptors are XML files describing which package to use, where and how to run a task. Task Descriptors are submitted to a Task Manager which schedules and instantiates the task on a Host Runtime which meets user-defined constraints.

Tasks have states:

CREATED
Initial state of the task.

SUBMITTED
The state after the task is submitted to a Task Manager.

ACCEPTED
The state after a task is accepted on a Host Runtime to be run.

RUNNING
The state indicates that the task is running on a Host Runtime.

FINISHED
Indicates successful completion of the task.

ABORTED
Indicates that the task failed while running or cannot be run at all (for example because of a missing BPK).

2.2.3 Contexts

EverBEEN contexts group related tasks to achieve a shared goal. Contexts are not runnable entities, their life cycle is derived from states of contained tasks. Contexts are described by *Task Context Descriptor* XML files.

Task context states:

RUNNING
Contained tasks are running, scheduled or waiting to be scheduled.

FINISHED
All contained tasks finished without an error.

FAILED
At least one task from the context failed.

2.2.4 Benchmarks

Benchmark are user-written tasks with additional capabilities (in form of the *Benchmark API*). Benchmark tasks generate task contexts which are submitted to the framework.

2.2.5 Results

Results are task generated objects representing certain values — for example measured code characteristics.

2.2.6 Evaluators

Evaluators are special purpose tasks which generate *evaluator results* the framework knows how to interpret, for example a graph image.

2.2.7 Node types

In EverBEEN `node` is a program capable of running BEEN services. The node must be able to interact with other nodes through a computer network. Type of a node determines the mechanism used to connect to other nodes. Since EverBEEN uses Hazelcast as its means of connecting nodes, node types follow a design pattern from Hazelcast. Currently two types are supported:

`DATA node`
Data nodes form a cluster that *share distributed data.* The cluster can be formed either through broadcasting or by directly contacting existing nodes, see section 2.8.1.1 (Cluster Configuration). The Task Manager service must be run on each DATA node (this requirement is enforced by the framework). Be aware that DATA nodes incur overhead due to sharing data.

`NATIVE node`
Native nodes can be though of as cluster clients. They *do not* participate in sharing of distributed data and therefore do not incur overhead from it. NATIVE nodes connect directly to DATA nodes (failures are transparently handled). This also means that at all times at least one DATA node must be running in order for the framework to work. For configuration details see section 2.8.1.2 (Cluster Client Configuration).

All services except the Task Manager can run on both node types.

2.3 Deployment process

2.3.1 Running EverBEEN

The deployment process assumes a set of interconnected computers on which the framework is supposed to run and a running MongoDB instance. See section 2.1 (Requirements) and MongoDB installation guide[1] for details.

Deploying EverBEEN consists of two steps:

[1] http://docs.mongodb.org/manual/installation/

- Copying EverBEEN onto each machine — single executable jar file is provided
- Creating clustering configurations

The exact configuration is highly dependent on the network topology. In the following example configuration two scenarios will be presented depending on how the cluster will be formed.

Usually, there will be a few *DATA* nodes and as many *NATIVE* nodes running the Host Runtime service as needed.

We will also assume that MongoDB instance is running on `mongodb.example.com`. All nodes must use the same *group* and *group password*.

2.3.1.1 Broadcasting scenario

The cluster is formed through broadcasting.

```
been.cluster.mapstore.db.hostname=mongodb.example.com
mongodb.hostname=mongodb.example.com

been.cluster.multicast.group=224.2.2.4
been.cluster.multicast.port=54326

been.cluster.group=dev
been.cluster.password=dev-pass
```

Only the first two configuration options are needed, rest of options have sane defaults.

2.3.1.2 Direct connection scenario

The cluster will be formed by directly connecting nodes.

```
been.cluster.mapstore.db.hostname=mongodb.example.com
mongodb.hostname=mongodb.example.com

been.cluster.join=tcp
been.cluster.tcp.members=195.113.16.40:5701;host1.example.com;host2.example.com

been.cluster.group=dev
been.cluster.password=dev-pass
```

The `been.cluster.tcp.members` option specifies a (potentially partial) list of nodes to which the connecting node will try to connect. If no node in the list is responding a new cluster will be formed.

2.3.1.3 Connecting NATIVE nodes

NATIVE nodes must be informed to which DATA nodes to connect:

```
been.cluster.client.members=host1.example.com:5701;host2.example.com
been.cluster.group=dev
been.cluster.password=dev-pass
```

The `been.cluster.client.members` option is important, again specifying a (potentially partial) list of DATA nodes to connect to.

The configuration can be copied directly onto the hosts or can be referenced by an URL (which is the preferred way).

2.3.1.4 Configuring EverBEEN services

The next step is to decide which BEEN services will be run and where. In the simplest and most straight forward case one node will be running *Software repository*, *Object repository*, *Host Runtime* and implicitly the *Task Manager*.

```
java -jar been.jar -r -sw -rr -cf http://been.example.com/been.properties
```

Other nodes thus can run only the *Host Runtime* service.

```
java -jar been.jar -t NATIVE -r -cf http://been.example.com/been-clients.properties
```
for NATIVE nodes

```
java -jar been.jar -r http://been.example.com/been-broadcast.properties
```
in case of the broadcasting scenario

```
java -jar been.jar -r -sw -rr -cf http://been.example.com/been-direct.properties
```
in case of the direct connection scenario

To list available command line options run EverBEEN with:

```
`java -jar been.jar --help`
```

2.3.1.5 Running the Web Interface

The last step consists of deploying and running the *Web Interface*. The supplied war file can be deployed to a standard Java Servlet container (e.g. Tomcat). Or can be run directly by

```
java -jar web-interface-3.0.0-SNAPSHOT.war
```

using an embedded container.

2.3.2 Node directory structure

Node working directory is created on startup.

```
1.  .HostRuntime/
2.      \___ tasks/
3.          \___ 1378031207851/
4.          \___ 1378038338005/
5.          \___ 1378038763308/
6.          \___ 1378040071618/
7.              \___ example-task-a_1bdcaeb4/
8.              |   \___ config.xml
9.              |   \___ files/
10.             |   \___ lib/
11.             |   \___ stderr.log
12.             |   \___ stdout.log
13.             |   \___ tcds/
14.             |   \___ tds/
15.             \___ example-task-b_6a2ccc11/
16.             \___ ...
17.             \___ ...
```

- **.HostRuntime** directory (1) — Host Runtime global working directory. It can be configured by changing the property `hostruntime.wrkdir.name`. The default name is `.HostRuntime`.

- Each run of EverBEEN creates separate working directory for its tasks in the **tasks** subdirectory (2).
- On restart a new working directory for tasks (3,4,5,6) is created. Names of these directories are based on the node startup (wall clock) time. EverBEEN on each start checks these directories and if their number exceeds 4 (by default), the oldest one is deleted. This prevents an unexpected growth of the Host Runtime working directory size, but allows debugging failed tasks when the underlying Host Runtime is terminated and restarted. The number of backed up directories is configurable by the `hostruntime.tasks.wrkdir.maxHistory` configuration option.
- Working directories of tasks (7,15,16,17) contain files from an extracted BPK (8,9,10,13,14) and log files for the standard error output (11) and standard output (12).

The working directory of a task is deleted only if the task finished its execution without error, otherwise the directory remains unchanged. Alternatively, you can either clean up the directory manually or use the Web Interface for that purpose.

2.3.3 Limitations

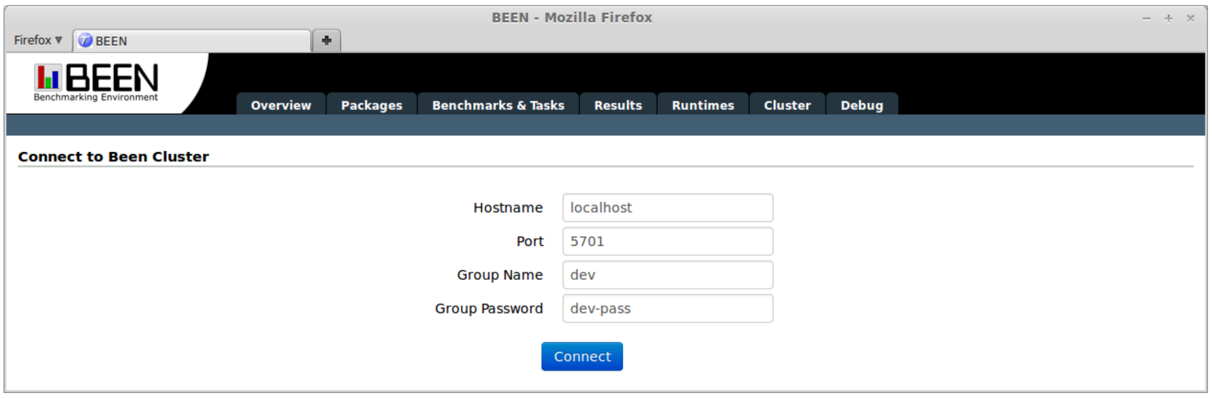
- If you want to run more than one Host Runtime on the same machine we **strongly recommend** to start each node with a different working directory name. Running multiple instances concurrently with the same working directory *is not supported.*
- Running EverBEEN for a long time without clearing directories after failed tasks can result in low disk space.

2.4 Web Interface

The Web Interface is the tool to interact with the EverBEEN framework.

2.4.1 Connecting to the cluster

First, the Web Interface needs to connect to the EverBEEN cluster (*Figure 2.1*). You have to provide cluster connection credentials. If you run your nodes with default configuration, default host name, port (type of the node must be *DATA*), group name and group password is prefilled in the login form. Click on **connect** to establish a connection with the cluster.



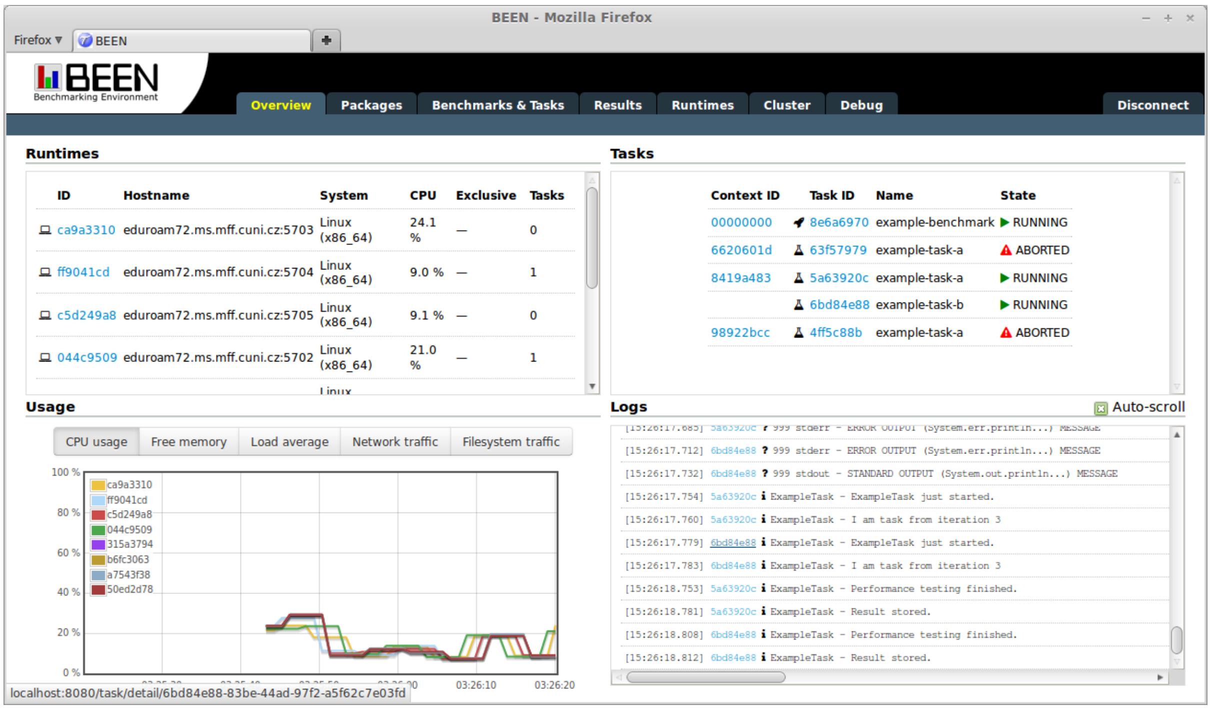
Connect to Been Cluster

Hostname	localhost
Port	5701
Group Name	dev
Group Password	dev-pass

Figure 2.1: Login

2.4.2 Cluster overview

The overview page (*Figure 2.2*) shows a quick overview of connected nodes, node resources, currently active or failed tasks and task logs.



Runtimes

	ID	Hostname	System	CPU	Exclusive	Tasks
💻	ca9a3310	eduroam72.ms.mff.cuni.cz:5703	Linux (x86_64)	24.1 %	—	0
💻	ff9041cd	eduroam72.ms.mff.cuni.cz:5704	Linux (x86_64)	9.0 %	—	1
💻	c5d249a8	eduroam72.ms.mff.cuni.cz:5705	Linux (x86_64)	9.1 %	—	0
💻	044c9509	eduroam72.ms.mff.cuni.cz:5702	Linux (x86_64)	21.0 %	—	1

Tasks

Context ID	Task ID	Name	State
00000000	8e6a6970	example-benchmark	▶ RUNNING
6620601d	63f57979	example-task-a	⚠ ABORTED
8419a483	5a63920c	example-task-a	▶ RUNNING
	6bd84e88	example-task-b	▶ RUNNING
98922bcc	4ff5c88b	example-task-a	⚠ ABORTED

Logs

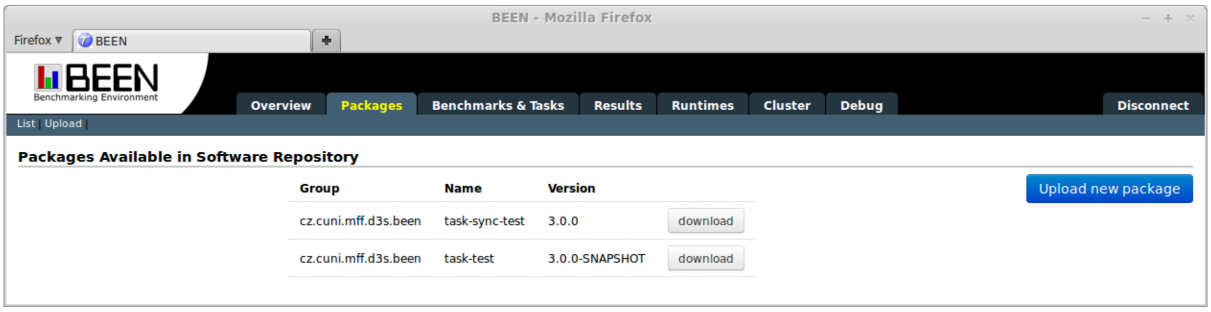
```
[15:26:17.685] 5a63920c ? 999 stderr - ERROR OUTPUT (System.err.println...) MESSAGE
[15:26:17.712] 6bd84e88 ? 999 stderr - ERROR OUTPUT (System.err.println...) MESSAGE
[15:26:17.732] 6bd84e88 ? 999 stdout - STANDARD OUTPUT (System.out.println...) MESSAGE
[15:26:17.754] 5a63920c i ExampleTask - ExampleTask just started.
[15:26:17.760] 5a63920c i ExampleTask - I am task from iteration 3
[15:26:17.779] 6bd84e88 i ExampleTask - ExampleTask just started.
[15:26:17.783] 6bd84e88 i ExampleTask - I am task from iteration 3
[15:26:18.753] 5a63920c i ExampleTask - Performance testing finished.
[15:26:18.781] 5a63920c i ExampleTask - Result stored.
[15:26:18.808] 6bd84e88 i ExampleTask - Performance testing finished.
[15:26:18.812] 6bd84e88 i ExampleTask - Result stored.
```

localhost:8080/task/detail/6bd84e88-83be-44ad-97f2-a5f62c7e03fd

Figure 2.2: Cluster overview

2.4.3 Package listing and package uploading

Click on the **Packages** tab. If the *Software Repository* is connected, you can list and download already uploaded packages (*Figure 2.3*).



Packages Available in Software Repository

Group	Name	Version	
cz.cuni.mff.d3s.been	task-sync-test	3.0.0	download
cz.cuni.mff.d3s.been	task-test	3.0.0-SNAPSHOT	download

Figure 2.3: Uploaded packages

Additionally you can upload new packages directly through the Web Interface (*Figure 2.4*).

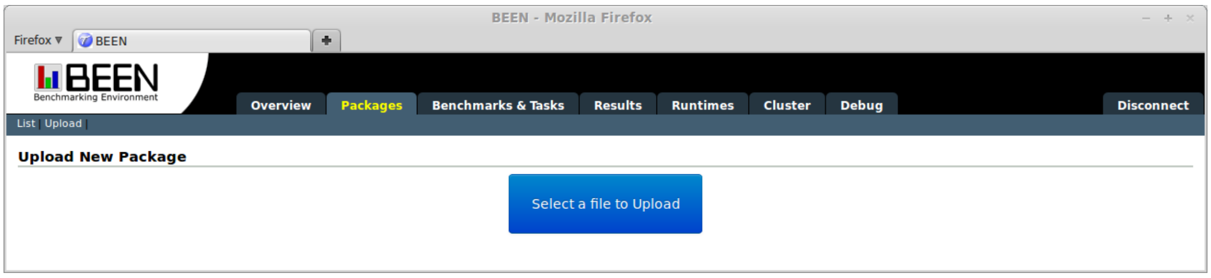
2.4.4 Cluster information and service logs

To view information about the cluster click on the **Cluster** tab (*Figure 2.5*). The page displays a list of cluster members, information about connected services and their states. A cluster member is a EverBEEN `DATA node` - `NATIVE` nodes will not be shown here.

The **Service logs** tab allows to download service logs (*Figure 2.6*).

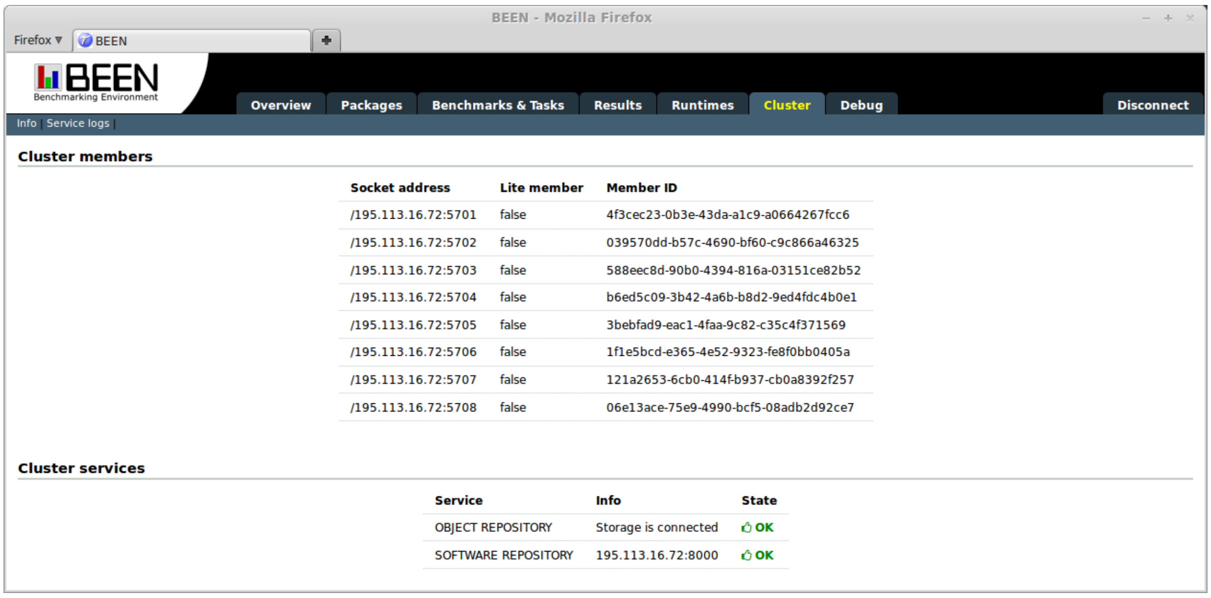
2.4.5 Runtimes

The **Runtimes** tab displays all connected *Host Runtimes* in a table along with basic information on each runtime (*Figure 2.7*).



