INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

CO[CO]^{2}NUT

Concurrently Virtualising User Code Compilation

Elisabeth Lempa

Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Niels Heller
Abgabe am	19.06.2018

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 19.06.2018 Elisabeth Lempa

Abstract

This thesis describes the conception and development of a concurrent virtualisation software for the e-learning platform Backstage 2. This *concurrent code compile unit* "co[co]^{2}nut" ties a REST-API to dynamic creation of Docker containers in order to compile and execute user code in a secure, virtualised environment.
The conceptual software architecture is explained, and reasons for design choices, especially regarding the scheduling of concurrent requests, are given.
The implementation of the application's main components and their interactions are described in-depth. The API of the REST server is documented, and a brief guide on configuration and set-up of the server is given.
The performance of the software, especially in a concurrent setting, is discussed. Different factors that have an effect on the performance, like the use of a RAM drive, are analyzed.
Finally, there is some discussion of extensibility of the software to accommodate additional use cases.

Zusammenfassung

Diese Bachelorarbeit dient der Beschreibung von Konzeptualisierung und Entwicklung einer parallelisierenden Virtualisierungssoftware für die E-Learning-Plattform Backstage 2.
Diese *concurrent code compile unit* "co[co]^{2}nut" bildet Anfragen an eine REST-API auf die dynamische Erzeugung von Docker-Containern ab, um User-Code in sicherer, vitualisierter Umgebung zu kompilieren und auszuführen.
Bestandteile der Arbeit sind unter anderem eine Beschreibung der Software-Architektur, insbesondere Erläuterung der Hintergründe spezifischer Design-Entscheidungen bezüglich des Schedulings parallel bearbeiteter Anfragen.
Außerdem enthält die Arbeit eine Beschreibung der Implementierung der Hauptkomponenten der Software, sowie ihrer Interaktion miteinander, eine Dokumentation der API des REST-Servers, und eine kurze Anleitung zur Konfiguration und Inbetriebnahme des Server-Programms.
Die Performanz der Software, insbesondere bei paralleler Ausführung, wird diskutiert, ebenso wie verschiedene Faktoren die auf diese Performanz Einfluss haben, wie beispielsweise die Verwendung einer RAM-Disk.
Zuletzt wird die Erweiterbarkeit der Software im Hinblick einiger weiterer Anwendungsfälle besprochen.

Acknowledgments

I would first like to thank my thesis supervisor Prof. Dr. François Bry. He allowed me to complete this interesting and challenging project, and provided his continuously valuable insight and remarks during the entire process of its creation.
Furthermore, I would like to thank my advisor Niels Heller. His door was not only always open, I also spent a very large portion of my time on the interior side of it. He has offered me not only the space to create this project as my own work, both literally and figuratively, but also an immeasurable amount of valuable insight, advice, and puns.
I also would like to acknowledge that the enlightening idea of using a RAM disk for performance optimisation was Sebastian Mader's. I cannot allow for it to go unmentioned here, especially because it came completely unprompted, and from another room.
I was able to complete this project in a very welcoming and supporting environment, which all of the members of the PMS teaching and research unit, (as well as, to some extent, the TCS teaching and research unit, simply due to proximity and sharing of a kitchenette) helped create. Working with them (or just adjacent