Upload New Package

Figure 2.4: Uploading new package



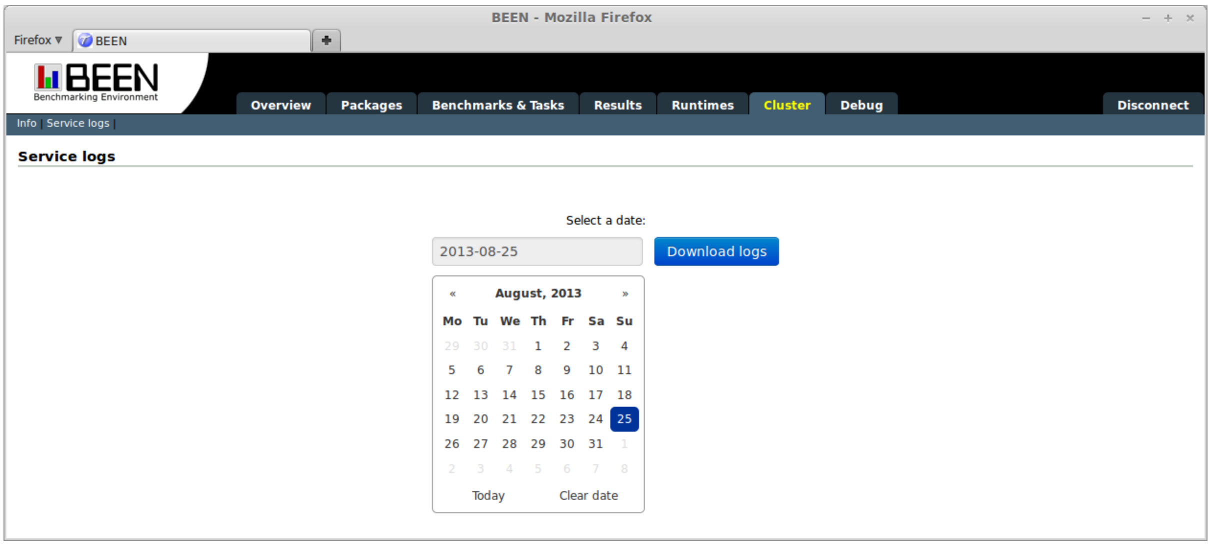
Cluster members

Socket address	Lite member	Member ID
/195.113.16.72:5701	false	4f3cec23-0b3e-43da-a1c9-a0664267fcc6
/195.113.16.72:5702	false	039570dd-b57c-4690-bf60-c9c866a46325
/195.113.16.72:5703	false	588eec8d-90b0-4394-816a-03151ce82b52
/195.113.16.72:5704	false	b6ed5c09-3b42-4a6b-b8d2-9ed4fdc4b0e1
/195.113.16.72:5705	false	3bebfad9-eac1-4faa-9c82-c35c4f371569
/195.113.16.72:5706	false	1f1e5bcd-e365-4e52-9323-fe8f0bb0405a
/195.113.16.72:5707	false	121a2653-6cb0-414f-b937-cb0a8392f257
/195.113.16.72:5708	false	06e13ace-75e9-4990-bcf5-08adb2d92ce7

Cluster services

Service	Info	State
OBJECT REPOSITORY	Storage is connected	OK
SOFTWARE REPOSITORY	195.113.16.72:8000	OK

Figure 2.5: Cluster info



Service logs

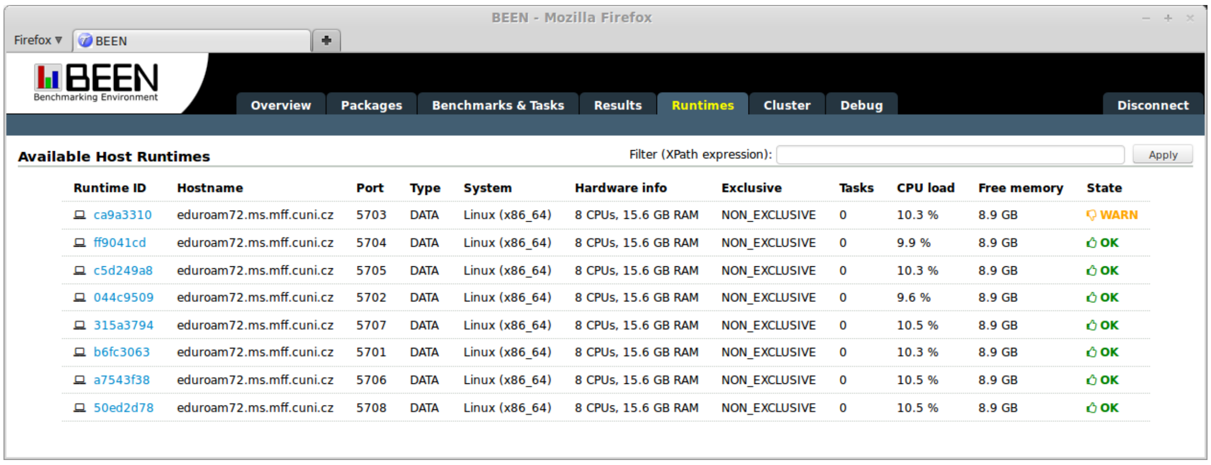
Select a date:

2013-08-25

« August, 2013 »

Mo	Tu	We	Th	Fr	Sa	Su
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

Figure 2.6: Service logs



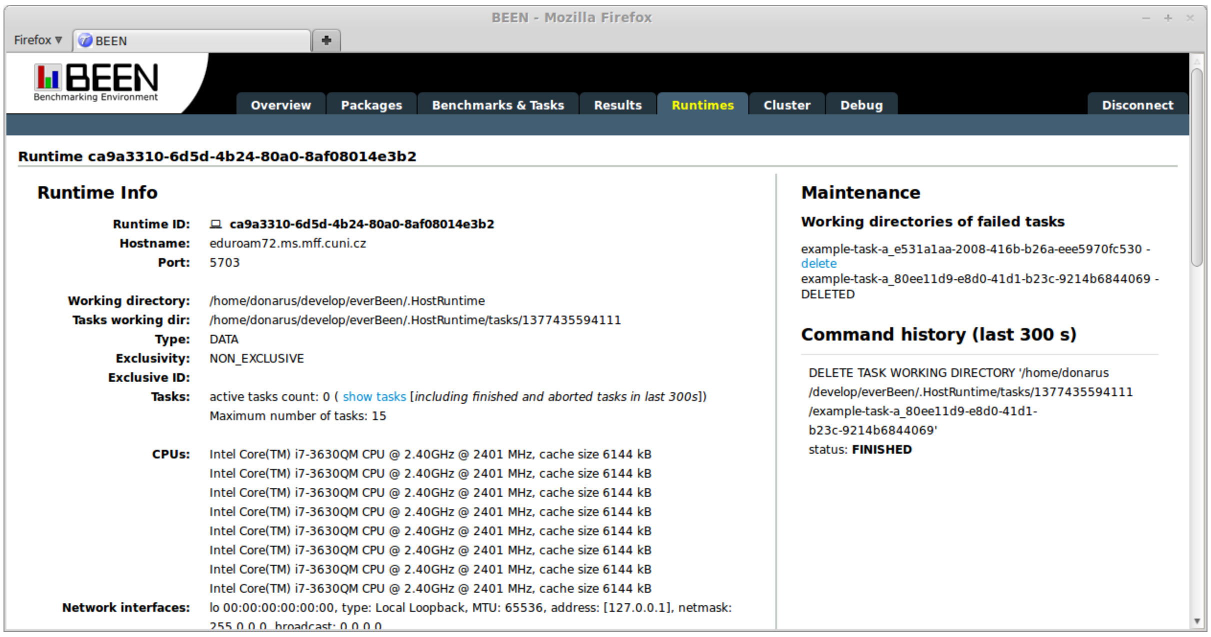
Available Host Runtimes

Filter (XPath expression):

Runtime ID	Hostname	Port	Type	System	Hardware info	Exclusive	Tasks	CPU load	Free memory	State
ca9a3310	eduroam72.ms.mff.cuni.cz	5703	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.3 %	8.9 GB	WARN
ff9041cd	eduroam72.ms.mff.cuni.cz	5704	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	9.9 %	8.9 GB	OK
c5d249a8	eduroam72.ms.mff.cuni.cz	5705	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.3 %	8.9 GB	OK
044c9509	eduroam72.ms.mff.cuni.cz	5702	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	9.6 %	8.9 GB	OK
315a3794	eduroam72.ms.mff.cuni.cz	5707	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.5 %	8.9 GB	OK
b6fc3063	eduroam72.ms.mff.cuni.cz	5701	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.3 %	8.9 GB	OK
a7543f38	eduroam72.ms.mff.cuni.cz	5706	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.5 %	8.9 GB	OK
50ed2d78	eduroam72.ms.mff.cuni.cz	5708	DATA	Linux (x86_64)	8 CPUs, 15.6 GB RAM	NON_EXCLUSIVE	0	10.5 %	8.9 GB	OK

Figure 2.7: Listing runtimes

You can display runtime details by clicking on its ID (*Figure 2.8*).



Runtime ca9a3310-6d5d-4b24-80a0-8af08014e3b2

Runtime Info

Runtime ID:	ca9a3310-6d5d-4b24-80a0-8af08014e3b2
Hostname:	eduroam72.ms.mff.cuni.cz
Port:	5703
Working directory:	/home/donarus/develop/everBeen/.HostRuntime
Tasks working dir:	/home/donarus/develop/everBeen/.HostRuntime/tasks/1377435594111
Type:	DATA
Exclusivity:	NON_EXCLUSIVE
Exclusive ID:	
Tasks:	active tasks count: 0 (show tasks [*including finished and aborted tasks in last 300s*]) Maximum number of tasks: 15
CPUs:	Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB Intel Core(TM) i7-3630QM CPU @ 2.40GHz @ 2401 MHz, cache size 6144 kB
Network interfaces:	lo 00:00:00:00:00:00, type: Local Loopback, MTU: 65536, address: [127.0.0.1], netmask: 255.0.0.0, broadcast: 0.0.0.0

Maintenance

Working directories of failed tasks

example-task-a_e531a1aa-2008-416b-b26a-eee5970fc530 - delete
example-task-a_80ee11d9-e8d0-41d1-b23c-9214b6844069 - DELETED

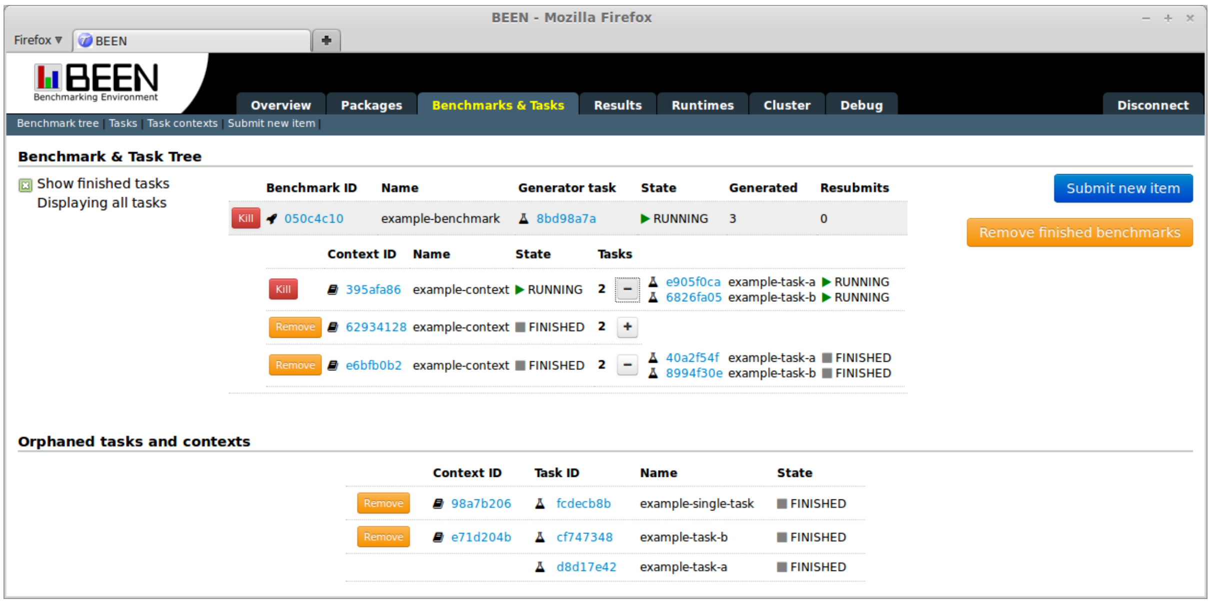
Command history (last 300 s)

DELETE TASK WORKING DIRECTORY '/home/donarus/develop/everBeen/.HostRuntime/tasks/1377435594111/example-task-a_80ee11d9-e8d0-41d1-b23c-9214b6844069'
status: **FINISHED**

Figure 2.8: Runtime detail

2.4.6 Benchmarks and tasks

To run a task, task context or benchmark click on the **Benchmarks & Tasks** tab. The page (*Figure 2.9*) presents information about running tasks, task contexts and benchmarks. You can kill them or remove them from the BEEN cluster. To to kill a benchmark, click on the **kill** button next to the *benchmark id*. All running tasks will be finished and no new tasks and contexts will be started. When the benchmark is killed or finished, you can remove it from the cluster by clicking on the **remove** button next to the benchmark id. All entities related to the benchmark and its task contexts and all persisted records of the benchmark will be deleted, including logs and results. To remove all finished benchmarks, you can use the button **remove finished benchmarks** in the top right corner of the page.



Benchmark & Task Tree

Show finished tasks
Displaying all tasks

	Benchmark ID	Name	Generator task	State	Generated	Resubmits
Kill	050c4c10	example-benchmark	8bd98a7a	RUNNING	3	0

	Context ID	Name	State	Tasks			
Kill	395afa86	example-context	RUNNING	2	−	e905f0ca 6826fa05	example-task-a RUNNING example-task-b RUNNING
Remove	62934128	example-context	FINISHED	2	+		
Remove	e6bfb0b2	example-context	FINISHED	2	−	40a2f54f 8994f30e	example-task-a FINISHED example-task-b FINISHED

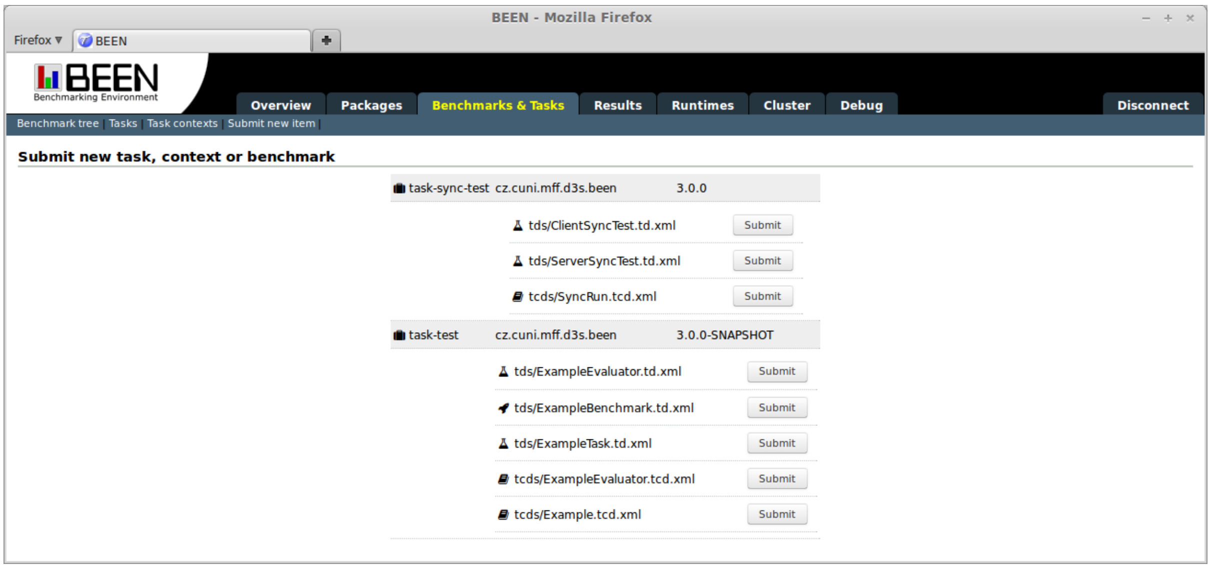
Orphaned tasks and contexts

	Context ID	Task ID	Name	State
Remove	98a7b206	fcdecb8b	example-single-task	FINISHED
Remove	e71d204b	cf747348	example-task-b	FINISHED
		d8d17e42	example-task-a	FINISHED

Figure 2.9: Benchmark tree

2.4.7 Submitting a new task, task context or benchmark

To submit and run a new task, task context or benchmark click the **Submit new item** button on the **Benchmarks & Tasks** page. The submit page (*Figure 2.10*) will present available descriptors from uploaded BPKs as well as user-saved descriptors to run.



Submit new task, context or benchmark

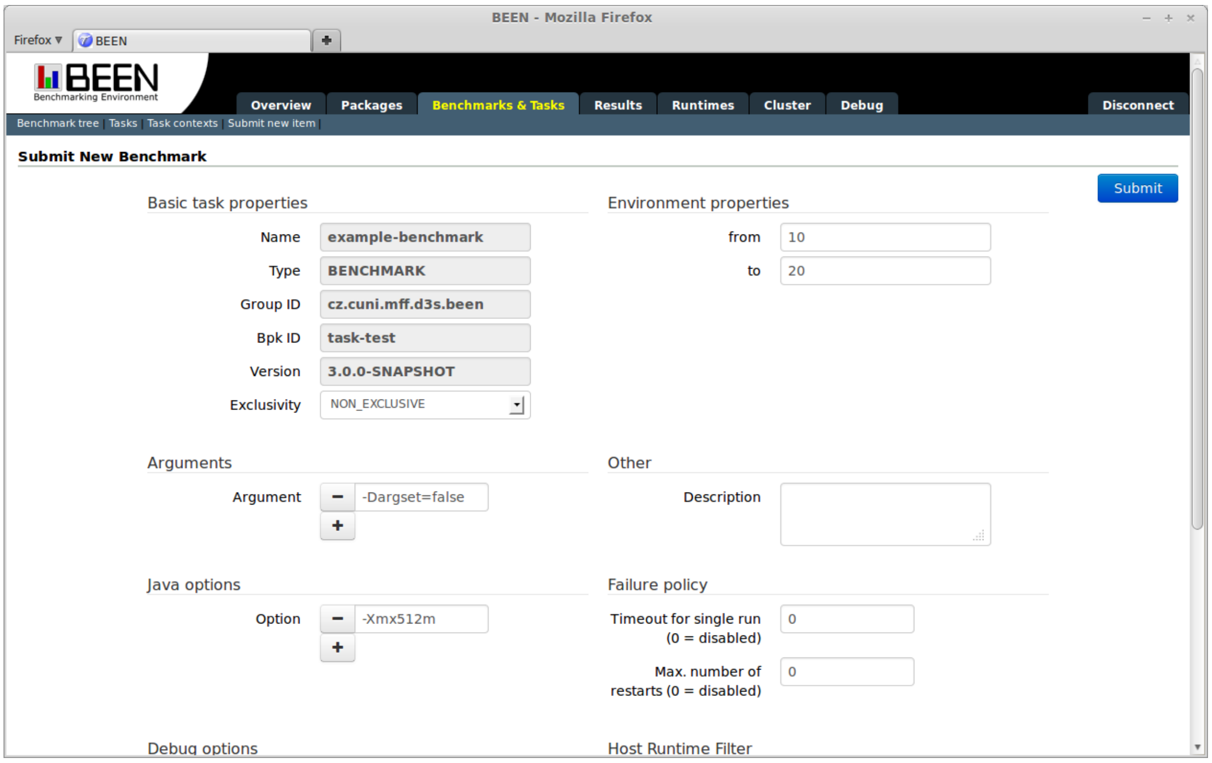
task-sync-test	cz.cuni.mff.d3s.been	3.0.0
	tds/ClientSyncTest.td.xml	Submit
	tds/ServerSyncTest.td.xml	Submit
	tcds/SyncRun.tcd.xml	Submit
task-test	cz.cuni.mff.d3s.been	3.0.0-SNAPSHOT
	tds/ExampleEvaluator.td.xml	Submit
	tds/ExampleBenchmark.td.xml	Submit
	tds/ExampleTask.td.xml	Submit
	tcds/ExampleEvaluator.tcd.xml	Submit
	tcds/Example.tcd.xml	Submit

Figure 2.10: Submitting new item

After clicking on the **submit** button, you can edit the selected descriptor (*Figure 2.1*). You can also save the descriptor for future use.

2.4.8 Listing tasks and task contexts

Instead of working with the benchmark tree, you can list tasks and task contexts independently. Go to the **Tasks** (*Figure 2.12*) or **Task contexts** tab (*Figure 2.13*) on the **Benchmarks & Tasks** page.



Submit New Benchmark

Basic task properties

Name	example-benchmark
Type	BENCHMARK
Group ID	cz.cuni.mff.d3s.been
Bpk ID	task-test
Version	3.0.0-SNAPSHOT
Exclusivity	NON_EXCLUSIVE

Environment properties

from	10
to	20

Arguments

Argument -Dargset=false

Other

Description

Java options

Option -Xmx512m

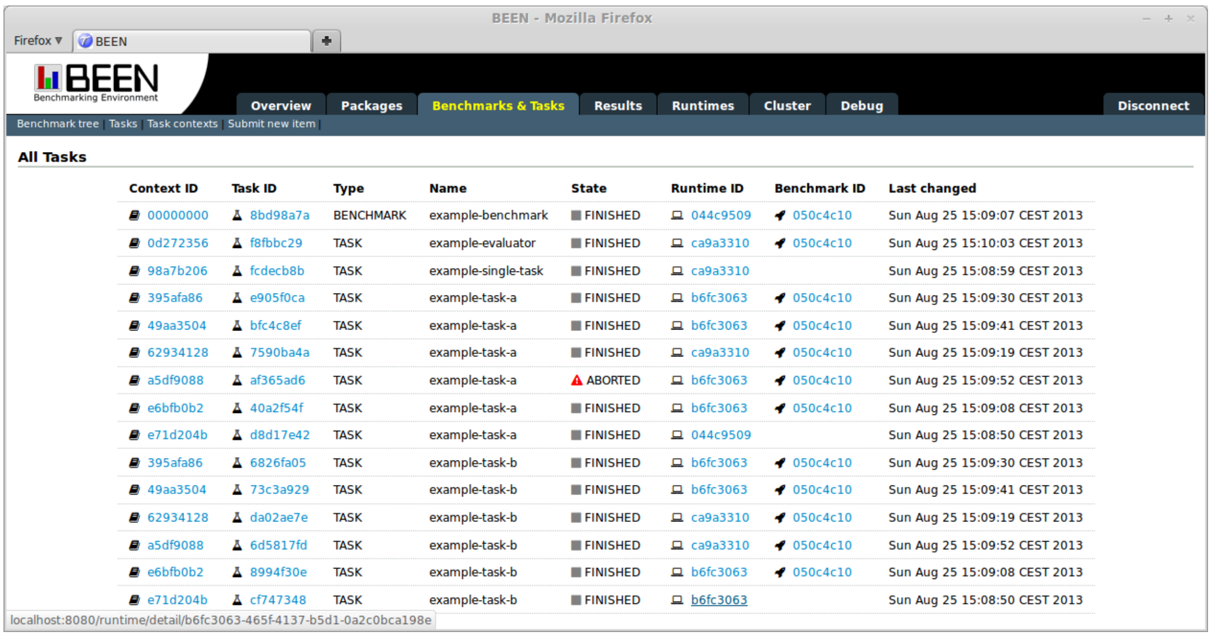
Failure policy

Timeout for single run (0 = disabled)	0
Max. number of restarts (0 = disabled)	0

Debug options

Host Runtime Filter

Figure 2.11: Submitting a benchmark

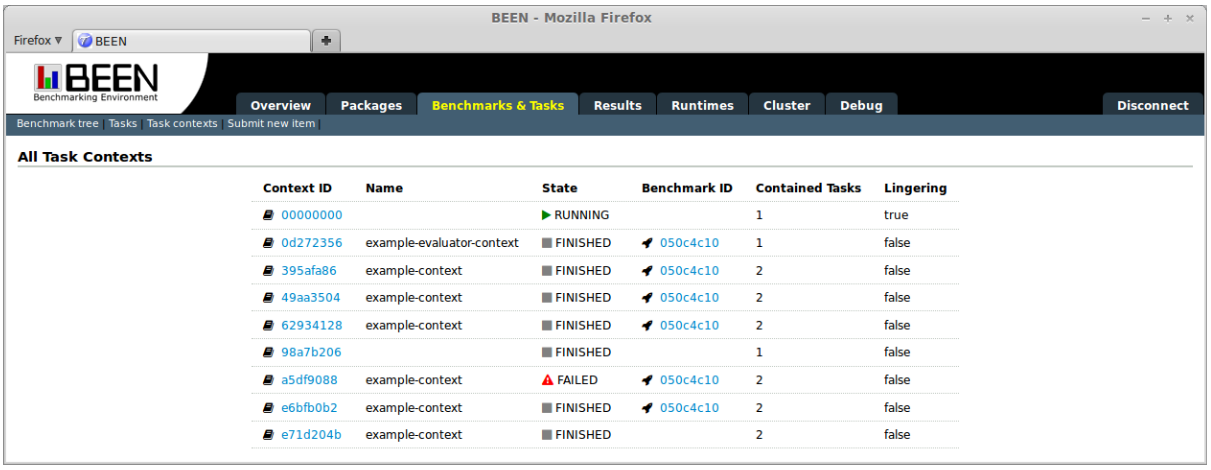


All Tasks

Context ID	Task ID	Type	Name	State	Runtime ID	Benchmark ID	Last changed
00000000	8bd98a7a	BENCHMARK	example-benchmark	FINISHED	044c9509	050c4c10	Sun Aug 25 15:09:07 CEST 2013
0d272356	f8fbbc29	TASK	example-evaluator	FINISHED	ca9a3310	050c4c10	Sun Aug 25 15:10:03 CEST 2013
98a7b206	fcdecb8b	TASK	example-single-task	FINISHED	ca9a3310		Sun Aug 25 15:08:59 CEST 2013
395afa86	e905f0ca	TASK	example-task-a	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:30 CEST 2013
49aa3504	bfc4c8ef	TASK	example-task-a	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:41 CEST 2013
62934128	7590ba4a	TASK	example-task-a	FINISHED	ca9a3310	050c4c10	Sun Aug 25 15:09:19 CEST 2013
a5df9088	af365ad6	TASK	example-task-a	⚠ ABORTED	b6fc3063	050c4c10	Sun Aug 25 15:09:52 CEST 2013
e6bfb0b2	40a2f54f	TASK	example-task-a	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:08 CEST 2013
e71d204b	d8d17e42	TASK	example-task-a	FINISHED	044c9509		Sun Aug 25 15:08:50 CEST 2013
395afa86	6826fa05	TASK	example-task-b	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:30 CEST 2013
49aa3504	73c3a929	TASK	example-task-b	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:41 CEST 2013
62934128	da02ae7e	TASK	example-task-b	FINISHED	ca9a3310	050c4c10	Sun Aug 25 15:09:19 CEST 2013
a5df9088	6d5817fd	TASK	example-task-b	FINISHED	ca9a3310	050c4c10	Sun Aug 25 15:09:52 CEST 2013
e6bfb0b2	8994f30e	TASK	example-task-b	FINISHED	b6fc3063	050c4c10	Sun Aug 25 15:09:08 CEST 2013
e71d204b	cf747348	TASK	example-task-b	FINISHED	b6fc3063		Sun Aug 25 15:08:50 CEST 2013

localhost:8080/runtime/detail/b6fc3063-465f-4137-b5d1-0a2c0bca198e

Figure 2.12: Listing tasks



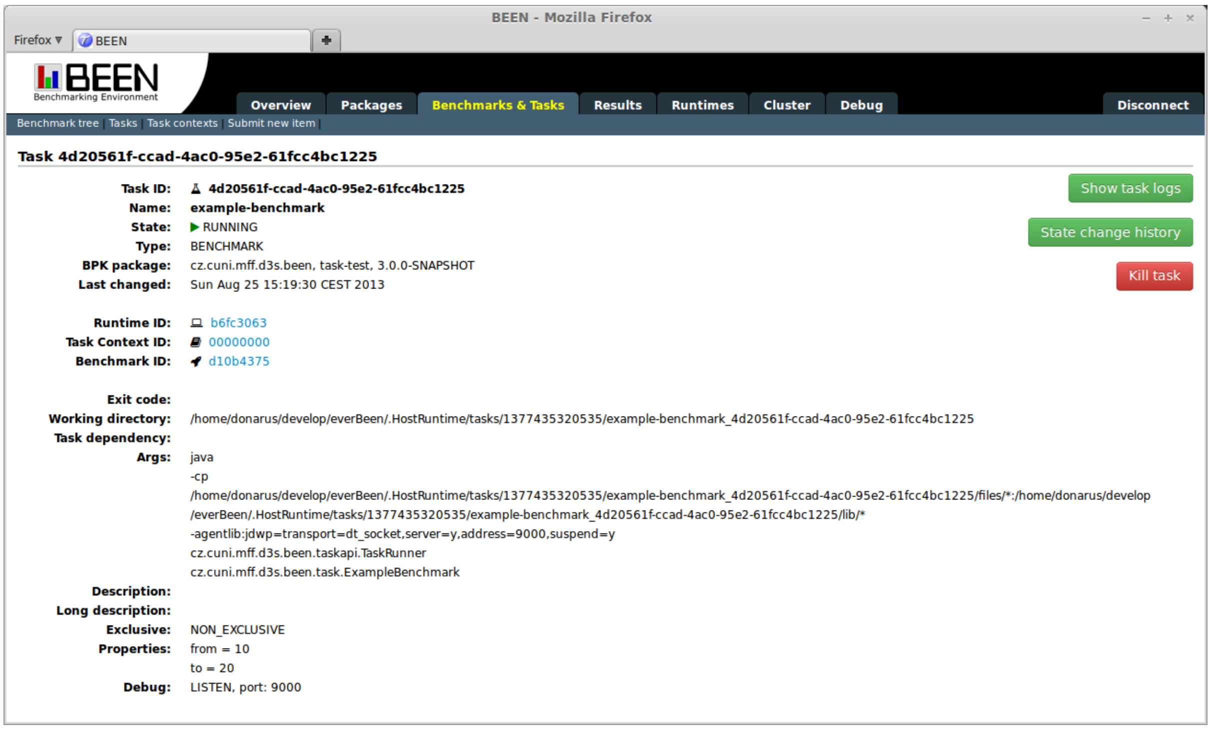
All Task Contexts

Context ID	Name	State	Benchmark ID	Contained Tasks	Lingering
00000000		▶ RUNNING		1	true
0d272356	example-evaluator-context	■ FINISHED	050c4c10	1	false
395afa86	example-context	■ FINISHED	050c4c10	2	false
49aa3504	example-context	■ FINISHED	050c4c10	2	false
62934128	example-context	■ FINISHED	050c4c10	2	false
98a7b206		■ FINISHED		1	false
a5df9088	example-context	⚠ FAILED	050c4c10	2	false
e6bfb0b2	example-context	■ FINISHED	050c4c10	2	false
e71d204b	example-context	■ FINISHED		2	false

Figure 2.13: Listing task contexts

2.4.9 Task, task context and benchmark detail

To see a task detail (*Figure 2.14*), task context detail (*Figure 2.15*) or benchmark detail (*Figure 2.15*), click on its id anywhere on the page. If the task, context or benchmark is running, you can kill it by clicking on the **kill** button in the top right corner of the page. If the task, context or benchmark is finished or failed, you will see the **remove** button instead of the kill button in top right of the page. Click on the button to delete all results, logs and all service information about task from EverBEEN.



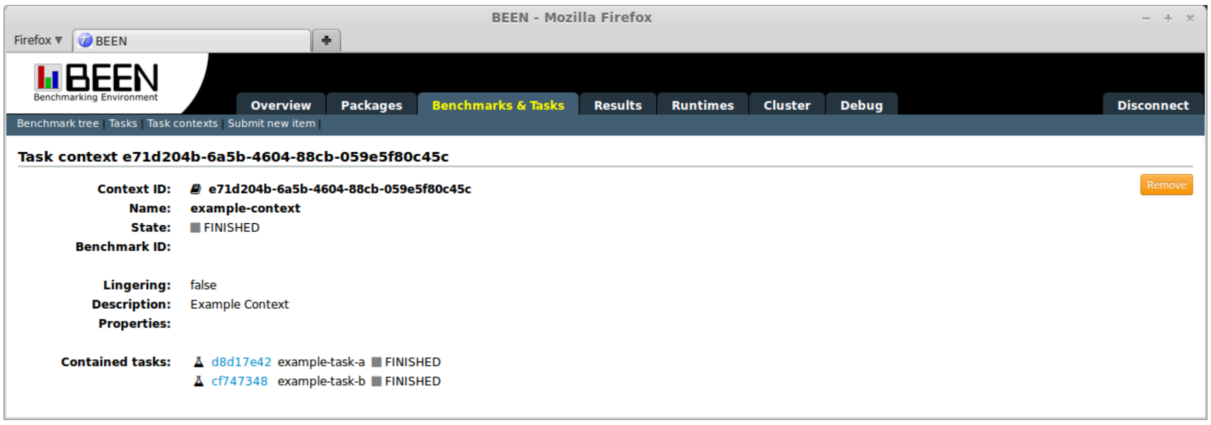
Task 4d20561f-ccad-4ac0-95e2-61fcc4bc1225

Task ID:	**4d20561f-ccad-4ac0-95e2-61fcc4bc1225**
Name:	**example-benchmark**
State:	▶ RUNNING
Type:	BENCHMARK
BPK package:	cz.cuni.mff.d3s.been, task-test, 3.0.0-SNAPSHOT
Last changed:	Sun Aug 25 15:19:30 CEST 2013
Runtime ID:	b6fc3063
Task Context ID:	00000000
Benchmark ID:	d10b4375
Exit code:	
Working directory:	/home/donarus/develop/everBeen/.HostRuntime/tasks/1377435320535/example-benchmark_4d20561f-ccad-4ac0-95e2-61fcc4bc1225
Task dependency:	
Args:	java -cp /home/donarus/develop/everBeen/.HostRuntime/tasks/1377435320535/example-benchmark_4d20561f-ccad-4ac0-95e2-61fcc4bc1225/files/*:/home/donarus/develop /everBeen/.HostRuntime/tasks/1377435320535/example-benchmark_4d20561f-ccad-4ac0-95e2-61fcc4bc1225/lib/* -agentlib:jdwp=transport=dt_socket,server=y,address=9000,suspend=y cz.cuni.mff.d3s.been.taskapi.TaskRunner cz.cuni.mff.d3s.been.task.ExampleBenchmark
Description:	
Long description:	
Exclusive:	NON_EXCLUSIVE

Properties:	from = 10
	to = 20

Debug:	LISTEN, port: 9000

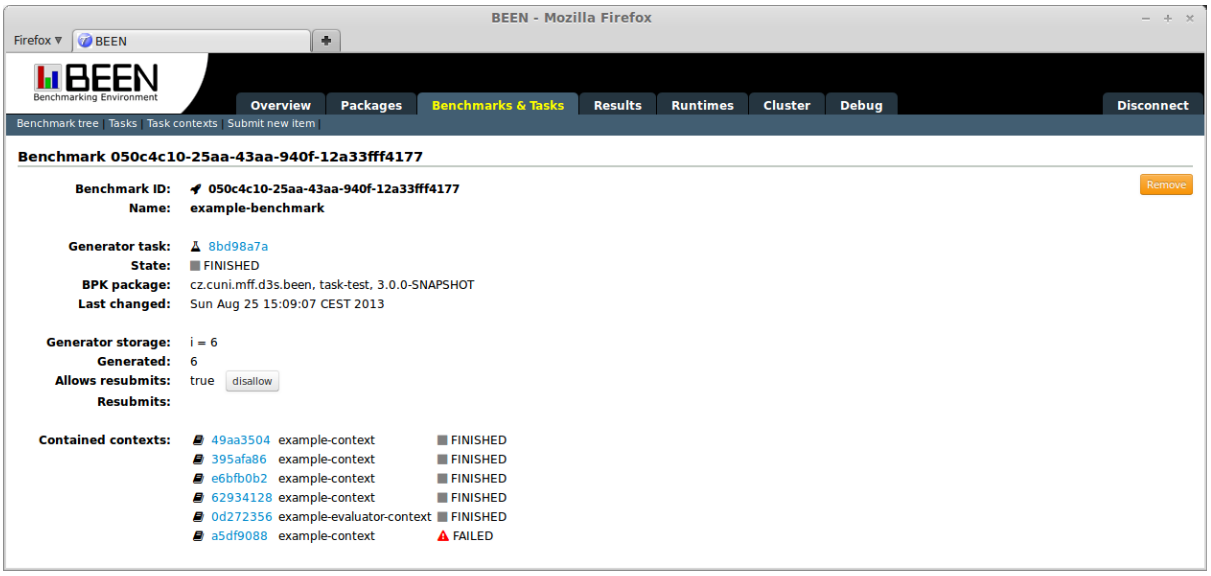
Figure 2.14: Task detail



Task context e71d204b-6a5b-4604-88cb-059e5f80c45c

Context ID:	e71d204b-6a5b-4604-88cb-059e5f80c45c
Name:	example-context
State:	FINISHED
Benchmark ID:	
Lingering:	false
Description:	Example Context
Properties:	
Contained tasks:	d8d17e42 example-task-a FINISHED
	cf747348 example-task-b FINISHED

Figure 2.15: Task context detail



Benchmark 050c4c10-25aa-43aa-940f-12a33fff4177

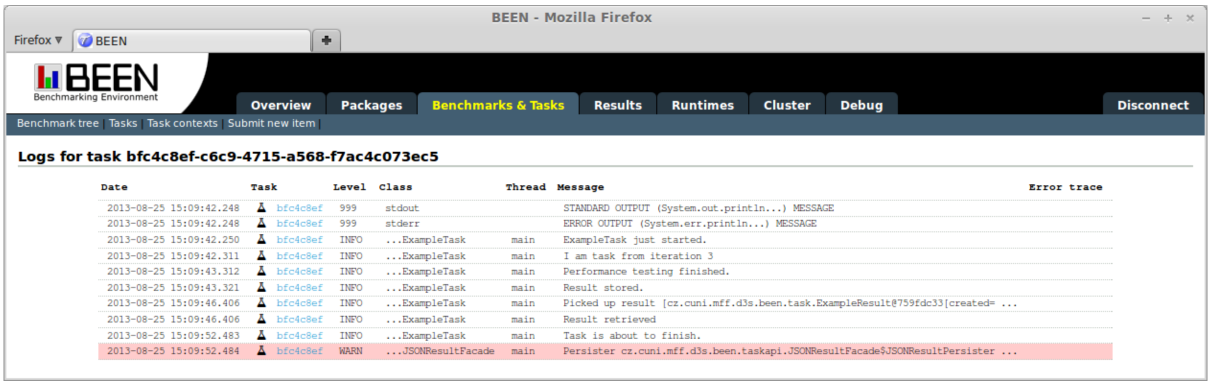
Benchmark ID:	050c4c10-25aa-43aa-940f-12a33fff4177
Name:	example-benchmark
Generator task:	8bd98a7a
State:	FINISHED
BPK package:	cz.cuni.mff.d3s.been, task-test, 3.0.0-SNAPSHOT
Last changed:	Sun Aug 25 15:09:07 CEST 2013
Generator storage:	i = 6
Generated:	6
Allows resubmits:	true
Resubmits:	

Contained contexts:			
	49aa3504	example-context	FINISHED
	395afa86	example-context	FINISHED
	e6bfb0b2	example-context	FINISHED
	62934128	example-context	FINISHED
	0d272356	example-evaluator-context	FINISHED
	a5df9088	example-context	FAILED

Figure 2.16: Benchmark detail

2.4.10 Displaying logs from tasks

To display logs from tasks (*Figure 2.17*), go to the page with task details and press the **show logs** button in the top right corner of the page.

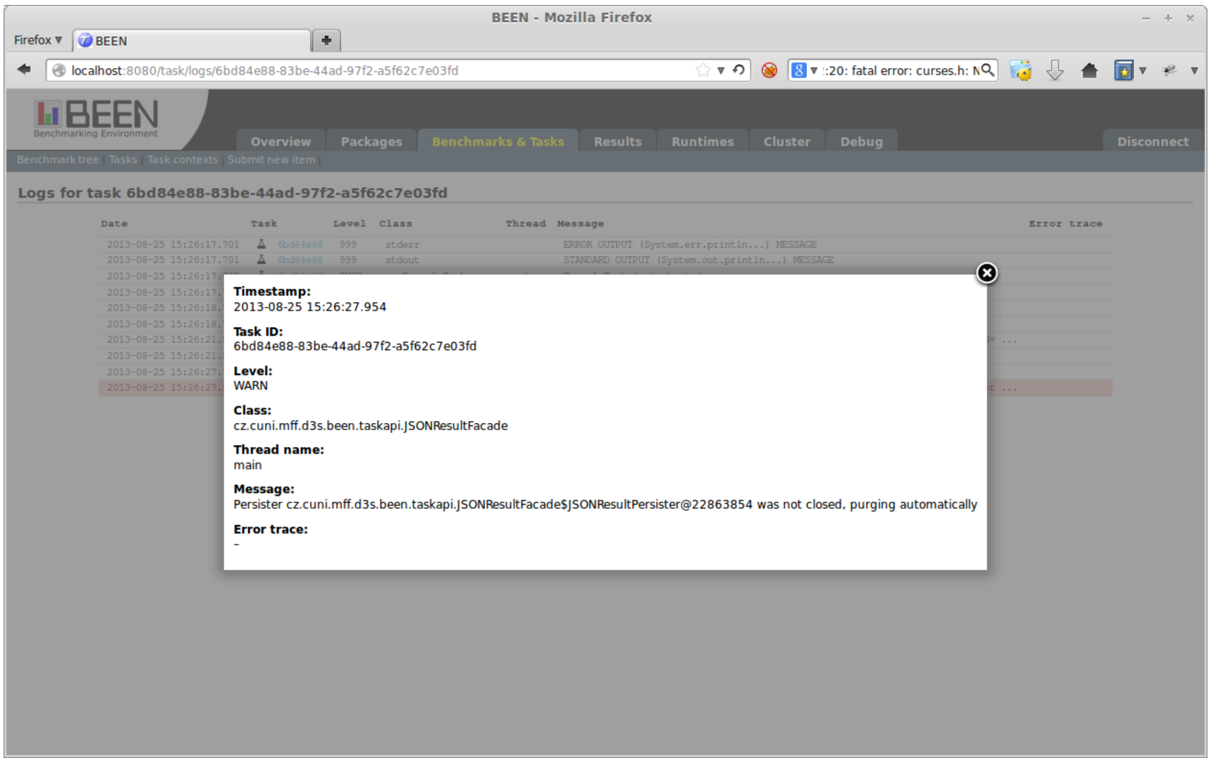


Logs for task bfc4c8ef-c6c9-4715-a568-f7ac4c073ec5

```
Date                      Task       Level  Class                Thread  Message                                                                     Error trace
2013-08-25 15:09:42.248   bfc4c8ef   999    stdout                       STANDARD OUTPUT (System.out.println...) MESSAGE
2013-08-25 15:09:42.248   bfc4c8ef   999    stderr                       ERROR OUTPUT (System.err.println...) MESSAGE
2013-08-25 15:09:42.250   bfc4c8ef   INFO   ...ExampleTask       main    ExampleTask just started.
2013-08-25 15:09:42.311   bfc4c8ef   INFO   ...ExampleTask       main    I am task from iteration 3
2013-08-25 15:09:43.312   bfc4c8ef   INFO   ...ExampleTask       main    Performance testing finished.
2013-08-25 15:09:43.321   bfc4c8ef   INFO   ...ExampleTask       main    Result stored.
2013-08-25 15:09:46.406   bfc4c8ef   INFO   ...ExampleTask       main    Picked up result [cz.cuni.mff.d3s.been.task.ExampleResult@759fdc33[created= ...
2013-08-25 15:09:46.406   bfc4c8ef   INFO   ...ExampleTask       main    Result retrieved
2013-08-25 15:09:52.483   bfc4c8ef   INFO   ...ExampleTask       main    Task is about to finish.
2013-08-25 15:09:52.484   bfc4c8ef   WARN   ...JSONResultFacade  main    Persister cz.cuni.mff.d3s.been.taskapi.JSONResultFacade$JSONResultPersister ...
```

Figure 2.17: Task logs

If you want to see detailed information, e.g. a stack trace (*Figure 2.18*), click on the line with an appropriate log message.



BEEN - Mozilla Firefox
Firefox
BEEN
localhost:8080/task/logs/6bd84e88-83be-44ad-97f2-a5f62c7e03fd
:20: fatal error: curses.h: N
BEEN Benchmarking Environment
Overview
Packages
Benchmarks & Tasks
Results
Runtimes
Cluster
Debug
Disconnect
Benchmark tree | Tasks | Task contexts | Submit new item |

Logs for task 6bd84e88-83be-44ad-97f2-a5f62c7e03fd

```
Date                      Task       Level  Class    Thread  Message                                             Error trace
2013-08-25 15:26:17.701   6bd84e88   999    stderr           ERROR OUTPUT (System.err.println...) MESSAGE
2013-08-25 15:26:17.701   6bd84e88   999    stdout           STANDARD OUTPUT (System.out.println...) MESSAGE
```

Timestamp:
2013-08-25 15:26:27.954

Task ID:
6bd84e88-83be-44ad-97f2-a5f62c7e03fd

Level:
WARN

Class:
cz.cuni.mff.d3s.been.taskapi.JSONResultFacade

Thread name:
main

Message:
Persister cz.cuni.mff.d3s.been.taskapi.JSONResultFacade$JSONResultPersister@22863854 was not closed, purging automatically

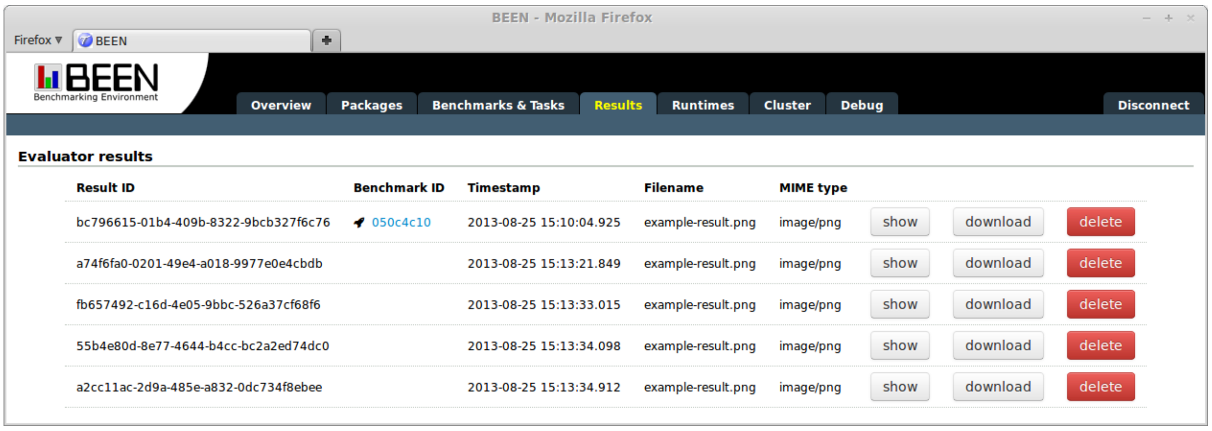
Error trace:
-

Figure 2.18: Detail of a task log

2.4.11 Listing and displaying evaluator results

To list results (*Figure 2.19*), switch to the **Results** tab. The page lists all Evaluators results. You can download or delete them. Currently only evaluator results can be displayed and downloaded directly through the Web Interface.

You can also display an evaluator result directly (*Figure 2.20*), but its `MIME` type must be supported. Supported MIME types are:

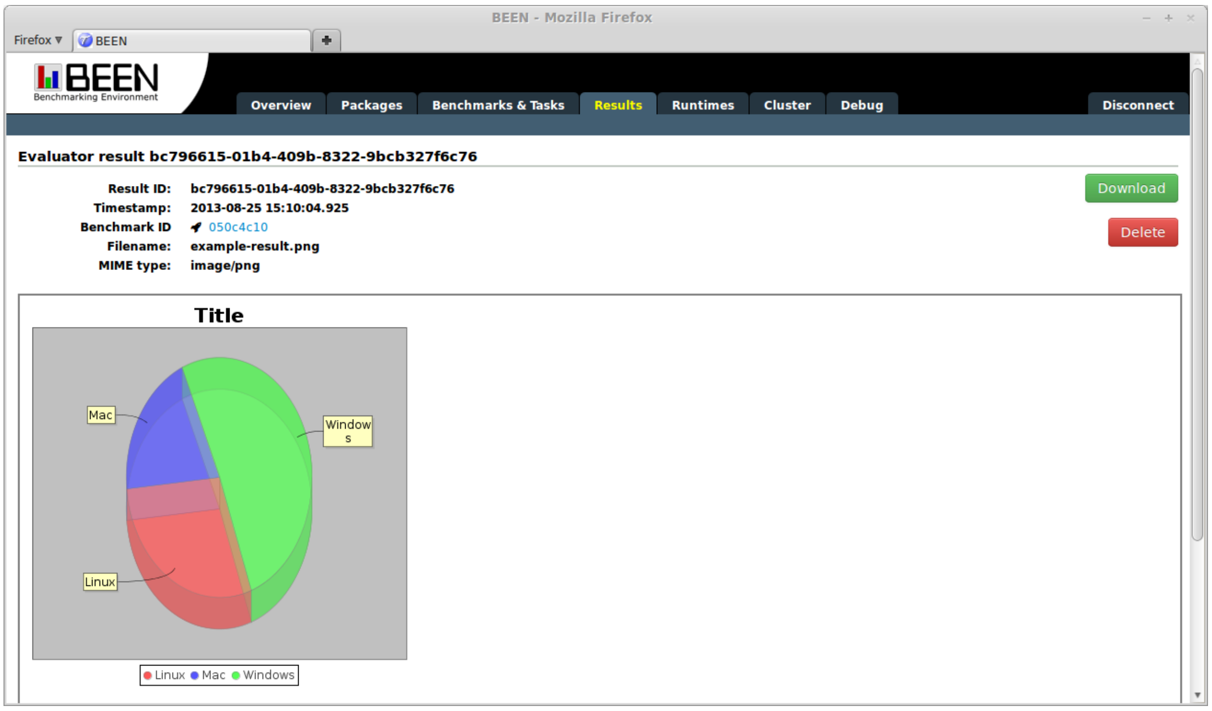


Evaluator results

Result ID	Benchmark ID	Timestamp	Filename	MIME type			
bc796615-01b4-409b-8322-9bcb327f6c76	050c4c10	2013-08-25 15:10:04.925	example-result.png	image/png	show	download	delete
a74f6fa0-0201-49e4-a018-9977e0e4cbdb		2013-08-25 15:13:21.849	example-result.png	image/png	show	download	delete
fb657492-c16d-4e05-9bbc-526a37cf68f6		2013-08-25 15:13:33.015	example-result.png	image/png	show	download	delete
55b4e80d-8e77-4644-b4cc-bc2a2ed74dc0		2013-08-25 15:13:34.098	example-result.png	image/png	show	download	delete
a2cc11ac-2d9a-485e-a832-0dc734f8ebee		2013-08-25 15:13:34.912	example-result.png	image/png	show	download	delete

Figure 2.19: Listing of evaluator results

- image/png
- image/jpeg
- image/gif
- text/html
- text/plain



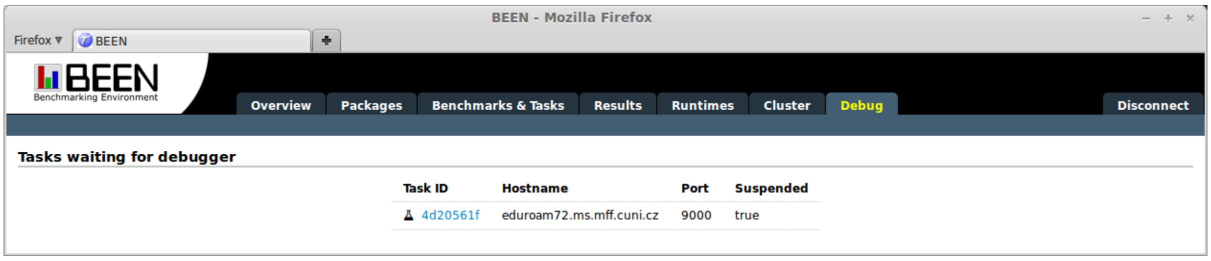
Evaluator result bc796615-01b4-409b-8322-9bcb327f6c76

Result ID:	bc796615-01b4-409b-8322-9bcb327f6c76
Timestamp:	2013-08-25 15:10:04.925
Benchmark ID	050c4c10
Filename:	example-result.png
MIME type:	image/png

Figure 2.20: Example of an evaluator result

2.4.12 Debugging tasks

To see which tasks running in *listen* debug mode (*Figure 2.21*), switch to the **Debug** tab. The page displays information about host names and ports where the Java debugger can connected. Currently only JVM-based task can be debugged.



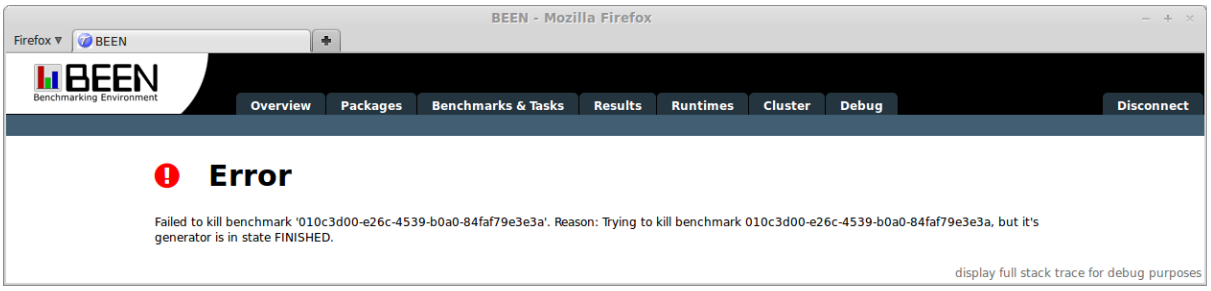
Tasks waiting for debugger

Task ID	Hostname	Port	Suspended
4d20561f	eduroam72.ms.mff.cuni.cz	9000	true

Figure 2.21: Debug page

2.4.13 Handling web interface errors

If something goes wrong or you are trying to invoke an invalid operation, the web interface will present a simple error message (*Figure 2.22*).



Error

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

Figure 2.22: Error page example

If you are interested in the stack trace of the error, click on the **show detailed stack trace** link in bottom right corner of the page.

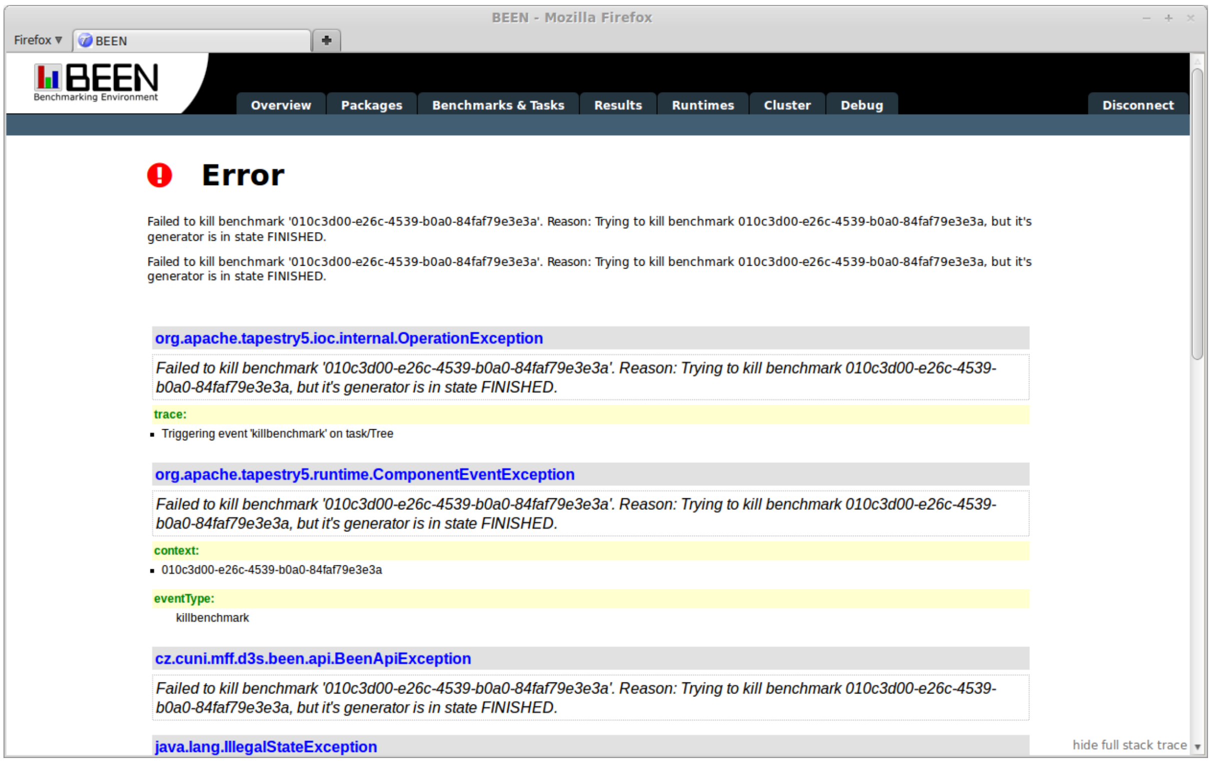
2.5 Task and Benchmark API

One of the main goals of the EverBEEN project was to make the task API as simple as possible and to minimize the amount of work needed to create a benchmark.

EverBEEN works with three concepts of user-supplied code and configuration:

- **Task**, is an elementary unit of code that can be submitted to and run by EverBEEN. Tasks are created by subclassing the abstract `Task` class and implementing appropriate methods. Each task has to be described by a XML **task descriptor** which specifies the main class to run and parameters of the task.
- **Task context** is a container for multiple tasks. Containers can interact, pass data to each other and synchronize among themselves. Tasks contexts do not contain any user-written code, they only serve as wrappers for the contained tasks. Each task context is described by a XML **task context descriptor** that specifies which tasks should be contained within the context.
- **Benchmark** is a first-class object that *generates* task contexts based on its **generator task**, which is again a user-written code created by subclassing the abstract `Benchmark` class. Each benchmark is described by a XML **benchmark descriptor** which specifies the main class to run and parameters of the benchmark. A benchmark is different from a task, because its API provides features for generating task contexts and it can also persist its state so it can be re-run when an error occurs and the generator task fails.

All these three concepts can be submitted to EverBEEN and run individually, e.g. if you only want to test a single task, you can submit it without providing a task context or benchmark.



Error

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

org.apache.tapestry5.ioc.internal.OperationException

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

trace:
- Triggering event 'killbenchmark' on task/Tree

org.apache.tapestry5.runtime.ComponentEventException

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

context:
- 010c3d00-e26c-4539-b0a0-84faf79e3e3a

eventType:
killbenchmark

cz.cuni.mff.d3s.been.api.BeenApiException

Failed to kill benchmark '010c3d00-e26c-4539-b0a0-84faf79e3e3a'. Reason: Trying to kill benchmark 010c3d00-e26c-4539-b0a0-84faf79e3e3a, but it's generator is in state FINISHED.

java.lang.IllegalStateException

Figure 2.23: Debugging the Web Interface

2.5.1 Maven Plugin and Packaging

The easiest way to create a submittable item (e.g. a task) is by creating a Maven project and adding a dependency on the appropriate EverBEEN module (e.g. `task-api`) in the `pom.xml` of the project:

```
<dependency>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>task-api</artifactId>
    <version>3.0.0-SNAPSHOT</version>
</dependency>
```

Tasks, contexts and benchmark must be packaged into a BPK file, which can then be uploaded to EverBEEN. Each BPK package can contain multiple submittable items and multiple XML descriptors. The problem of packaging is made easier by the supplied `bpk-maven-plugin` Maven plugin. The preferred way to use it is to add the plugin to the `package` Maven goal in `pom.xml` of the project:

```
<plugin>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>bpk-maven-plugin</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <executions>
        <execution>
            <goals>
                <goal>buildpackage</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        ...
    </configuration>
</plugin>
```

In the plugin's configuration the user must specify at least one descriptor of a task, context or benchmark. To add a descriptor into the BPK, it should be added as a standard Java resource file and then referenced in the plugin configuration by using a `<taskDescriptors>` or `<taskContextDescriptors>` element. For example, the provided sample benchmark called `nginx-benchmark` uses this configuration:

```
<configuration>
    <taskDescriptors>
        <param>src/main/resources/cz/cuni/mff/d3s/been/nginx/NginxBenchmark.td.xml</param>
    </taskDescriptors>
</configuration>
```

This specifies that the package should publish a single descriptor named `NginxBenchmark.td.xml` which is located in the specified resource path. With such a configuration, creating the BPK package is simply a matter of invoking `mvn package` on the project — it will produce a `.bpk` file that can be uploaded to EverBEEN.

2.5.1.1 Maven repositories

Maven repositories are available. Put the following declarations into the `pom.xml` to transparently resolve dependencies:

```
<pluginRepositories>
    <pluginRepository>
        <id>everbeen.cz-plugins-snapshots</id>
        <url>http://everbeen.cz/artifactory/plugins-snapshot-local</url>
    </pluginRepository>
</pluginRepositories>

<repositories>
    <repository>
        <id>everbeen.cz-snapshots</id>
        <url>http://everbeen.cz/artifactory/libs-snapshot-local</url>
    </repository>
</repositories>
```

The current version of the `bpk-maven-plugin` is `1.0.0-SNAPSHOT`.

2.5.2 Descriptor Format

There are two types of descriptors, task descriptors and task context descriptors. Note that benchmarks don't have a special descriptor format, instead you only provide a task descriptor for a generator task of the benchmark. These descriptors are written in XML and they must conform to the supplied XSD definitions (task-descriptor.xsd[2] and task-context-descriptor.xsd[3]).

The recommended naming practice is to name your task descriptors with the filename ending with `.td.xml` and your task context descriptors ending with `.tcd.xml`.

A simple task descriptor for a single task can look like this:

```
<?xml version="1.0"?>
<taskDescriptor xmlns="http://been.d3s.mff.cuni.cz/task-descriptor"
                groupId="my.sample.benchmark" bpkId="hello-world" version="3.0.0-SNAPSHOT"
                name="hello-world-task" type="task">
    <java>
```

[2]http://www.everbeen.cz/xsd/task-descriptor.xsd
[3]http://www.everbeen.cz/xsd/task-context-descriptor.xsd

```
        <mainClass>my.sample.benchmark.HelloWorldTask</mainClass>
    </java>
</taskDescriptor>
```

It specifies the main class and package that should be used to run the task. Apart from this, you can specify what parameters the task should receive and their default values:

```
<properties>
    <property name="key">value</property>
</properties>
```

These properties will be presented to the user in the web interface before submitting the task and the user can modify them. Next, you can specify command line arguments passed to Java:

```
<arguments>
    <argument>-Xms4m</argument>
    <argument>-Xmx8m</argument>
</arguments>
```

For debugging purposes, you can specify the `<debug>` element which will enable remote debugging when running the task (also available from the Web Interface).

2.5.2.1 Host Runtime selection

With the `<hostRuntimes>` element you can constrain the Host Runtimes the task can be run on. The value of this setting is an expression in XML Path Language (XPath) Version 1.0[4].

The most useful options for host selection are presented here. For full specification see runtime-info.xsd[5], hardware-info.xsd[6], monitor.xsd[7]

Basic Information about a Host Runtime

```
/id
/port
/host
```

Java runtime specification

```
/java/version
/java/vendor
/java/runtimeName
/java/VMVersion
/java/VMVendor
/java/runtimeVersion
/java/specificationVersion
```

Operation system information

```
/operatingSystem/name
/operatingSystem/version
/operatingSystem/arch
/operatingSystem/vendor
/operatingSystem/vendorVersion
/operatingSystem/dataModel
/operatingSystem/endian
```

[4]http://www.w3.org/TR/xpath
[5]http://www.everbeen.cz/xsd/runtime-info.xsd
[6]http://www.everbeen.cz/xsd/hardware-info.xsd
[7]http://www.everbeen.cz/xsd/monitor.xsd

CPU information (there can be multiply CPUs)

```
/hardware/cpu/vendor
/hardware/cpu/model
/hardware/cpu/mhz
/hardware/cpu/cacheSize
```

File system information (there can be multiply file systems)

```
/filesystem/deviceName
/filesystem/directory
/filesystem/type
/filesystem/free
/filesystem/total
```

Network information (there can be multiply network interfaces)

```
/hardware/networkInterface/name
/hardware/networkInterface/hwaddr
/hardware/networkInterface/type
/hardware/networkInterface/mtu
/hardware/networkInterface/netmask
/hardware/networkInterface/broadcast
/hardware/networkInterface/address
```

Main memory information

```
/hardware/memory/ram
/hardware/memory/swap
```

Examples

The following example will select the Host Runtime with host name `eduroam40.ms.mff.cuni.cz`.

```
<hostRuntimes>
    <xpath>host = "eduroam40.ms.mff.cuni.cz"</xpath>
</hostRuntimes>
```

```
<hostRuntimes>
    <xpath>//networkInterface[address = "195.113.16.40"]</xpath>
</hostRuntimes>
```

Selects the Host Runtime with an IPv4 address of `195.113.16.40`.

```
<hostRuntimes>
    <xpath>/hardware/networkInterface[contains(address,"195.113.16")]</xpath>
</hostRuntimes>
```

Selects all Host Runtimes whose IP address contains “195.113.16”.

```
<hostRuntimes>
    <xpath>contains(/operatingSystem/name, "Linux")</xpath>
</hostRuntimes>
```

Selects all Linux operating systems.

Selection expression can be tested on the `runtime/list` page in the Web Interface.

2.5.3 Task API

To create a task submittable into EverBEEN, you should start by subclassing the `Task`[8] abstract class. The `run` method needs to be overridden.

EverBEEN uses SLF4J[9] as its logging mechanism and provides a logging backend for all user-written code. This means that you can simply use the standard loggers and any logs will be automatically stored through EverBEEN.

Knowing this, the simplest task that will only log a message looks like this:

```
package my.sample.benchmark;

import cz.cuni.mff.d3s.been.taskapi.Task;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorldTask extends Task {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldTask.class);

    @Override
    public void run(String[] args) {
        log.info("Hello, world!");
    }
}
```

If this class is in a Maven project as described in the 2.5.1 (Maven Plugin and Packaging) section, it can be packaged into a BPK package by invoking `mvn package`. This package can be uploaded and run from the Web Interface.

BEEN provides several APIs for user-written tasks:

- *Properties* — Tasks are configurable either from their descriptors or by the benchmark that generated them. These properties are again configurable by the user before submitting the task. All properties have a name and a simple string value and these can be accessed via the `getTaskProperty` method of the abstract `Task` class.
- *Result storing* — Each task can persist a result that it has gathered by using the API providing access to the persistence layer. To store a result, use a `Persister` object, which can be created by using the method `createResultPersister` from the `Task` abstract class.
- *Synchronization and communication* — When multiple tasks run in a task context, they can interact with each other either for synchronization purposes or to exchange data. API is provided by the `CheckpointController`[10] class. EverBEEN provides the concepts of **checkpoints** and **latches**. Latches serve as context-wide atomic numbers with the methods for setting a value, decreasing the value by one and waiting until the latch reaches zero. Checkpoints are also waitable objects that store a value.

2.5.4 Task Properties

Every task has a key-value property storage. These properties can be set from various places: From the XML descriptor, from user input when submitting, inherited from a task context or set from a benchmark when it generates a task context. To access these values, you can use the `getTaskProperty` method of the `Task` class:

```
int numberOfClients = Integer.parseInt(this.getTaskProperty("numberOfClients"));
```

[8] http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/Task.html
[9] http://www.slf4j.org/
[10] http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/CheckpointController.html

These properties are inherited, in the sense that that when a task context has a property, the task can see it as well. But when a task has the same property with a different value, the task's value will be override the previous one.

One important property recognized by the Task API is `task.log.level` which sets the log level for a task. The property can have the following values (in increasing severity):

- TRACE
- DEBUG
- INFO
- WARN
- ERROR

The default log level is INFO.

***Warning* for the TRACE log level**
Note that the TRACE log level is used by the Task API (instead of the DEBUG level which is reserved for user code). Setting the TRACE level will print Task API debug messages.

2.5.5 Persisting Results

The persistence layer provided by EverBEEN is capable of storing user-supplied types and classes. To create a class that can be persisted, simply create a subclass of the `Result`[11] class and ensure that all contained fields are serializable. Also make sure to include a default non-parameterized constructor so that the object can be deserialized.

Each result type is identified with a string `Group ID` (we recommend to create a constant). The Group ID is an identification of a group of related results - each benchmark should use its unique own `Group ID(s)`. A naming convection is recommended to distinguish between multiple types of results. An example of a result:

```
public class SampleResult extends Result {
    public static final String GROUP_ID = "example-data";

    public int data;

    public SampleResult() {}
}
```

All fields will be stored (even private). Setters and getters are not necessary but still recommended.

Persisting the result is then only a simple matter of creating the appropriate object, instantiating the `Persister`[12] class through the supplied `results` field and calling `persist` on it:

```
SampleResult result = results.createResult(SampleResult.class);
Persister persister = results.createResultPersister(SampleResult.GROUP_ID));
persister.persist(result);
```

The `results.createResult(SampleResult.class)` call properly initializes results with *taskId*, *contextId* and (if running as part of a benchmark) *benchmarkId*. These parameters are useful in identifying results.

The `Persister` can be reused, but the `close()` method should be called once you are done with it. The best way to achieve this is to use the try-with-resources statement (the Persister implements `AutoCloseable`):

```
try (Persister persister = results.createResultPersister(SampleResult.GROUP_ID)) {
    persister.persist(result);
}
```

[11]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/results/Result.html
[12]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/Persister.html

2.5.6 Querying Results

Tasks can also query stored results. Note that results storage is asynchronous and may take some time. Usually this is not a problem. Blocking results persistence is a planned feature.

First a `Query`[13] specifying what results to select must be built using the `ResultQueryBuilder`[14]. The ResultQueryBuilder uses a fluent API to build a query.

Following example creates a query which will fetch results from the SampleResult.GROUP_ID group, requesting that the *taskId* property is set to the ID of the current task and data property is 47;

```
Query query = new ResultQueryBuilder().on(SampleResult.GROUP_ID)
    .with("taskId", getId()).with("data", 47).fetch();
```

The query can be now used to fetch a collection of results, again using the `results` helper object which is part of the `Task` object:

```
Collection<ExampleResult> taskResults = results.query(query, ExampleResult.class);
```

Currently tasks can only fetch results, not delete them (this is design decision, the code is fully capable of issuing deletes).

An overview of the ResultQueryBuilder API follows:

`public ResultQueryBuilder on(String group)`
Sets the Group ID of the results to fetch.

`public ResultQueryBuilder with(String attribute, Object value)`
Adds a criterion to the query, where the `attribute` is the name of the property, and `value` is the expected value of the property.

`public ResultQueryBuilder without(String attribute)`
Removes a criterion from the query, the value of `attribute` will not be fetched (beware of NullPointerExceptions).

`public ResultQueryBuilder retrieving(String... attributes)`
Sets attributes to fetch. Other attributes will be omitted and will not be set.

2.5.7 Checkpoints and Latches

Checkpoints provide a powerful mechanism for synchronization and communication among tasks contained in a single context. Tasks can wait for the value of a Checkpoint (most usually set by another task). This waiting is passive and once a value is assigned to a checkpoint, the waiter will receive it.

To use checkpoints, create a `CheckpointController`[15], which is an `AutoCloseable`[16] object so the preferred way to use it is inside the try-with-resources block to ensure the object will be properly destroyed:

```
try (CheckpointController requestor = CheckpointController.create()) {
    ...
} catch (MessagingException e) {
    ...
}
```

[13]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/Query.html
[14]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/ResultQueryBuilder.html
[15]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/CheckpointController.html
[16]http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html

Each checkpoint has a name, which is context-wide. You don't have to explicitly create a checkpoint, it will be created automatically once a task uses it. Setting a value to a checkpoint can be done with:

```
requestor.checkPointSet("mycheckpoint", "the value");
```

A typical scenario is that one tasks wants to wait for another to pass a value. To wait until a value is set and also to receive the value you can use:

```
String value = requestor.checkPointWait("mycheckpoint");
```

This call passively waits (possibly indefinitely) until a value is set to the checkpoint. There is also a variant of this method that takes another argument specifying a timeout, after which the call will throw an exception. Another method called `checkPointGet` can be used to retrieve the current value of a checkpoint without waiting.

Checkpoints initially do not have any value, and once a value is set, it cannot be changed. They work as a proper synchronization primitive, and setting a value is an atomic operation. The semantics don't change if you start waiting before or after the value is set.

Another provided synchronization primitive is a *latch*. Latches work best for implementing rendez-vous synchronization. A latch provides a method to set an integer value:

```
requestor.latchSet("mylatch", 5);
```

Another task can then call an atomic method to decrease the value of the latch:

```
requestor.latchCountDown("mylatch");
```

You can then wait until the value reaches zero:

```
requestor.latchWait("mylatch");
```

All operations on latches are atomic and the waiting is passive. Latches has to be created (by the set method) before calling the count down or wait operation.

2.5.8 Benchmark API

Writing a benchmark's generator task is similar to writing an ordinary task in the sense that you have to write a subclass, package it and run it on a Host Runtime. However, the benchmark API is different, because the purpose of the benchmark is to provide long-running code that will eventually generate new task contexts.

To create a benchmark, subclass the abstract `Benchmark`[17] class and implement appropriate methods. The main method to implement is the `generateTaskContext` which is called periodically by EverBEEN `Benchmark API` and it is expected to return a newly generated task context. This context is then submitted and run. When the context finishes, this method is called again. The loop ends whenever the method returns `null`.

This approach is chosen to cover several possible use cases. When the benchmark does not have data for a new task context, it can simply block until it is possible to create a new context. On the other hand, the benchmark cannot overhaul the cluster by submitting too many contexts. Instead, it's up to the cluster to call the `generateTaskContext` method whenever it seems fit.

For creating task contexts you should use the provided `ContextBuilder`[18] class. This supports loading a task context from a XML file, modifying it and setting values of properties inside the context descriptor. If you have a prepared `.tcd` file with a context descriptor, a sample benchmark that will indefinitely generate this context can look like this:

[17] http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/benchmarkapi/Benchmark.html
[18] http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/benchmarkapi/ContextBuilder.html

```
package my.sample.benchmark;

import cz.cuni.mff.d3s.been.benchmarkapi.Benchmark;
import cz.cuni.mff.d3s.been.benchmarkapi.BenchmarkException;
import cz.cuni.mff.d3s.been.benchmarkapi.ContextBuilder;
import cz.cuni.mff.d3s.been.core.task.TaskContextDescriptor;
import cz.cuni.mff.d3s.been.core.task.TaskContextState;

public class HelloWorldBenchmark extends Benchmark {
    @Override
    public TaskContextDescriptor generateTaskContext() throws BenchmarkException {
        ContextBuilder contextBuilder =
            ContextBuilder.createFromResource(HelloWorldBenchmark.class, "context.tcd.xml");
        TaskContextDescriptor taskContextDescriptor = contextBuilder.build();
        return taskContextDescriptor;
    }

    @Override
    public void onResubmit() { }

    @Override
    public void onTaskContextFinished(String s, TaskContextState taskContextState) { }
}
```

Notice the methods `onResubmit` and `onTaskContextFinished` which are used as notifications for the benchmark. You can use these methods for whatever error handling or logging you need.

You are supposed to implement the logic for generating the contexts. When your benchmark is done and it will not generate any more contexts, return `null` from the `generateTaskContext` method.

2.5.9 Creating Task Contexts

The preferred way of creating task contexts is to use the `ContextBuilder` class to load a XML file representing the context descriptor from a resource. This class also provides various methods for modifying the context descriptor and the contained tasks.

You can add tasks into the context via the `addTask` method, these tasks can be created using the `newEmptyTask` method. The context descriptor can also provide *task templates* which can be used to create tasks.

Preferably you should create the whole descriptor in the XML file and only use the `setProperty` method to set the parameters to the task contexts. When the descriptor is ready call the `build` method to generate object representation of the descriptor which can be returned to the framwork.

2.5.10 Resubmitting and Benchmark Storage

Benchmarks are supposed to be long-running and EverBEEN provides a mechanism to keep benchmarks running even after a failure occurs. When a generator task exits with an error (e.g. power outage), it will get resubmitted and the benchmark will continue. To support this behavior, you should use the provided benchmark key-value storage for the internal state of the benchmark and avoid using instance variables.

The `Benchmark` abstract class provides methods `storageGet` and `storageSet` which will use the cluster storage for the benchmark state. This storage will be restored whenever the generator task is resubmitted. The implementation of a benchmark that uses this storage can look like this:

```
@Override
public TaskContextDescriptor generateTaskContext() throws BenchmarkException {
    int currentRun = Integer.parseInt(this.storageGet("i", "0"));
    TaskContextDescriptor taskContextDescriptor;
    if (currentRun < 5) {
        // generate a regular context
        taskContextDescriptor = ...;
    } else {
        // we're done
        taskContextDescriptor = null;
    }

    currentRun++;
    this.storageSet("i", Integer.toString(currentRun));

    return taskContextDescriptor;
}
```

2.5.11 Evaluators

EverBEEN provides a special task type called **evaluator**. The purpose of such a task is to query the stored results, perform statistical analyses and return an interpretation of the data that can be shown back to the user via the Web Interface. Evaluators are again tasks and they can be run manually (as a single task) or within a benchmark or a context. It's up to the user when and how to run an evaluator.

To create an evaluator, subclass the abstract class `Evaluator`[19] and implement the method `evaluate`. This method is supposed to return an `EvaluatorResult`[20] object which will be stored through the persistence layer. The object holds a byte array of data and its MIME type. EverBEEN supports a few MIME types which can be displayed in the Web Interface, e.g. a JPEG image.

An evaluator needs to retrieve data from the persistence layer, and it can do so using the provided `ResultFacade` interface. This object is available as an instance method on the `Task` superclass. Queries can be build using the `QueryBuilder` object which supports various conditions and query parameters. A simple query that will retrieve a collection of results can have this form:

```
Query query = new QueryBuilder().on(...).with(...).fetch();
Collection<MyResult> data = results.query(query, MyResult.class);
```

For an example of a simple evaluator that output a plot chart with the measured data and error intervals, see the sample `nginx-benchmark`.

2.6 BPKs and Software repository

2.6.1 Been package (BPK)

Been package, shortly BPK, contains binaries and descriptors needed to run tasks, task contexts and benchmarks. Each package has its own unique identifier structured in a way resembling identifiers used by Maven. The identifier consists of three parts:

1. **Group ID** — A universally unique identifier for a BPK. It is a good practice to use a fully-qualified package name to distinguish it from other BPK packages with a similar name (eg. cz.been.example).
2. **Bpk ID** — The identifier of the BPK that is unique within the group given by the group ID.
3. **Version** — The current version of the BPK.

BPK package is represented by a single file with a ***.bpk** suffix. In fact the bpk file is a zip file with a predefined structure described below.

[19]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/Evaluator.html
[20]http://everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/evaluators/EvaluatorResult.html

```
package.bpk
   \___ config.xml
   \___ files/
   \___ lib/
   \___ tcds/
   \___ tds/
```

1. ***config.xml*** file — Main configuration of the BPK. It consists of two main sections:
   - **metaInf** section — specifies unique identifier of the BPK.
   - **runtime** section — specifies runtime type. Been supports two runtime types - **JavaRuntime** and **NativeRuntime**.
     - **JavaRuntime** — defines tasks written in JVM based language (e.g. Java, Scala, Groovy). The name of the jar with the implementation is required.
     - **NativeRuntime** — defines tasks written in other languages. This type requires the name of the executable to be used.

   Following examples show valid descriptors for *java* and *native* runtimes.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:bpkConfiguration xmlns:ns2="http://been.d3s.mff.cuni.cz/bpk/config">

    <ns2:metaInf>
        <groupId>fully-quallified.group.id</groupId>
        <bpkId>bpkId</bpkId>
        <version>3.0.0-EXAMPLE-ALPHA</version>
    </ns2:metaInf>

    <ns2:runtime xsi:type="ns2:JavaRuntime"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <jarFile>packed_jar_with_tasks.jar</jarFile>
    </ns2:runtime>

</ns2:bpkConfiguration>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:bpkConfiguration xmlns:ns2="http://been.d3s.mff.cuni.cz/bpk/config">

    <ns2:metaInf>
        <groupId>fully-quallified.group.id</groupId>
        <bpkId>bpkId</bpkId>
        <version>3.0.0-EXAMPLE-ALPHA</version>
    </ns2:metaInf>

    <ns2:runtime xsi:type="ns2:NativeRuntime"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <binary>name_of_executable</binary>
    </ns2:runtime>

</ns2:bpkConfiguration>
```

2. ***files/*** directory — contains executable and library files. In the case of the ***JavaRuntime*** it must contain an appropriate .jar file, ***NativeRuntime*** must place its executable and library files there.
3. ***lib/*** directory — ***JavaRuntime***s can place additional jars which will be added to the `classpath` of a task once running on a Host Runtime.
4. ***tcds/*** directory — contains Task Context descriptors.

5. ***tds/*** directory — must contain at least one Task descriptor.

When a task is started, *lib/*, *files/*, *tds/* and *tcds/* directories are copied into the working directory of the task.

Even though it is possible to create a BPK "by hand", this is not recommended. The standard way of assembling a BPK file is to use the Been Bpk Plugin for Maven which does all the hard work.

Note: You can see additional XML elements (not mentioned above) in *context.xml* file generated by Been Bpk Plugin for Maven and you can also find definitions of these elements in corresponding *.xsd file, but these elements are not used by the current version of EverBEEN.

2.6.2 Software repository

The main purpose of the software repository is to store BPKs for future use. The Software Repository is implemented as a service which can be started on an arbitrary EverBEEN node.

2.6.3 BPK versioning

Software repository does not allow re-uploading of BPKs with the same `groupId`, `bpkId` and `version`. If you want to re-upload a BPK, you have to change its version. The reason for this limitation is simple — it prevents inconsistencies and unpredictable behavior caused by version mismatches.

To facilitate the development of new tasks and benchmarks BPKs can be created with version suffixed by **'-SNAPSHOT'**. Such versions of BPKs can be re-uploaded to the Software Repository. Also Host Runtimes will download such versions instead of using cached packages. It is not recommended to use **'-SNAPSHOT'** in a production environment.

2.7 Persistence layer

EverBEEN persistence layer serves as a bridge between EverBEEN distributed memory and a database of choice, rather than a direct storage component. This enables EverBEEN to run without a persistence layer, at the cost of heap space and a risk of data loss in case of an unexpected cluster-wide shutdown. EverBEEN doesn't *need* a persistence layer per se at any given point in time. User tasks, however, might attempt to work with previously acquired results. Such attempts will result in task-scope failures if the persistence layer is not running. Log archives, too, will be made unavailable if the persistence layer is offline.

2.7.1 Characteristics

An overview of the main characteristics of EverBEEN's persistence layer follows.

2.7.1.1 Bridging

The EverBEEN persistence layer doesn't offer any means of storing objects. It only functions as an abstract access layer to an existing storage component (e.g. a database). EverBEEN comes with a default implementation of this bridge for the MongoDB database, but it is possible to port it to a different database (see extension point notes for more details). The user is responsible for setting up, running and maintaining the actual storage software.

2.7.1.2 Eventual persistence

As mentioned above, object-persisting commands (result stores, logging) do not, by themselves, execute insertions into the persistence layer. They submit objects into EverBEEN's distributed memory. When a persistence layer node is running, it continually drains this distributed memory, enacting the actual persistence of drained objects. This offers the advantage of being able to pursue persisting operations even in case the persistence layer is currently unavailable.

The downside of the bridging approach is that persisted objects might not find their way into the actual persistence layer immediately. It also means that should a cluster-wide shutdown occur while some objects are still in the shared memory, these objects will get lost. All that can be guaranteed is that submitted objects will eventually be persisted, provided that some data nodes and a persistence layer are running. Experience shows that the transport of objects through the cluster and to the persistence layer is a matter of fractions of a second.

2.7.1.3 Scalability

As mentioned above, EverBEEN does not strictly rely on the existence of a persistence node for running user code, only to present the required data to the user. That being said, EverBEEN can also run multiple persistence nodes. In such case, it is the user's responsibility to set up these nodes in a way that makes sense.

While running multiple nodes, please keep in mind that these storage components will be draining the shared data structures concurrently and independently. It is entirely possible to setup EverBEEN to run two persistence nodes on two completely separate databases, but it will probably not result in any sensibly expectable behavior, as potentially related data will be scattered randomly across two isolated database instances.

Generally speaking, having multiple persistence layer nodes is only useful if you:

- Have highly limited resources for each persistence node and wish to load-balance accesses to the same database.
- Have a synchronization/sharding strategy set up.

Additional use-cases may arise if you decide to write your own database adapter. In that case, consult the extension point for more detail.

2.7.1.4 Automatic cleanup

To prevent superfluous information from clogging the data storage, the persistence layer runs a Janitor component that performs database cleanup on a regular basis. The idea is to clean all old data for failed jobs and all metadata for successful jobs after a certain lifecycle period has passed. For lifecycle period and cleanup frequency adjustment, see the 2.8.1.5 (janitor configuration) section.

2.7.2 Components

A brief description of components that contribute to forming the EverBEEN persistence layer follows.

2.7.2.1 Object Repository

It goes without saying that EverBEEN needs some place to store all the data your tasks will produce. That's what the Object Repository does. Each time a task issues a command to submit a result, or logs a message, this information gets dispatched to the cluster, along with the associated object. The Object Repository provides a functional endpoint for this information. It effectively concentrates distributed data to its intended destination (a database, most likely). In addition, the Object Repository is also in charge of dispatching requested user data back.

2.7.2.2 Storage

The Storage component supplies a database connector implementation. All communication between the Object Repository and the database is done through the Storage API.

The Storage component gets loaded dynamically by the Object Repository at startup. If you want to use a different database than MongoDB, this is the component you'll be replacing (potentially along with the MapStore component).

2.7.2.3 MapStore

Where the ObjectRepository stores user data, the MapStore is used to map EverBEEN cluster memory to a persistent storage, which enables EverBEEN to preserve job state memory through cluster-wide restarts. The MapStore runs on all DATA nodes.

2.8 EverBEEN configuration

Configuration of the framework is done through a single, standard property file. The configuration is propagated to all services, each service uses a subset of the options.

A user property file is supplied to EverBEEN by the `-cf [file|URL]` (or `--config-file`) command line option. The value can be either a file or a URL pointing to the file. Using configuration by specifying a URL simplifies deployment in large environments, by reducing the need to distribute the file among the machines on which the framework runs.

To check the values in effect use `-dc` (or `--dump-config`) command line option (possibly along with the `-cf` option). It prints the configuration which will be used — the output provides a basic configuration file (options with default value are commented out with `#`).

Default configuration values are supplied, before you change any of them, consult documentation and make sure you understand the implications.

2.8.1 Configuration options

A detailed description of available configuration options of the EverBEEN framework follows. The default value for each configuration option is provided.

2.8.1.1 Cluster Configuration

Cluster configuration describes how nodes will form a cluster and how the cluster will behave. The configuration is directly mapped to Hazelcast configuration. These options are applicable only to DATA nodes.

It is *essential* that all cluster nodes use the same configuration for these options, otherwise they may not form a cluster.

`been.cluster.group`=*dev*
Group to which the nodes belong. Nodes whose group settings differ will not form a cluster

`been.cluster.password`=*dev-pass*
Password for the group. Nodes whose group password settings differ will not form a cluster

`been.cluster.join`=*multicast*
Manages how nodes form the cluster. Two values are possible: `multicast` which implies only `been.cluster.multicast.*` options will be used, and `tcp` which implies only `been.cluster.tcp.members` option will be used.

`been.cluster.multicast.group`=*224.2.2.3*
Specifies the multicast group to use.

`been.cluster.multicast.port`=*54327*
Specifies the multicast port to use.

`been.cluster.tcp.members`=*localhost:5701*
Semicolon separated list of `[ip|host][:port]` nodes to connect to.

`been.cluster.port`=*5701*
The port on which the node will listen.

`been.cluster.interfaces`=
Semicolon separated list of interfaces Hazelcast should bind to, the '*' wildcard can be used, e.g. `10.0.1.*`.

`been.cluster.preferIPv4Stack`=*true*
Whether to prefer the IPv4 stack over IPv6.

`been.cluster.backup.count`=*1*
The number of backups the cluster should keep.

`been.cluster.logging`=*false*
Enables/Disables logging of Hazelcast messages. Note that Hazalcast log messages are not persisted as other service logs.

`been.cluster.mapstore.use`=*true*
Wheather to use MapStore to persist cluster runtime information.

`been.cluster.mapstore.write.delay`=*0*
Delay in seconds with which to write to the MapStore. *0* means *write-through*, values bigger than zero mean *write-back*. Certain Map Store implementations will be more efficient in write-back mode.

`been.cluster.mapstore.factory`=*cz.cuni.mff.d3s.been.mapstore.mongodb.MongoMapStoreFactory*
Implementation of MapStore, must be on the classpath when starting a node.

`been.cluster.socket.bind.any`=*true*
Whether to bind to local interfaces.

2.8.1.2 Cluster Client Configuration

Cluster client configuration options are used when a node is connected to the cluster in *NATIVE* client mode. Cluster Configuration options are ignored in that case.

`been.cluster.client.members`=*localhost:5701*
Semicolon separated list of '[ip|host][:port] cluster members to connect to. At least one member must be available.

`been.cluster.client.timeout`=*120*
Inactivity timeout in seconds. The client will disconnect after the timeout.

2.8.1.3 Task Manager Configuration

Task Manager configuration options are used to tune the Task Manager. Use with care!

`been.tm.benchmark.resubmit.maximum-allowed`=*10*
Maximum number of resubmits of a failed benchmark task the Task Manager will allow.

`been.tm.scanner.period`=*30*
Period in second of the Task Manager's local key scanner.

`been.tm.scanner.delay`=*15*
Initial delay in seconds of the Task Manager's local key scanner.

2.8.1.4 Cluster Persistence Configuration

Configuration for the persistence transport layer. See chapter 2.7 (Persistence) for more details.

`been.cluster.persistence.query-timeout`=***10***
The timeout for queries into the persistence layer.

`been.cluster.persistence.query-processing-timeout`=***5***
The timeout for a query's processing time in the persistence layer. Processing time includes the trip the data has to make back to the requesting host.

2.8.1.5 Persistence Janitor Configuration

Configuration for the persistence layer janitor component. See 2.7 (Persistence) for more details.

`been.objectrepository.janitor.finished-longevity`=***168***
Number of hours objects with a `FINISHED` status stay persistent.

`been.objectrepository.janitor.failed-longevity`=***96***
Number of hours objects with a `FAILED` status stay persistent.

`been.objectrepository.janitor.service-log-longevity`=***168***
Number of hours EverBEEN service logs stay persistent.

`been.objectrepository.janitor.load-sample-longevity`=***168***
Number of hours EverBEEN node load monitor samples stay persistent. If set to `0`, load sample cleanup will be disabled.

`been.objectrepository.janitor.cleanup-interval`=***10***
Period in minutes of janitor cleanup checks.

2.8.1.6 Monitoring Configuration

Host Runtime monitoring configuration options.

`been.monitoring.interval`=***5000***
Interval of Host Runtime system monitoring samples, in milliseconds.

2.8.1.7 Host Runtime Configuration

Host Runtime configuration options.

`hostruntime.tasks.max`=***15***
Maximum number of tasks per Host Runtime.

`hostruntime.tasks.memory.threshold`=***90***
Host Runtime memory threshold in percent. If the threshold is reached no other task will be run on the Host Runtime. The value must be between 20% - 100&. The threshold is compared to the value of '(free memory/available memory)*100'.

`hostruntime.wrkdir.name`=***.HostRuntime***
Relative path to the Host Runtime working directory.

`hostruntime.tasks.wrkdir.maxHistory`=***4***
Maximum number of task working directories a Host Runtime will keep. When this number is exceeded at the boot of a Host Runtime service, the oldest existing directory is deleted.

2.8.1.8 MapStore Configuration

MapStore configuration options.

`been.cluster.mapstore.db.hostname`=***localhost***
Host name (full connection string including port). If no port is specified, default port is used.

`been.cluster.mapstore.db.dbname`=***BEEN_MAPSTORE***
Name of the database instance to use.

`been.cluster.mapstore.db.username`=***null***
User name to use to connect to the database.

`been.cluster.mapstore.db.password`=***null***
Password to use to connect to the database.

2.8.1.9 Mongo Storage Configuration

Configuration options for the MongoDB based Object Storage.

`mongodb.hostname`=***localhost***
Host name (full connection string including port). If no port is specified, default port is used.

`mongodb.dbname`=***BEEN***
Name of the database instance to use.

`mongodb.username`=***null***
User name to use to connect to the database.

`mongodb.password`=***null***
Password to use to connect to the database.

2.8.1.10 Software Repository Configuration

`swrepository.port`=***8000***
Port on which the Software Repository should listen for requests.

2.8.1.11 File System Based Store Configuration

`hostruntime.swcache.folder`=***.swcache***
Caching directory for downloaded software on Host Runtimes, relative to the working directory of a node.

`swrepository.persistence.folder`=***.swrepository***
Default storage directory for Software Repository server, relative to the working directory of a node.

2.9 EverBEEN best practices

To avoid potential problems please keep in mind the following recommendations:

- Read the documentation.
- Check network and firewall settings.
- Do not run EverBEEN instances with shared working directory.
- Use provided tools (such as the `bpk-maven-plugin`).
- Start with fewer `DATA` nodes, use `NATIVE` nodes to run the Host Runtime service.
- MongoDB instance is needed to properly use the framework.
- If a host has more network interfaces, configure the one(s) you want to use. See 2.8.1.1 (Cluster Configuration).
- If writing a task or benchmark see provided examples.

2.10 EverBEEN extension points

As mentioned above, EverBEEN comes with a default persistence solution for MongoDB. We realize, however, that this might not be the ideal use-case for everyone. Therefore, the MongoDB persistence layer is fully replaceable if you provide your own database connector implementation.

There are two persistence components you might want to override - the Storage and the MapStore.

If your goal is to relocate EverBEEN user data (benchmark results, logs etc.) to your own database and don't mind running a MongoDB as well for EverBEEN service data, you will be fine just overriding the Storage. If you want to port the entire EverBEEN's persistence layer, you will have to reimplement `MapStore` as well.

2.10.1 Storage extension

As declared above, the *Storage* component is fully replaceable by an implementation different from the default MongoDB adapter. However, we would like to avoid letting you plunge into this extension point override without the necessary guidelines and warnings.

2.10.1.1 Override warning

The issue with *Storage* implementation is that the persistence layer is designed to be completely devoid of any type knowledge. The reason for this is that *Storage* is used to persist and retrieve objects from user tasks. Should the *Storage* have any RTTI knowledge of the objects it works with, imagine what problems could arise when two tasks using two different versions of the same objects attempted to use the same *Storage*.

To avoid this, the *Storage* only receives the object JSON and some information about the object's placement. This being said, the *Storage* still needs to perform efficient querying based on some attributes of the objects it stores.

This is generally not an issue with NoSQL databases or document-oriented stores, but it can be quite hard if you use a traditional ORM. The ORM approach additionally presents the aforementioned class version problem, which you would need to solve. If you choose ORM be prepared to run into the following:

- **EverBEEN classes** - You will probably need to map some of these in your ORM.
- **User types** - You will likely need to share a user-type library with your co-developers to aggree on permitted result objects.
- **User type versions** - Should the version of this user-type library change, you will need to restart the *Storage* before running any new tasks on EverBEEN. Restarting EverBEEN will likely result in malfunction of tasks using an older version of the user-type library.

2.10.1.2 Override implementation overview

It is highly recommended that you use Apache Maven[21] to build your implementation. Extension without Maven is possible, but will not be covered in this booklet. Additionally, you will need git[22] to check out EverBEEN sources.

Once both Maven and git are ready, you will need to check out the EverBEEN project and install the artifacts to your local repository:

```
git clone git@github.com:ever-been/everBeen.git
mvn install
```

You will need to import two EverBEEN modules to provide a *Storage* implementation, as follows:

[21] http://maven.apache.org/
[22] http://git-scm.com/

```
<dependency>
        <groupId>cz.cuni.mff.d3s.been</groupId>
        <artifactId>core-data</artifactId>
        <version>${been.version}</version>
</dependency>

<dependency>
        <groupId>cz.cuni.mff.d3s.been</groupId>
        <artifactId>storage</artifactId>
        <version>${been.version}</version>
</dependency>
```

Then create a **been.version** property in your Maven module that corresponds to the EverBEEN version you checked out and installed.

Now that you have your project set up, you can start working on the implementation. To replace the *Storage* implementation, you will need to implement the following:

- Storage[23] — the main interface providing the actual store management
- StorageBuilder[24] — an instantiation/configuration tool for your *Storage* implementation
- SuccessAction<EntityCarrier>[25] — an isolated action capable of persisting objects

Additionally, you will need to create a **META-INF/services** folder in the jar with your implementation, and place a file named **cz.cuni.mff.d3s.been.storage.StorageBuilder** in it. You will need to put a single line in that file, containing the full class name of your `StorageBuilder` implementation.

We also strongly recommend that you implement these as well:

- QueryRedactorFactory[26] (along with QueryRedactor[27] implementations)
- QueryExecutorFactory[28] (along with QueryExecutor[29] implementations)

The general idea is to implement the *Storage* component and to provide the *StorageBuilder* service, which configures and instantiates your *Storage* implementation. The **META-INF/services** entry is for the *ServiceLoader* EverBEEN uses to recognize your *StorageBuilder* implementation on the classpath. EverBEEN will then pass the *Properties* from the *been.conf* file (see 2.8 (configuration)) to your *StorageBuilder*. That way, you can use the common property file to configure your *Storage*.

The `Storage` interface is the main gateway between the Object Repository and the database. When overriding the Storage, there will be two major use-cases you will have to implement: the asynchronous persist and the synchronous query.

2.10.1.3 Asynchronous persist

All *persist* requests in EverBEEN are funneled through the `Storage#store` method. You will receive two parameters in this method:

entityId The *entityId* is meant to determine the location of the stored entity. For example, if you're writing an SQL adapter, it should determine the table where the entity will be stored. For more information on the *entityId*, see section 2.10.1.6 persistent object info.

JSON A serialized JSON representation of the object to be stored.

[23]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/Storage.html
[24]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/StorageBuilder.html
[25]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/SuccessAction.html
[26]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryRedactorFactory.html
[27]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/QueryRedactor.html
[28]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryExecutorFactory.html
[29]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryExecutor.html

Generally, you will need to decide where to put the object based on its *entityId* and then somehow map and store it using its *JSON*.

The `Storage#store` method is asynchronous. It doesn't return any outcome information, but always throws a *DAOException* when the persist attempt fails. This informs the *ObjectRepository* that the operation failed and an action to prevent data loss must be taken.

2.10.1.4 Query / Answer

The other type of requests supported by *Storage* are queries. They are synchronous and a *Query* is always answered with a *QueryAnswer*. In order to support queries, you could implement all the querying mechanics by yourself (if you wish to do that, see the 2.5.6 (Task API) for more details), but this is unnecessary. The `QueryTranslator`[30] adapter is designed to help you interpret queries without having to iterate through the entire query structure.

The preferred way of interpreting queries is to create a `QueryRedactor`[31] implementation (or several, in fact). The *QueryRedactor* class is designed to help you construct database-specific query interpretations using callbacks. This way, you instantiate the *QueryTranslator*, call its *interpret* method passing in your instance of the *QueryRedactor* and the *QueryTranslator* calls the appropriate methods on your *QueryRedactor*. Once configured, your *QueryRedactor* can be used to assemble and perform the expected query. There are additional interfaces that can help you in the process (`QueryRedactorFactory`[32], `QueryExecutor`[33] and `QueryExecutorFactory`[34]).

Once you execute the query, you will need to synthesize a `QueryAnswer`[35], which you can do using `QueryAnswerFactory`[36]. If there is data associated with the result of the query, you need to create a *data answer* using `QueryAnswerFactory#fetched(...)`. The other *QueryAnswerFactory* methods are used to indicate the query status. See the method in-code comments for more details about available answer types.

2.10.1.5 Auxiliary methods

In addition to persisting and querying, the *Storage* interface features auxiliary methods you will need to implement.

- **createPersistAction** — returns an instance of your implementation of `SuccessAction\<EntityCarrier\>`[37]; its *perform* method is presumed to call your `Storage#store()` implementation.
- **isConnected** — a situation may occur when the *Object Repository* is running, but the database it uses is not; this simple method is designed to help EverBEEN detect such a situation by returning `false` should the database connection drop.
- **isIdle** — a database usage heuristics function that helps the *Object Repository* janitor detect cleanup windows (to interfere less with user data processing).

2.10.1.6 General persistent object info

Although the *Storage* doesn't implicitly know any RTTI about the object it's working with, there are some safe assumptions you can make based on the *entityId* that comes with the object.

The *entityId* is composed of *kind* and *group*. The *kind* is supposed to represent what the persisted object actually is (e.g. a log message). The following kinds are currently recognized by EverBEEN:

- **log** - log messages and host load monitoring

[30]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/QueryTranslator.html
[31]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/QueryRedactor.html
[32]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryRedactorFactory.html
[33]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryExecutor.html
[34]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/QueryExecutorFactory.html
[35]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/QueryAnswer.html
[36]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/QueryAnswerFactory.html
[37]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/SuccessAction.html

- **result** - stored task results
- **descriptor** - *task/context* configuration; used to store run-time parameters of a *task* or *context*
- **named-descriptor** - *task/context* configuration; user-stored configuration templates for *task* or *context* runs
- **evaluation** - output of evaluations performed on task results; these objects contain serialized BLOBs - see Chapter 2.5.11 (Evaluators) for more detail
- **outcome** - meta-information about the state and outcome of jobs in EverBEEN; these are used in automatic cleanup

The *group* is supposed to provide a more granular grouping of objects and depends entirely on the object's *kind.*

If you need more detail on objects that you can encounter, be sure to also read the next section, which denotes where various EverBEEN classes can be expected and what *entityIds* can carry user types.

2.10.1.7 The ORM special

If you are really hell-bent on creating an ORM implementation of the *Storage*, your module will need to know several extra EverBEEN classes to be able to perform the mapping. The following table covers their *entityIds*, their meaning and the dependencies you will need to get them.

kind	*group*	meaning	class	module
log	task	message logged by a task	`TaskLogMessage`[38]	core-data
log	service	message logged by a service	`ServiceLogMessage`[39]	core-data
log	monitoring	host monitoring sample	`MonitorSample`[40]	core-data
descriptor	task	task runtime configuration	`TaskDescriptor`[41]	core-data
descriptor	context	task context runtime configuration	`TaskContextDescriptor`[42]	core-data
named-descriptor	task	saved task configuration	`TaskDescriptor`	core-data
named-descriptor	context	saved task context configuration	`TaskContextDescriptor`	core-data
result	*	task result	user class extending Result[43]	*n/a* (results)
evaluation	*	task result evaluation	`EvaluatorResult`[44]	results
outcome	task	task state service records	`PersistentTaskState`[45]	persistence
outcome	context	task context state service records	PersistentContextState[46]	persistence

Thus, if you need to infer the knowledge of the runtime type of all these classes to your module, you need to add the following to Maven dependencies:

```
<dependency>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>persistence</artifactId>
    <version>${been.version}</version>
</dependency>
```

[38]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/logging/TaskLogMessage.html
[39]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/logging/ServiceLogMessage.html
[40]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/ri/MonitorSample.html
[41]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskDescriptor.html
[42]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskContextDescriptor.html
[43]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/results/Result.html
[44]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/evaluators/EvaluatorResult.html
[45]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/task/PersistentTaskState.html
[46]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/task/PersistentContextState.html

```
<dependency>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>results</artifactId>
    <version>${been.version}</version>
</dependency>
```

Additionally, you will probably need to inject a dependency containing your pre-defined result types (*Result* extenders used by your benchmarks). As mentioned before, you will need to be very careful about the versioning of this module.

2.10.1.8 Replacing the Storage implementation

After you implement your own *Storage* back-end, you need to sew it back into EverBEEN. EverBEEN is bundled using the Maven Assembly Plugin[47], which unpacks EverBEEN modules along with their dependencies, combines their class files and creates the ultimate jar. That means that to actually swap the *Storage* implementation, you will need to rebuild EverBEEN with some modifications.

First, build your *Storage* module using `mvn install`. That will deploy your artifact to the local Maven repository, where EverBEEN can see it. For further reference, let's assume your storage artifact identifier is `my.group:my-storage:2.3.4`.

Then, you will need to rebuild EverBEEN using your *Storage* module instead of the default one. For that, you will need a deployment project. This project will use `pom` packaging and will only contain the `pom.xml` with instructions for Maven Assembly Plugin. Because writing the assembly descriptor is tedious, we have created the `pom` for you as a quick starter:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>my.group</groupId>
    <artifactId>my-been-flavor</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>

    <properties>
        <been.version>3.0.0</been.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>cz.cuni.mff.d3s.been</groupId>
            <artifactId>node</artifactId>
            <version>${been.version}</version>

            <exclusions>
                <exclusion>
                    <groupId>cz.cuni.mff.d3s.been</groupId>
                    <artifactId>mongo-storage</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
```

[47]http://maven.apache.org/plugins/maven-assembly-plugin/

```
                <groupId>my.group</groupId>
                <artifactId>my-storage</artifactId>
                <version>2.3.4</version>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-assembly-plugin</artifactId>
                    <version>2.4</version>

                    <configuration>
                        <finalName>myBeenFlavor</finalName>
                        <appendAssemblyId>false</appendAssemblyId>
                        <archive>
                            <manifest>
                                <mainClass>cz.cuni.mff.d3s.been.node.Runner</mainClass>
                            </manifest>
                        </archive>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                    </configuration>
                </plugin>
            </plugins>
        </build>
    </project>
```

Just to explain what's going on above:

- your deployment project has the `cz.cuni.mff.d3s.been:node` artifact as its dependency; this is the artifact into which we funnel all the runnable EverBEEN modules, so you will have the entire EverBEEN portfolio into your assembly just by linking that module
- however, in that dependency, you specify that the `cz.cuni.mff.d3s.been:mongo-storage` artifact should be excluded; that is the artifact containing the default MongoDB implementation of *Storage*
- then, your deployment project links the `my.group:my-storage:2.3.4` which you installed earlier in your maven repository; that means **your** *Storage* implementation will be placed in the assembly
- finally, there's the assembly plugin configuration, saying that a `jar` file named `myBeenFlavor.jar` should be deployed into the `target` folder of your deployment project, assembling classes from all dependencies, with `cz.cuni.mff.d3s.been.node.Runner` for main class

Finally, you will need to create your assembly, which can be done by invoking `mvn assembly:assembly` in the root of your deployment project

This will result in a runnable EverBEEN node `jar`, with `cz.cuni.mff.d3s.been:mongo-storage` excluded, with `my.group:my-storage:2.3.4` included on the classpath, which will cause *Object Repository* to see **your** implementation instead of the default one.

2.10.2 MapStore extension

EverBEEN uses the *MapStore* to maintain persistent knowledge about the state of your tasks (and other jobs). You only need to override the default MongoDB implementation if you need to get rid of MongoDB completely.

The EverBEEN *MapStore* is a direct bridge between Hazelcast (the technology EverBEEN uses for clustering) and a persistence layer, so overriding it is pretty straightforward. You need to do the following:

- **Implement the Hazelcast MapStore**[48] **interface** — see above for links.
- **Implement the Hazelcast MapStoreFactory**[49] **interface** — again, see above for the link. Do not get confused by the fact that *MapStoreFactory* returns a *MapLoader* instance. The *MapStore* extends the *MapLoader* with storing methods, which you will need, so you need to to return an instance of your *MapStore* implementation in `YourMapStoreFactory#newMapStore()`.
- **Configure EverBEEN to use your MapStore** — in the `been.conf` (or any other EverBEEN config file you are using), you need to set the `been.cluster.mapstore.factory` property to the **fully qualified class name** of your **MapStoreFactory** implementation.
- **Get your package on the EverBEEN classpath** — Make sure to use the same *MapStore* implementation on all EverBEEN cluster nodes otherwise you might end up with your job status data being partitioned across multiple completely different databases.

[48]http://www.hazelcast.com/docs/2.5/javadoc/com/hazelcast/core/MapStore.html
[49]http://www.hazelcast.com/docs/2.5/javadoc/com/hazelcast/core/MapStoreFactory.html

Chapter 3

EverBEEN developer documentation

3.1 Design goals

The original goal of the EverBEEN project (as stated in the former assignment[1]) was mainly to cleanup the existing WillBEEN[2] project and replace the RMI[3] framework by a more robust networking solution.

However, feedback from previous attempts of deployment in the corporate sector showed that framework stability was not the only issue. The tools for easy creation of WillBEEN jobs were lacking at best, and we experienced the reported difficulties in WillBEEN deployment first-hand. Furthermore, experience showed that some advanced features of WillBEEN (namely the Results Repository) had poor real-case use. Jiří Täuber's master thesis[4], aimed at analyzing real case WillBEEN deployment, clearly marks these issues as a major factor of WillBEEN's failure in a production environment.

These findings made us focus not only on a complete reimplementation of WillBEEN, but also on the user perspective of EverBEEN deployment and regression benchmarking in general. As a result, we set up several goals which we tried to stand up to during EverBEEN design and development.

3.1.1 Scalability, Redundancy, Reliability

As we were deciding which networking technology EverBEEN will use, we were driven to make EverBEEN as robust as possible in face of network failures and OS freezes. The choice of Hazelcast as a networking technology took this idea to new heights, enabling us to build EverBEEN as a truly distributed system, rather than just a network of interconnected nodes.

As a result, we decentralized all the decision-making in EverBEEN. Decisions are made on the basis of distributed shared memory and as long as multiple data nodes are running, there is no single point of failure. The failure of a single partaking host was seen as an eventuality, rather than an unrecoverable error, and was counted with from the start of EverBEEN development, as was the case of a temporary disconnection of the persistence layer.

3.1.2 Modularity

Modularity was the first code characteristic we noted as lacking in WillBEEN. Although some pseudo-modules were present, the entire codebase was compiled together, leading to frequent cross-references in the project. Not only does this pose an issue with code maintainability, but it also makes component overrides very demanding in terms of the user's knowledge of the entire system. With the aid of modern building tools (mostly Apache Maven), we made EverBEEN a modular project where component overriding is possible.

[1]http://ksvi.mff.cuni.cz/~holan/SWP/zadani/ebeen.txt
[2]http://been.ow2.org/
[3]http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html
[4]https://is.cuni.cz/webapps/zzp/detail/78663/4417375

In reaction to previous problems with WillBEEN's result storage, we specifically interfaced the *Object Repository* (formerly *Results Repository*) database connector out of the project, thus making it easily replaceable if need be (see 2.10 extension points for a guide).

3.1.3 Ease of use

When we started attempts at refactoring the WillBEEN code, we were told that it took tens to hundreds hours to deploy WillBEEN and get some basic benchmarks working. We saw this quantity of time as unacceptable, hence the major focus of our work was on making the EverBEEN project easier to use.

First, we decided to completely invert the way EverBEEN services are programmed. WillBEEN services were tailored to work with each other, compiled together, but launched separately. In EverBEEN, we developed services separately, and only fused them together in the final assembly step. Thus, the communication between services only happens on the basis of a small common codebase containing relevant protocol objects. As opposed to WillBEEN, the order in which EverBEEN services are launched is not critical to the correct function of the cluster. We believe this considerably simplifies EverBEEN deployment, and reduces the study time necessary to make a benchmark run.

Second, we decided to simplify the process of task creation as much as possible. The decisions we had to make to see this goal through were particularly difficult, as simplification directly opposes the generality the rest of the framework had to offer. We came to a similar conclusion as WillBEEN authors did, and picked one technology we fully support — Java in combination with Apache Maven. As arbitrary as this decision may seem, it comes with huge benefits — the user can have a simple EverBEEN task up and running within minutes.

3.2 Decision timeline

June 2012
We took over the codebase of WillBEEN and started working on a new incarnation of the project, called EverBEEN.

July 2012
We decided to use Apache Maven as the build system instead of Apache Ant and to split the project into several modules.

August 2012
Attempts to "mavenize" the project discovered a lot of mutual dependencies among apparently independent parts of the codebase.

We started to consider various communication middleware frameworks as a replacement for RMI.

September 2012
`SLF4J` is chosen as a logging framework and `Logback` as the basic logging backend. We decided to unify all existing logging mechanisms.

October 2012
We started to implement a basic re-implementation of the project using the Hazelcast middleware, which was chosen as the best alternative from various other candidates, such as JMS and JGroups. Hazelcast offers a great combination of both scalability and decentralization which matched the project's goals best.

November 2012
Attempts to refactor the existing RMI-based code incrementally and switch to Hazelcast were catastrophic and we decided to actually rewrite the project from scratch instead.

Current use of One-Jar plugin was dropped in favor of Maven Assembly plugin.

We chose Sigar to be used as the implementation of hardware detectors.

December 2012
We acknowledged that it is impossible to create a high-level API independent on the low-level transport and communication protocol. We decided to make Hazelcast an integral dependency of EverBEEN.

February 2013
We decided to implement the Software Repository component as a HTTP server with a RESTful API. This allows us to reuse existing libraries for HTTP communication and achieve correct streaming of large file transfers.

March 2013
For the purposes of both inter- and intra-process communication, we chose 0MQ.

We decided to use JSON as a transport format for various inter-component communication, with Jackson as a serialization library. The (de-)serialization is easier and more flexible than XML.

April 2013
MongoDB is to be used as the default storage engine. Nevertheless, the persistence layer is to be implemented with a universal interface that would allow any other common database storage to be used instead. Also, MongoDB has a lot of features[5] that fit the EverBEEN use case.

The web interface will be written in Java using the Tapestry5 web framework. This will allow us to reuse existing data structures and classes and will take less time to write than pure JSP.

May 2013
The API for user-written benchmarks is settled to be a special form of task that will be called by EverBEEN and will generate task contexts on demand.

June 2013
We agreed to open-source the project on GitHub under the LGPL license.

July 2013
The API for evaluators and presentation of results in the web interface is settled.

August 2013
We chose the Markdown language and Pandoc[6] for writing the project documentation.

3.3 EverBEEN architecture

Unlike its predecessor, WillBEEN[7], EverBEEN was designed as a fully distributed application from the start. Despite the differences in the design process and the overall system architecture, we tried to stick to the time-proven concept of the original BEEN[8] as much as possible. The EverBEEN architecture is best explained on **figure 3.1**.

3.3.1 Cluster

Key characteristic of EverBEEN is its clustered (distributed) nature. EverBEEN is designed to be run on an open network of interconnected nodes (EverBEEN JVM processes, presumably on different computers). These nodes serve as a platform for launching user code or EverBEEN services.

3.3.2 Services

Probably the most notable fact in the above schema is the presence of clustered services, namely:

- **Software Repository** — Handles user code package distribution.
- **Host Runtime** — Runs user code.
- **Object Repository** — Stores user code outputs.
- **User Interface** — Generates cluster control-flow, display cluster state.

[5]http://www.youtube.com/watch?v=b2F-DItXtZs
[6]http://johnmacfarlane.net/pandoc/
[7]http://been.ow2.org/
[8]http://d3s.mff.cuni.cz/publications/download/Submitted_1404_BEEN.pdf

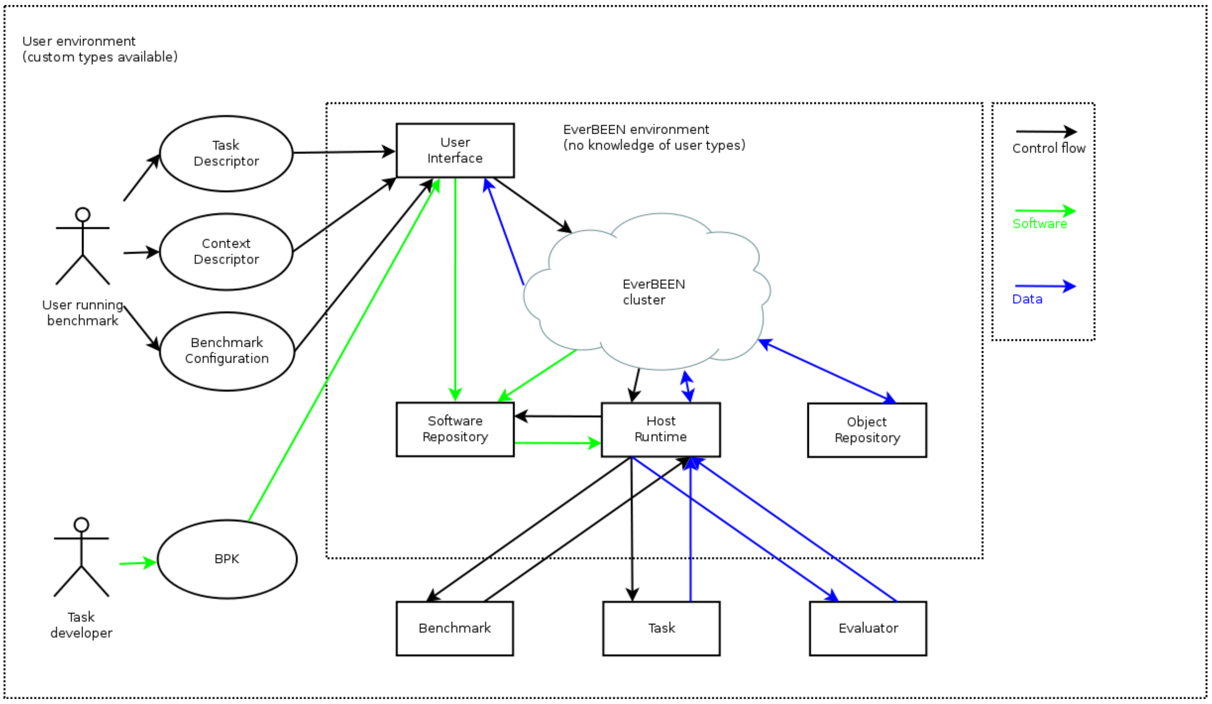


Figure 3.1: EverBEEN architecture

These services are run on EverBEEN cluster nodes, by configuring the node at launch time. While EverBEEN relies on the eventual availability of its services, it remains oblivious to their actual location, as long as they're reachable within the cluster. The only exception to this is the *Software Repository*, which emits its location to the cluster to provide software packages via a simple HTTP protocol.

However, the overview of EverBEEN services would not be complete without *Task Manager*, not seen on this diagram. The *Task Manager* is a component responsible for all the house-keeping around scheduling user code execution. As such, it plays an essential role in the EverBEEN coordination. This led us to make its decision-making process decentralized and ensure that multiple *Task Manager* instances could co-exist in the cluster. The *Task Manager* is run on every DATA node, which represents a transparent fail-over strategy in case one of the multiple data nodes has to terminate.

3.3.3 Native Nodes, Data Nodes

As mentioned above, EverBEEN is based around the idea of cluster nodes. Because it may be in the best interest to limit unnecessary load on nodes running EverBEEN services, we enabled EverBEEN nodes to run in two modes:

Data Nodes
Nodes running in this mode fully participate in cluster-wide data sharing. All data nodes run a *Task Manager* instance. These nodes add extra redundancy, but need to perform additional house-keeping, which increases the load they generate.

Native Nodes
Low-profile nodes that run without a *Task Manager* instance. Nodes running in this mode have access to all shared data, but are not responsible for any shared objects. They bring no additional redundancy, but generate less load and are more suitable for running EverBEEN services.

3.3.4 User code

Another factor that needs to be taken into consideration is the execution of user code in cooperation with the system. For security reasons, user code is always launched in a separate process in EverBEEN. As opposed to a thread-based solution, this approach offers better memory management and error handling. Moreover, it alleviates the restriction of user logic to JVM code.

3.3.5 User code zone

The clear separation of user and framework code zones is one of the major features introduced with EverBEEN. The motivation for this division is the absence of RTTI in system processes. WillBEEN's handling of user types involved these:

- Forcing the user to describe the data he persists using a Java-based meta-language.
- Class bytecode generation based on meta-language description.
- ORM mapping of so generaged classes using the Hibernate[9] framework.

This approach to persistence leads to several problems:

- To enact the ORM mapping, the generated class must be loaded. Once that happens, it cannot be unloaded using conventional means.
- Having multiple versions of meta-language description for the same ORM binding leads to conflicts (both classpath and SQL table).
- The user is forced to duplicate the definition of his data structures, which gives more room for errors.

In order to avoid this kind of hassle, we strove to rid the EverBEEN framework of all knowledge of user types, which ultimately leads to the code zone division discussed above.

3.3.6 User Interface

The EverBEEN cluster is controlled through a web interface, deployable to standard Java Servlet containers. To communicate with EverBEEN, it connects to the cluster as a native node, and issues commands through a facade called `BeenApi`. In that sense, the web interface component is both a client (cluster scope) and a server (user scope). Any number of web interface instances can run on an EverBEEN cluster.

3.4 Principal features

There are several EverBEEN features we are particularly proud of, mostly because we believe them to be a good match to the project goals assigned to us at the beginning of the project, or the design goals we set up ourselves when considering the deficiencies of previous project incarnations.

Scalability
Adding nodes to the EverBEEN cluster transparently increases the scale of benchmarks you can perform. There is no master node to bottleneck the decision-making even if you create a large cluster. The assumed (although untested) advantage of using `MongoDB` is its sharding ability, which should provide a database back-end scaling strategy implicitly compatible with EverBEEN.

Easy deployment
Deploying EverBEEN can be as simple as installing a database and running a few executable `jars` with a few command-line options. No shady deployment scripts. No installation. Just pure Java, with a database adapter and a synoptic front-end webapp to go with. Configuration is concentrated into one file, which you can load from a *URL* to quicken mass reconfiguration.

[9]http://www.hibernate.org/

Easy measures
If you use Java, creating a simple EverBEEN *task* is a matter of minutes, rather than hours. All you need is a `pom` file (for Apache Maven), one method override and a *task descriptor*. Once you have that, the `bpk-maven-plugin` will bundle your *BPK* with a simple `mvn package`. Once your *BPK* gets more complex, you can create *task descriptor* and *context descriptor* templates, which you can then tweak from the web interface before you run them. Once you get familiar with *tasks* and *task contexts*, creating a *benchmark* is easy, because you just need to write a *task* that creates *task contexts*, except that you can comfortably modify the *context* XML using Java-to-XML bindings.

User type transparency
For Java tasks, support for user result types is reduced to extending one class. Once you do that, your result objects are serialized, stored, queried and de-serialized without you needing to do any extra work - the *Task API* does it for you. If you happen to update your *task* code and change a result class's version (add a field, for example), you won't get into trouble if you apply a minimum of caution.

Extensibility
EverBEEN is modular and therefor extensible. If you don't like `MongoDB`, you can port EverBEEN completely to a different database by implementing two modules and creating one descriptor. Substituting the default `LOGBack` implementation for another `slf4j` implementation is fairly easy, too. Other possible extensions are in store for the future.

Maintainable code
Using modular design, modern technologies and flexible programming techniques, we managed to shrink the core EverBEEN codebase to under 70,000 lines of code while preserving most of the original project's functionality. Compared to over 160,000 lines of WillBEEN code, we have created an easily maintainable piece of software without sacrificing important features.

3.5 EverBEEN services

EverBEEN services are functional bundles run on cluster nodes in addition to the common core bundle. They are configured 'per-node' at boot time and define the node's role in the cluster.

3.5.1 Host Runtime

The Host Runtime is the service responsible for managing running tasks. It also functions as a gateway between its tasks and the rest of the framework.

The service was completely rewritten since the code quality was poor. The rewrite enabled the EverBEEN team to do necessary refactoring as well as to introduce libraries, such as Apache Commons Exec producing more modular and maintainable code.

Even though the service was completely rewritten, its purpose and basic functions remain similar to previous BEEN versions.

A Host Runtime can run on any type of EverBEEN node. It makes sense to run it on a *NATIVE* node in order to avoid costs associated with running a *DATA* node. Typically, deployment will have a few DATA nodes and as many NATIVE nodes with Host Runtime instances as needed.

Available configuration options are listed in the chapter 2.8 Configuration.

3.5.1.1 Host Runtime overview

Responsibilities of a Host Runtime include

- Task environment setup (working directory, environment properties, command line etc.).
- Downloading packages from the Software Repository (on a task's behalf).
- Running and managing a task (spawning a process, changing task's state, exit code, etc.).
- Mediating data transfer between tasks and the rest of the framework (logs, results, etc.).

- Cleanup after tasks.
- Monitoring the host it runs on.

Each Host Runtime manages only its own tasks – it remains oblivious to the rest.

The implementation can be found in the *host-runtime* module within the `cz.cuni.mff.d3s.been.hostruntime` package.

3.5.1.2 Local task management

The Host Runtime interacts with the rest of the framework primarily by listening for messages (HostRuntimeMessageListener[10] through a distributed topic. Messages contain requests which are dispatched to appropriate message handlers (`ProcessManager`[11]).

A task begins its life on a Host Runtime with incoming `RunTaskMessage`[12] message. The Host Runtime can either accept the task or return it to the Task Manager. In former case a complete environment is prepared and a new process is spawned (`TaskProcess`[13]). This process includes:

- Downloading task BPK (`SoftwareResolver`[14])
- Creating a working directory and unpacking the BPK into it (`ProcessManager`)
- Preparing environment properties and command line (`CmdLineBuilderFactory`[15])

The task is supervised in a separate thread, waiting for the task to either finish or be aborted by a user generated request. Task state changes are propagated through the `TaskEntry`[16] structure associated with the given task through `TaskHandle`[17].

3.5.1.3 Interaction with tasks

Any communication between a task and the rest of the framework is mediated by the task's Host Runtime. This includes:

- Logs, output from standard output and standard error (`TaskLogHandler`[18])
- Results, result queries
- Task context related operations (Checkpoints, latches, etc.)

The communication protocol is based on 0MQ and messages are encoded in JSON. This allows implementing the Task API in different languages. The EverBEEN project currently implements extensive support for JVM based languages.

The output a of task is dispatched to the appropriate destination through Hazelcast distributed structures. The Host Runtime routes this information to its correct destination, but is otherwise oblivious to how such data is actually processed.

3.5.1.4 Task protocol

Follows overview of the protocol between Host Runtime and a task.

As was mentioned above the protocol is based on 0MQ with messages encoded in JSON format.

[10]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/HostRuntimeMessageListener.html
[11]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/ProcessManager.html
[12]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/protocol/messages/RunTaskMessage.html
[13]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/task/TaskProcess.html
[14]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/SoftwareResolver.html
[15]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/task/CmdLineBuilderFactory.html
[16]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskEntry.html
[17]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/task/TaskHandle.html
[18]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/hostruntime/tasklogs/TaskLogHandler.html

A task must send appropriate messages through 0MQ ports in order to communicate with its Host Runtime. Connection details are passed as environment properties upon task process spawning. Names of these environment properties are specified in `NamedSockets`[19]. Message serialization to JSON is handled in the Task API – the current implementation uses the Jackson library to serialize/deserialize messages from/to *Plain Old Java Objects*.

There are currently four types of messages recognized by the framework. For the sake of brevity, Java implementation classes are mentioned here. If the need for different implementation of the TASK API arises the message format can be inferred from their direct mapping to JSON.

Log Messages - *TaskLogs* - `LogMessage`

Example message:

```
LOG_MESSAGE#{
    "created":1378147630541,
    "taskId":"4b7c3169-7a30-4ca7-8ac1-ebb973ac0b4d",
    "contextId":"16f50281-0bb5-44d8-ab33-eea33e895b31",
    "benchmarkId":"",
    "message":{
        "name":"com.example.been.ExampleTask",
        "level":1,
        "message":"Mae govannen!",
        "errorTrace":null,
        "threadName":"main"
    }
}
```

Notice that there currently is *LOG_MESSAGE#* before the actual message.

Check Points - *TaskCheckpoints* - `CheckpointRequest`[20]

Examples of CheckPoint messages:

The first example shows the "Check Point Get" message:

```
{
    "selector":"checkpoint",
    "value":null,
    "timeout":0,
    "type":"GET",
    "taskId":"272028b5-9cba-4730-b672-385469efa7e3",
    "taskContextId":"ebbae46a-ad8f-4653-9225-49df327cb90e"
}
```

The format is the same for all types of CheckPoint messages:

`selector`
name of the requested entity

`value`
string representation of value to be passed, applicable according to message type, e.g. value of a CheckPoint to set

`timeout`
timeout in milliseconds of the request if applicable, zero means infinity

`type`
defines type of the request, supported types are to be found in `CheckpointRequestType`[21]

[19]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/socketworks/NamedSockets.html
[20]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/task/checkpoints/CheckpointRequest.html
[21]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/task/checkpoints/CheckpointRequestType.html

`taskId`
taskId of the requesting task

`taskContextId` : task context id of the requesting entity

The response might look like this:

```
{
    "replyType":"OK",
    "value":"42"
}
```

`replyType`
it's either *OK* if operation succeeded or *ERROR* otherwise

`value`
value returned from the operation, in case of ERROR reason why the operation failed

Here is the request for Count Down Latch wait with 1s timeout:

```
{
    "selector":"example-latch",
    "value":null,
    "timeout":1000,
    "type":"LATCH_WAIT",
    "taskId":"272028b5-9cba-4730-b672-385469efa7e3",
    "taskContextId":"ebbae46a-ad8f-4653-9225-49df327cb90e"
}
```

And the reply after the timeout occurred:

```
{
    "replyType":"ERROR",
    "value":"TIMEOUT"
}
```

See CheckpointController[22] implementation details of other operations.

Results - *TaskResults* - `Result`[23] along with `EntityID`[24] wrapped in EntityCarrier[25]

Let us use following example result in Java:

```
public class ExampleResult extends Result {
    public int data;
    public String name;

    /** Results must have non-parametric constructor.*/
    public ExampleResult() {}
}
```

Example result corresponding to the Java class:

[22] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/taskapi/CheckpointController.html
[23] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/results/Result.html
[24] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/EntityID.html
[25] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/EntityCarrier.html

```
{
    "created":1378149926777,
    "taskId":"1dc48ac8-8a7f-42aa-a57c-f38b8c449864",
    "contextId":"762187bb-448e-42ba-9c3e-421091553c58",
    "benchmarkId":"",
    "data":47
}
```

`created`
is time when the result was created (UNIX time)

`taskId`, `contextId`, `benchmarkId` : are IDs of the task

`data`
corresponds to the result's `data` field

Result queries - *TaskResultQueries* - `FetchQuery`[26]

Queries are a little complicated - since they allow filtering and selecting of data.

Example of a query

```
Query query = new ResultQueryBuilder().on(GROUP_ID).with("taskId", getId()).with("name",
"Name42").retrieving("data").fetch();
```

The query is translated into

```
{
    "@class":"cz.cuni.mff.d3s.been.persistence.FetchQuery",
    "id":"1ad39fd6-172a-47c7-908e-4acc1bb66414",
    "entityID":{
        "kind":"result",
        "group":"example-data"
}, "selectors":{
        "taskId":{
            "@class":"cz.cuni.mff.d3s.been.persistence.EqAttributeFilter",
            "values":{
                "@eq":"e1df89e9-b893-4099-ad21-f1eb5291a48b"
            }
        },"name":{
            "@class":"cz.cuni.mff.d3s.been.persistence.EqAttributeFilter",
            "values":{
                "@eq":"Name42"
            }
        }
    },
    "mappings":["data"]
}
```

The `@class` fields are a bit unfortunate since they refer to Java implementation classes. We acknowledge this as awkward, yet necessary – the Jackson deserializer to recognize the proper runtime type for unmarshalling.

The details of note are:

- The specification of `EntityID`[27]

[26]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/persistence/FetchQuery.html
[27]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/EntityID.html

- Selectors which filter fields
- Mappings which select which fields to fetch.

The `mappings` field is a JSON array of attribute names that should be retrieved from the persistence layer. The resulting data will only contain these fields. This feature is primarily intended for saving network traffic by limiting queries to minimal necessary information.

The `selectors` field is a JSON map containing *filters* identified by retrieved object attribute names. The filters can be any of the following:

@class	expected attributes	meaning
EqAttributeFilter	@eq	`v == @eq`
NotEqAttributeFilter	@eq	`v != @eq`
PatternAttributeFilter	@like	`v` matches the pattern in `@like`
IntervalAttributeFilter	@lo	`v >= @lo`
	@hi	`v < @hi`
	@lo, @hi	`@lo <= v < @hi`

In the above table, `v` represents the value of the filtered attribute. All the mentioned classes are taken from the `cz.cuni.mff.d3s.been.persistence` package, so their fully qualified name needs to be prefixed accordingly. For the sake of implementation simplicity, the number of filters is limited to one per attribute.

Results might look something like this:

```
{
    "@class":"cz.cuni.mff.d3s.been.persistence.DataQueryAnswer",
    "status":"OK",
    "objects":[
        "{ \"data\" : 42}"
    ]
}
```

Notice that `object` is an array of returned entities.

3.5.1.5 Host Runtime monitoring

Monitoring samples are taken through the Sigar library which uses native libraries to gather system information. The period of sampling is configurable. Samples are persisted through the Object Repository.

In case Sigar native library is not available for a platform (as is currently the case for FreeBSD 8 and later) Java fallback is provided. The Java implementation does not supply as much information as Sigar does (the striking example is information about system free memory which cannot be obtained, as far as we know, directly from Java standard libraries).

3.5.2 Task Manager

The Task Manager is at the heart of the EverBEEN framework, its responsibilities include:

- task scheduling
- context scheduling
- benchmark scheduling
- context state changes

- detection and correction of error states (benchmark failures, Host Runtimes failures, etc.)

Main characteristic:

- event-driven
- distributed
- redundant (in default configuration)

3.5.2.1 Distributed approach to scheduling

The most important characteristic of the Task Manger is that the computation is event-driven and distributed among the *DATA* nodes. The implication from such approach is that there is no central authority, bottleneck or single point of failure. If a data node disconnects (or crashes), its responsibilities (along with related data) are transparently taken over by the rest of the cluster.

Distributed architecture is the major difference from previous versions of the BEEN framework.

3.5.2.2 Implementation

The implementation of the Task Manager is heavily dependent on Hazelcast distributed data structures and its semantics, especially the `com.hazelcast.core.IMap`[28].

3.5.2.3 Workflow

The basic event-based work flow:

1. Receive asynchronous Hazelcast event
2. Generate appropriate message describing the event
3. Generate appropriate action from the message
4. Execute the action

Internal message handling is also message-driven, based on the 0MQ library, somewhat resembling the Actor model. The advantage resides in separating message reception and deserialization from actual handling logic. Internal messages are executed in one thread, which also removes the need for explicit locking and synchronization (which happens, but is not responsibility of the Task Manager developer). A more detailed description of the *message/action* is a part of the source code and associated JavaDoc[29], and can be found in the `cz.cuni.mff.d3s.been.manager.msg` and `cz.cuni.mff.d3s.been.manager.action` packages.

3.5.2.4 Data ownership

An important concept to remember is that an instance of the Task Manager only handles entries it owns whenever possible (e.g. task entries). Data ownership means that the object in question is stored in local memory and the node is responsible for it. The design of Task Manager takes advantage of the locality and most operations are local with regard to data ownership. This approach is highly desirable for the Task Manger to scale.

3.5.2.5 Main distributed structures

- BEEN_MAP_TASKS - map containing runtime task information
- BEEN_MAP_TASK_CONTEXTS - map containing runtime context information
- BEEN_MAP_BENCHMARKS - map containing runtime context information

These distributed data structures are also backed by the MapStore (enabled by default).

[28] http://www.hazelcast.com/javadoc/com/hazelcast/core/IMap.html
[29] http://www.everbeen.cz/javadoc/everBeen/index.html

3.5.2.6 Task scheduling

The following section discusses task states, which are described in detail in section 2.2.2 (Basic concepts) of the user manual.

The Task Manager is responsible for scheduling tasks, which boils down to finding a Host Runtime on which the task can run. The description of possible restrictions can be found in the 3.5.1 Host Runtime section.

A distributed query[30] is used to find suitable Host Runtimes, spreading the load among `DATA` nodes.

An appropriate Host Runtime is also chosen based on Host Runtime utilization, less loaded Host Runtimes are preferred. Among equal hosts a Host Runtime is chosen randomly.

The lifecycle of a task is commenced by inserting a `TaskEntry`[31] in `SUBMITTED` state into the task map under a random key. Inserting a new entry to the map causes an event which is handled by the owner of the key — the Task Manager responsible for the key. The event is converted to a `NewTaskMessage`[32] object and sent to the processing thread. The handling logic is separated in order not to block the Hazelcast service threads. In this regard, message handling is serialized on the particular node. The message then generates `ScheduleTaskAction`[33], which is responsible for figuring out what to do. Several things might happen

- the task cannot be run because it's waiting on another task, the state is changed to WAITING
- the task cannot be run because there is no suitable Host Runtime for it, the state is changed to WAITING
- the task can be scheduled on a chosen Host Runtime, the state is changed to SCHEDULED and the runtime is notified.

If the task is accepted the chosen Host Runtime is responsible for the task until it finishes or fails.

`WAITING` tasks remain under the responsibility of the Task Manager, which can try to reschedule when an event occurs, e.g.:

- another tasks is removed from a Host Runtime
- a new Host Runtime is connected

3.5.2.7 Benchmark Scheduling

Benchmark tasks are scheduled in the same fashion as other tasks. The main difference is that if a benchmark task fails (host failure, programming error, etc.), the framework can re-schedule the task on a different Host Runtime.

A problem can arise from re-scheduling an incorrectly written benchmark which fails too often. There is a configuration option which limits how many re-submits are allowed for a benchmark task.

3.5.2.8 Context Handling

Contexts are not scheduled as an entity on Host Runtimes, as they are mere containers for related tasks. The Task Manager handles detection of contexts state changes. The state of a contexts is decided from the states of its tasks.

Possible task context states:

- WAITING – for future use
- RUNNING – contained tasks are running, scheduled or waiting to be scheduled

[30] http://hazelcast.com/docs/2.6/manual/single_html/#MapQuery
[31] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskEntry.html
[32] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/msg/NewTaskMessage.html
[33] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/action/ScheduleTaskAction.html

- FINISHED – all contained tasks finished without an error
- FAILED – at least one task from the context failed

Future improvements may include heuristics for scheduling contexts as an entity (i.e. detection that the context can not be scheduled at the moment), which is difficult because of the distributed nature of scheduling – any information gathered might be obsolete by the time it is read.

3.5.2.9 Handling exceptional events

The current Hazelcast implementation (as of version 2.6) has one limitation. When a key migrates[34] the new owner does not receive any event (`com.hazelcast.partition.MigrationListener`[35] is not very useful in this regard, since it does not contain enough information). This might be a problem, for example when a node crashes and an event of type "new task added" is lost. To mitigate the problem the Task Manager periodically scans (`LocalKeyScanner`[36]) its *local keys* looking for irregularities. If an anomaly is found, a message is created to remedy the problem.

There are several situations where similar problems might arise:

- Host Runtime failure
- Key migration
- Cluster restart

Note that the `LocalKeyScanner` solution is mainly a safety net – most of the time the framework will receive an event on which it can react appropriately (e.g. Host Runtime failed).

In the case of cluster restart, there might be stale tasks which do not run anymore. In such cases, the task state information loaded from the MapStore will be inconsistent. Such situation are recognized and corrected by the scan.

3.5.2.10 Hazelcast events

These are main sources of cluster-wide events, received from Hazelcast:

- Task Events – in `LocalTaskListener`[37]
- Host Runtime events – in `LocalRuntimeListener`[38]
- Contexts events – in `LocalContextListener`[39]

3.5.2.11 Locking

Certain EverBEEN objects are possibly concurrently modified by different services (and possibly different nodes). One of such objects is the `TaskEntry`, which is accessed by both a Task Manager and a Host Runtime. Unfortunately, such cases must be be resolved through the usage of distributed Hazelcast locks. Such locking is costly, so we tried to avoid it on performance critical paths. Moreover, the number of parties trying to obtain the lock is never high. In the case of `TaskEntry`, concurrent accesses are attempted by one Host Runtime and at most two Task Manager instances (two in case of a key migration), and the locks are owned by the task's current Task Manager.

The recently released Hazelcast 3.0 introduced the `Entry Processor`[40] feature that could help improve throughput, should the need arise.

[34]http://hazelcast.com/docs/2.5/manual/single_html/#InternalsDistributedMap
[35]http://www.hazelcast.com/javadoc/com/hazelcast/partition/MigrationListener.html
[36]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/LocalKeyScanner.html
[37]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/LocalTaskListener.html
[38]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/LocalRuntimeListener.html
[39]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/manager/LocalContextListener.html
[40]http://hazelcast.com/docs/3.0/manual/single_html/#MapEntryProcessor

3.5.3 Software Repository

From user perspective, the Software Repository is a black box performing storage and retrieval of standalone BPK packages with task, task context and benchmark definitions. All user interaction with the Software Repository is mediated by the Web Interface. From a developer's perspective, the architecture of the Software Repository is based on file system storage and a very simple HTTP protocol.

3.5.3.1 HTTP

The Software Repository HTTP protocol supports the following actions:

- *get* **/bpk** - download BPK from software repository
  - request header: `Bpk-Identifier`, value: `cz.cuni.mff.d3s.been.bpk.BpkIdentifier`[41] (JSON), unique identifier of the BPK to be downloaded
  - valid response status codes: *2XX*
  - response body: binary content of the requested BPK file
- *put* **/bpk** - upload BPK to software repository
  - request header: `Bpk-Identifier`, value: `BpkIdentifier` (JSON), unique identifier uploaded BPK
  - request body: binary content of the uploaded BPK file
  - valid response status codes: *2XX*
- *get* **/bpklist** - list all BPKs stored in the Software Repository
  - valid response status codes: *2XX*
  - response body: `List<BpkIdentifier>` (JSON)
- *get* **/tdlist** - list all task descriptors[42] for a BPK stored in the Software Repository (identified by given `BpkIdentifier`)
  - request header: `Bpk-Identifier`, value: `BpkIdentifier` (JSON), unique identifier of the BPK for which the list of available descriptors should be returned
  - valid response status codes: *2XX*
  - response body: `Map<String, TaskDescriptor>` (JSON), the map key set are task descriptor file names
- *get* **/tcdlist** - list all task context descriptors[43] for BPK stored in Software Repository (identified by given `BpkIdentifier`)
  - request header: `Bpk-Identifier`, value: `BpkIdentifier` (JSON), unique identifier of the BPK for which the list of available descriptors should be returned
  - valid response status codes: *2XX*
  - response body: `Map<String, TaskContextDescriptor>` (JSON), the map key set are task context descriptor file names

If response is marked with an invalid status code, the standard HTTP response reason phrase will contain the reason of the failure. We chose the HTTP protocol for BPK transport, because it is better suited for large file transfers.

For JSON serialization and deserialization we use the `ObjectMapper` provided by the Jackson library.

[41]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/bpk/BpkIdentifier.html
[42]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskDescriptor.html
[43]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/task/TaskContextDescriptor.html

3.5.3.2 File system structure

Software Repository stores uploaded BPKs in the `bpks` subdirectory of its configurable working directory root. Each uploaded BPK is stored on the following path:

```
{groupId}/{bpkId}/{version}/{bpkId}-{version}.bpk
```

- *{groupId}* stands for the fully qualified groupId of BPK with dots substitued by slashes
- *{bpkId}* stands for the bpkId the BPK
- *{version}* stands for BPK version

To clarify, here is an example of Software Repository directory structure:

```
SR working directory (WD):
    e.g. /home/been/swrepository on Linux systems
    e.g. C:\been\swrepository on Windows systems

BPK store directory:
    {WD}/bpks

uploaded example BPK 1:
    filename: example.bpk
    groupId:  cz.cuni.mff.d3s.been.example
    bpkId:    example-bpk
    version:  1.1.beta-02

uploaded example BPK 2:
    filename: alpmexa.bpk
    groupId:  cz.cuni.mff.d3s.been.example
    bpkId:    alpmexa-bpk
    version:  0.1-SNAPSHOT

example BPK 1 will be stored in:
    {WD}/bpks/cz/cuni/mff/d3s/been/example/example-bpk/
                      1.1.beta-02/example-bpk-1.1.beta-02.bpk

example BPK 2 will be stored in:
    {WD}/bpks/cz/cuni/mff/d3s/been/example/alpmexa-bpk/0.1-SNAPSHOT/
                      alpmexa-bpk-0.1-SNAPSHOT.bpk
```

Some limitations:

- Software repository does not support BPK overwriting (uploading a BPK with the same groupId, bpkId and version as a BPK already present in the Software Repository). The only exception to this rule are BPKs with a version string suffixed by **-SNAPSHOT** (e.g. 1.0.0-SNAPSHOT).
- You have to start software repository on node visible for all other nodes and on port which is not blocked by the host's firewall.
- The Software Repository listens on the primary network interface selected by Hazelcast for cluster communication. We realize this might inconvenience you if you are running EverBEEN on atypical networks, and intend to add some configuration options to let you specify desired behavior manually.
- There is an **artifacts** folder in the Software Repository working directory root. This is because Software Repository implements uploading and downloading Maven artifacts, in addition to BPKs, but the feature has not yet been integrated into the rest of EverBEEN, and is staged for future development.

3.5.3.3 Software Repository client

Because some BPKs can be used multiple times on single Host Runtime, each host runtime has its own software repository cache. This cache uses the same file system structure as Software Repository does, and transparently reuses downloaded BPK bundles to save bandwidth and I/O resources.

3.5.4 Object Repository

The purpose of the *Object Repository* is to service user data persistence. While the actual persistence and querying code is isolated from the *Object Repository* by the `Storage`[44] interface and is database-dependent (the default MongoDB implementation can be found in the `mongo-storage` module), the *Object Repository* operates without any knowledge of user types or concrete database storage implementation. The main portion of its work is to communicate with the rest of the EverBEEN cluster, collect objects sent by other nodes for persistence, collect queries from other nodes and dispatch answers. The communication with the rest of the cluster is realized through shared queues and maps (distributed cluster-wide memory).

The *Object Repository* also features a *Janitor* sub-service, which is responsible for cleaning up old data once it is deemed unnecessary. The *Janitor* works on its local *Storage* instance and therefore doesn't partake in any cluster-wide activities.

3.5.4.1 Queue drains

As mentioned above, the *Object Repository*'s communication with the rest of the EverBEEN cluster is mostly based on distributed queues. The *Object Repository* continuously drains these queues using special *consumer* threads (spawned dynamically based on load-balancing heuristics). This idea is revisited in both persist requests and querying.

3.5.4.2 Persist request queue

The object persisting mechanism is simple:

- A node serializes its object `o` (`Entity`[45]) into JSON. Let's call the resulting string `ojson`.
- The node creates an special wrapper (`EntityCarrier`[46]) which combines the serialized object with a destination id (`EntityID`[47]) - let's call the specific id instance `oid`.
- The wrapper, containing both `ojson` and `oid`, gets submitted into a distributed queue.
- A few moments later, an *Object Repository* drains the wrapper from the distributed queue.
- The repository unpacks the wrapper and passes both `ojson` and `oid` to its *Storage* implementation.
- The locating conventions of the *Storage* implementation are transparent to the *Object Repository.*

If the *Storage* implementation refuses to store `ojson` for any reason, the *Object Repository* resubmits the wrapper, containing `ojson` and `oid`, back to the shared queue to prevent data loss.

From the above principle, it is obvious that multiple *Object Repository* instances can operate concurrently, without a negative impact on data integrity or performance. The condition is, however, that all of the *Object Repository* instances be accessing either the same database, or that the databases so accessed have a full data-sharing policy of their own.

Persist requests in EverBEEN are asynchronous, and no notification is sent back after a persist is done. Although this approach may limit the user's knowledge about the current state of his data, it comes at a considerable advantage: The shared memory can function as a buffer through *Object Repository* disconnects. This enables a hassle-free means of reconfiguring the *Object Repository* if need be.

[44] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/storage/Storage.html
[45] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/Entity.html
[46] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/EntityCarrier.html
[47] http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/core/persistence/EntityID.html

3.5.4.3 Query queue & Answer map

A similar approach regarding queues is taken for persistence layer queries. Just as serialized persistent objects do, queries get submitted to a distributed queue, where they wait for the *Object Repository* to process them. However, queries naturally need to provide an answer to the requesting party, so an object needs to be sent back. This is realized through a distributed map with listeners. To facilitate control flow for the requesting party, we made the query calls synchronous. The querying process is as follows:

- The requesting party creates a query.
- If the requesting party is a *task*, the query is serialized, sent to the corresponding *Host Runtime* and deserialized. The *Host Runtime* becomes the new requesting party while the *task* blocks in wait for an answer from the *Host Runtime.*
- The requesting party registers a listener on the query's ID in the distributed answer map.
- The requesting party submits the query to the distributed query queue.
- The requesting party blocks in wait for the answer to its query.
- Once the answer appears in the distributed answer map, the requesting party picks it up, removes it from the answer map and resumes processing.

Of course, such blocking behavior is prone to potential infinite waits in various corner-cases. To prevent that from happening, queries are subject to two types of timeout:

Query timeout
The requesting party only waits for this period of time for an answer to appear in the distributed answer map. If the answer doesn't appear in time, the requesting party attempts to cancel the query altogether by withdrawing it from the distributed query queue to prevent clotting the answer map with unused answers.

Processing timeout
If the answer doesn't appear in time, but the query can not be withdrawn from the distributed queue, it is assumed that an *Object Repository* instance has picked the query up, but did not yet process it. In such case, the requesting party waits for the *processing timeout* duration to give the *Object Repository* time to process the request. If the *Object Repository* responds within that interval, the answer it provided is returned normally. If the *processing timeout* is hit instead, a special timeout answer is returned instead.

Both of these timeouts are implemented on the client side to ensure that the requesting party always gets a valid answer or a timeout, even in case of unpredictable situations. Clearly, the maximum waiting time before the requesting party is guaranteed to receive an answer is `total_timeout = query_timeout + processing_timeout`.

For cases when the `total_timeout` is systematically being hit (as unlikely as they may be), there is a local eviction policy on answers submitted to the map, with `TTL = 5 * total_timeout`. That means answers submitted to the distributed answer map will be automatically deleted once the TTL expires.

3.5.4.4 Janitor

Every instance of *Object Repository* has its own *Janitor* thread that periodically checks the *Storage* for old objects and removes them. To enable this kind of cleanup, EverBEEN stores some service entries about *task* and *context* states, which are deleted once the cleanup of all other entries related to that *task* or *context* has been performed. The cleanup rules are as follows:

- EverBEEN features two configurable TTL properties:
  - `been.objectrepository.janitor.finished-longevity`
  - `been.objectrepository.janitor.failed-longevity`
- For successfully finished *tasks* and *contexts* past *finished longevity*, configurations (*descriptors*), results and evaluations thereof are kept, but service information (logs) are deleted

- For failed *tasks* and *contexts* past *failed longevity*, all entries are deleted

All of these deletes are implemented using queries similar to `DELETE FROM xyz AS o WHERE o.att='abcd'`, so even if multiple instances of *Janitor* are running and they all attempt to perform cleanup after the same *task* or *context*, the deletes do not result in failures.

There is a hypothetical case when the *Janitor component* performs a sweep which successfully deletes leftover information about a *task* or *context* and is followed by a persist of leftover data for that same *task* (*context*). This would mean that the late persisted object will never be deleted. It would take the following for this case to occur:

- Both the initial and terminal states of the *task* (*context*) get persisted, but some leftover data doesn't. That can happen due to a persist queue reorder (potentially due to a temporary *Storage* failure resulting in a requeue).
- *Object Repository* gets disconnected after the initial and terminal state has been drained, but before the late persisted object has been drained.
- *Object Repository* doesn't get reconnected for at least `been.objectrepository.janitor.finished-longevity` (or `been.objectrepository.janitor.failed-longevity`, depending on the terminal state of the *task*/*context*), but keeps running (or gets restarted with a bad network configuration).

This case is not handled, mainly because the default values for both longevities are in the order of days, and it would take the user not noticing an invalid cluster configuration for this long.

3.5.5 Map Store

MapStore allows EverBEEN to persist runtime information, which allows for a state restore after a cluster-wide restart or crash.

3.5.5.1 Role of the MapStore

EverBEEN runtime information (such as task, context and benchmark states) are persisted through the MapStore. This adds overhead to working with the distributed objects, but allows restoring of the state after a cluster restart, providing a user with more concise experience.

The implementation is build atop of Hazelcast Map Store - mechanism for storing/loading of Hazelcast distributed objects to/from a persistence layer. The EverBEEN team implemented a mapping to MongoDB.

The main advantage of using the MapStore is transparent and easy access to Hazelcast distributed structures with the ability to persist them - no explicit actions are needed.

3.5.5.2 Difference between the MapStore and the Object repository

Both mechanism are used to persist objects - the difference is in the type of objects being persisted. The Object Repository stores user generated information, whereas the MapStore handles (mainly) EverBEEN runtime information, which is essential for proper framework functioning.

The difference is also in level of transparency for users. Object persistence happens on behalf of an explicit user request, while MapStore works "behind the scene".

3.5.5.3 Extension point

Porting the *MapStore* adapter to a different persistence layer (such as a relational database) is relatively easy. By implementing the `com.hazelcast.core.MapStore` interface and specifying the implementation class at runtime.

3.5.5.4 Configuration

The *MapStore* layer can be configured to accommodate different needs:

- specify connection options (hostname, user, etc.)
- enable/disable
- change implementation
- write-through and write-back modes

Detailed description of configuration can be found in section 2.8 MapStore Configuration.

3.5.6 Web Interface

The EverBEEN web interface is a sophisticated utility able to monitor and control the EverBEEN cluster. It is not actually a real service but rather a standalone client. Nevertheless it is an indispensable part of the framework. Its implementation is based on the Tapestry5[48] framework and its extension, Tapestry5-jquery[49]. Describing the principles and conventions of Tapestry framework is not a part of the EverBEEN documentation but can be found on the official site of the framework. We would, however, like to include some information which could be helpful for Web Interface extenders.

3.5.6.1 Dependency Injection

Tapestry uses its own implementation of dependency injection called Tapestry IoC (Inversion of Control). This container is responsible for managing dependencies among pages, components, services and other parts of the application. Tapestry has several of its own services and we added two more:

- The `BeenApiService` is the most important, because it is in charge of cluster connection.
- The `LiveFeedService` handles communication with web browsers through web sockets.

These services are fully integrated to the Tapestry web application life cycle and can be injected to pages and components through standard Tapestry annotations.

3.5.6.2 Pages and Components

All pages are inherited from the base `Page` class. This class contains an injected instance of `BeenApiService`, from which you can obtain an instance of BeenApi[50]. The `BeenApi` enables you to manage the whole EverBEEN cluster. The global EverBEEN layout is defined by the `Layout` component. And all JavaScript and CSS resources can be found in the `src/main/webapp` subdirectory of the `web-interface` module.

3.5.6.3 Connecting WI to the cluster

Web interface is connected to the cluster using Hazelcast native client. It means that the Web Interface does not store any data and does not own (manage) any Hazelcast shared objects.

3.6 Modular approach

From the start, EverBEEN was developped as a modular project, and we backed our decision by Apache Maven[51] as EverBEEN's building tool from day one. The major benefit of this decision is easier code maintenance in the future, and cleaner code in general.

[48]http://tapestry.apache.org/
[49]http://tapestry5-jquery.com/
[50]http://www.everbeen.cz/javadoc/everBeen/cz/cuni/mff/d3s/been/api/BeenApi.html
[51]http://maven.apache.org/

3.6.1 Module overview

EverBEEN's modules can be categorized as follows.

3.6.1.1 Service modules

A subset of EverBEEN's modules corresponds exactly to the set of former WillBEEN services. The main motivation for such separation is to bar any potential in-code dependencies between services. That takes any cross service bug propagation (common with the use of RMI) out of the equation, making EverBEEN much less error-prone.

`host-runtime`
Lends an EverBEEN node the ability to run tasks and benchmarks. Keeps track of the node's hardware, OS and load. Handles all the house-keeping around task processes.

`object-repository`
Makes this node a bridge between EverBEEN and a persistence layer. Also gives the node the ability to handle persistence layer requests, run persistence layer cleanup etc.

`software-repository-server`
Enables the node to store and distribute *BPK* bundles (packages with user software). Needed for EverBEEN to be able to run tasks. At most one should be present in the EverBEEN cluster at any time.

`task-manager`
Enable task planning on this node. All DATA nodes run this service.

`web-interface`
A Java container (e.g. Apache Tomcat[52] webapp able of connecting to the EverBEEN cluster. Serves as the GUI component of the system.

3.6.1.2 Internal API modules

Most of the places where EverBEEN bridges with a major piece of third-party technology are separated by an internal API.

`been-api`
A general interface that covers interaction between the user and EverBEEN. All operations done through the GUI (web interface) go through the been API.

`mapstore`
Not an API 'per se', this module contains the definition of EverBEEN configuration properties related to the Hazelcast mapstore implementation used for EverBEEN service data storage.

`storage`
Generic persistence layer interface that covers user object storage and retrieval.

`service-logger`
Simple protocol that covers EverBEEN node log message submission to the cluster. Enables persistent storage of EverBEEN log messages and unified access to the logs of all cluster nodes.

`software-repository-store`
Persistence layer interface for storing user software bundles. Used by the *Software Repository* service as persistence and by the *Host Runtime* service as a cache.

[52]http://tomcat.apache.org/

3.6.1.3 Internal API default implementations

Of course, implementations of the internal API modules are extracted to separate modules as well. None of these are hardcoded to EverBEEN. Various means are used instead to inject the implementations at runtime.

`logback-appender`
A Logback appender[53] that pushes local log messages back to the cluster via the interface provided by `service-logger`.

`mongo-storage`
MongoDB implementation of the `storage` module.

`mongodb-mapstore`
MongoDB implementation of the Hazelcast MapStore[54] and MapStoreFactory[55]

`software-repository-fsbasedstore`
File system based implementation of the `software-repository-store` module.

3.6.1.4 System modules

Some of EverBEEN's modules provide additional functionality to existing EverBEEN components, and therefore do not quite make the case for an internal API.

`core-cluster`
Covers clustering mechanics (e.g. connection, data sharing etc.) and distributed data structure naming conventions.

`debug-assistant`
Enables task and benchmark JPDA support (remote debugging).

`detectors`
Performs hardware and OS detection on the host running the *Host Runtime*. Enables load monitoring.

`service-logger-handler`
Listens for log messages on the protocol defined in `service-logger` and pushes them into the cluster.

`socketworks-clients`
The client-side bundle for EverBEEN socket messaging. Used mainly in `task-api` and `benchmark-api` to communicate with `host-runtime`.

`socketworks-servers`
The server-side bundle for EverBEEN socket messaging. Used mainly in `host-runtime` to handle requests from `task-api` and `benchmark-api`.

`software-repository-client`
Client for the `software-repository-server`, used by `host-runtime` to fetch software bundles needed to run a task.

3.6.1.5 Protocol object modules

As mentioned before, EverBEEN services do not communicate directly. Instead, they do so by placing well-known object into well-known data structures within cluster-wide shared memory. These modules contain the definitions of types transfered between services.

`bpk-conventions`
Contains constants and utility methods for the *BPK* bundle format.

[53]http://logback.qos.ch/manual/appenders.html
[54]http://www.hazelcast.com/javadoc/com/hazelcast/core/MapStore.html
[55]http://www.hazelcast.com/javadoc/com/hazelcast/core/MapStoreFactory.html

`checkpoints`
Provides special request types for checkpoint state communication (checkpoints are a task synchronization primitive).

`core-data`
Basic EverBEEN objects. Contains protocol classes known to nearly all EverBEEN components. Most of these classes are defined using XSDs, which are then compiled to Java using the `xjc` compiler.

`core-protocol`
Defines a task control-flow protocol, used by the *Task Manager* to transmit commands regarding tasks to *Host Runtimes*.

`persistence`
Persistent EverBEEN objects. Mostly XSD-defined classes for well-known persistable objects.

`results`
Contains definitions of persistent objects that represent task outputs.

`software-repository`
Constants defining conventions for the communication between `software-repository-client` and `software-repository-server`.

3.6.1.6 User API modules

The EverBEEN environment expects to run user code. Therefore, some modules need to provide a separate API which enables the user-programmed runtime to interact with the system.

`benchmark-api`
User API for *benchmark* generation.

`bpk-plugin`
An Apache Maven plugin that aids the user in assembling *task* and *benchmark* software bundles *BPK*.

`task-api`
User API for *task* creation.

3.6.1.7 Utility modules

As virtually any project, even EverBEEN has its own flavor of utilities.

`util`
The regular bundle of ubiquitous utility methods and classes.

`xsd-catalog-resolver`
Mild hack of the `com.sun.org.apache.xml.internal.resolver.tools.CatalogResolver` class. Enables XSD imports from `jar` files, which is broken in the default implementation. Necessary for JAXB Maven plugins to be able to resolve inheritance.

`xsd-export`
In-package support for the `xsd-catalog-resolver`. Helps find the package resources (XSDs) and hands them to the resolver.

3.6.1.8 Deployment modules

Some EverBEEN modules were created with the sole purpose of deploying existing modules in some particular way.

`node`
Defines a configurable runnable class that launches an EverBEEN node, along with services specified using command-line options.

`node-deploy`
Helps assemble the `node` package along with all dependencies (service modules etc.) into an executable `jar` file.

`web-interface-standalone`
Provides support for the `web-interface` to be run in an embedded java container, in addition to manual deployment to a java container.

`mongo-storage-standalone`
Instantiates the MongoDB storage implementation over a MongoDB instance deployed at runtime. Used for `mongo-storage` module testing.

3.6.1.9 Debugging tools

EverBEEN features a pair of modules that provide command-line debugging tools.

`client-shell`
An interactive command-line client for EverBEEN. Intended a command-line alternative to `web-interface`. Still in incubation phase.

`client-submitter`
An executable jar designed to quickly connect to the EverBEEN cluster and submit a task. Useful for debugging task code.

3.7 Used technologies

Follows overview of used technologies in EverBEEN.

3.7.1 Hazelcast

Hazelcast[56] is the most important third-party framework used by our project. It is a highly scalable and configurable in-memory data grid. We chose the framework mostly because :

- provides automatic memory load ballancing between connected nodes
- provides failover data redundancy
- provides atomic acces to objects stored in the cluster
- provides SQL selectors for filtering stored data
- is highly scalable and configurable
- is fast and tested by many developers

We use the Community edition, which is open-source but has some (minor) limitations:

- All data are stored on the JVM heap of connected nodes - this may cause OutOfMemory problems when storing big amounts of data. In enterprise edition the off-heap technology can be used.
- Hazelcast Management Center (web console) is restricted to two nodes maximum.
- The Community Edition does not contain cluster security features.

[56] http://www.hazelcast.com/

3.7.2 0MQ

0MQ[57] is a message passing library which can also act as a concurrency framework. It supports many advanced features. Best source to learn more about the library is the official 0MQ Guide[58]:

The EverBEEN team chose the library as the primary communication technology between a Host Runtime and its tasks, especially because of:

- focus on message passing
- multi-platform support
- ease-of-use compared to plain sockets
- extensive list of language bindings
- support for different message passing patters
- extensive documentation

We decided to use the Pure Java implementation of libzmq[59] because of easier integration with the project without the need to either compile the C library for each supported platform or add external dependency on it.

As an experiment the Task Manager internal communication has been implemented on top of the library as well using the inter-process communication protocol, somewhat resembling the Actor concurrency model.

3.7.3 Apache Maven

Apache Maven[60] is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

3.7.4 Apache Commons

Instead of re-inventing the wheel once more our team decided to use time-proven Apache Commons[61] set of libraries for various purposes.

3.7.5 Apache HTTP Core/Components

Apache HTTP components[62] is a library focused on HTTP and associated protocols. The Software Repository server and client is based on this library.

3.7.6 Jackson

Jackson[63] is a fast, zero-dependency, lightweight and powerful Java library for processing JSON data.

3.7.7 JAXB

JAXB - Java XML Binding is standard which specifies how to map Java classes to XML representations. We use Maven 2 JAXB 2.x Plugin[64] which generates Java classes from XSDs.

[57]http://zeromq.org/
[58]http://zguide.zeromq.org/
[59]https://github.com/zeromq/jeromq
[60]http://maven.apache.org/
[61]http://commons.apache.org
[62]http://hc.apache.org/
[63]http://jackson.codehaus.org/
[64]https://java.net/projects/maven-jaxb2-plugin/pages/Home

3.7.8 Logback (logging implementarion)

Logback[65] is Java logging framework. It is intended as a successor to the popular “log4j” project. EverBEEN uses it as the logging mechanism.

3.7.9 MongoDB

MongoDB[66] is a cross-platform document-oriented database system which classifies as a “NoSQL” database. Persistence layer backend of EverBEEN is built on top of it.

3.7.10 SLF4J (logging interface)

The Simple Logging Facade for Java (SLF4J[67]) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time. The SLF4J is extensively used in EverBEEN as the logging facade.

3.7.11 Tapestry5

Apache Tapestry[68] is an open-source framework for creating dynamic, robust, highly scalable web applications in Java or other JVM languages. Tapestry complements and builds upon the standard Java Servlet API, and so it works in any servlet container or application server. The Web Interface is build on top of it.

3.7.12 Twitter Bootstrap

Twitter Bootstrap[69] is sleek, intuitive, and powerful front-end framework for faster and easier web development. Used in EverBEEN for the Web Interface design.

3.8 Current limitations and future work

The EverBEEN project, as any big project, has some limitations and opportunities for improvement. This chapter summarizes them and suggests possible directions which might be explored in future.

Support for non JVM-based tasks
The EverBEEN framework is fully capable of running non-JVM based tasks, such as scripts and binaries. What is missing is fully integrated environment for such tasks in the form of `native` Task API. The implementation of a native Task API should be straight forward. The protocol is described in the Host Runtime documentation. Preliminary work has begun on support for scripts, in form of a Python script - due to time constraints the support is in incubator phase. On the other hand the integration and support for JVM-based tasks is so extensive that most tasks can be easily implemented in it (including running of native binaries, commands and scripts).

BPK and artifact dependencies
Currently BPKs are created as self-contained. The original plan was to resolve dependencies as part of the task initialization process (similar to the Maven way of resolving and downloading dependencies). The success of the `bpk-maven-plugin` pushed such feature more or less aside as it was deemed not necessary at the moment. Implementation of such feature could reduce size of BPKs and decrease network usage (which is currently of no concern, since BPKs are relatively small).

[65]http://logback.qos.ch/
[66]http://www.mongodb.org/
[67]http://www.slf4j.org/
[68]http://tapestry.apache.org/
[69]http://getbootstrap.com

Command-line client
The command line client introduced in the WillBEEN project is not supported. The so called `bcmd` client and its accompanied service were sophisticated pieces of code. Porting the code to current architecture would amount to time consuming work which we lacked. The client could be for example implemented in Java using the BEEN API (the same API the Web Interface uses). Preliminary work has been done in this area (`client-shell` module) - the code is in incubator phase. While interesting feature, real use case should also be presented.

Results triggers are not supported
We feel that the removal of Result triggers introduced in WillBEEN is for the better. The fundamental problem with triggers was that there was no way how to debug/test them - the same problem as with the old Benchmark Manager API. In EverBEEN evaluators are normal tasks which can be run through the Benchmark API. If there is a real use case, triggers could be implemented using Hazelcast events.

Support for big files
Because the architecture of the EverBEEN framework depends on in-memory storage of data, transferring of big files is not recommended. We assume that benchmarking will be done in controlled environment, where the deployment of a network file system is not a problem if need be. Recently released version 3.0 of Hazelcast contains features which might be useful in implementing such feature. Or a separate service could be implemented - in which case we opt for the usage of network file system.

Database backend for the Software Repository
Interesting improvement might be adding database backend for the Software Repository. The feature is on the wish list but due to lack of time and resources was not implemented. The Software Repository was designed to easily change backends.

Decentralized Software Repository
Currently the Software Repository is centralized service. It might be interesting to explore new features in recently released Hazalcast 3.0[70] to allow cluster wide file distribution.

[70]http://hazelcast.com/docs/3.0/manual/single_html/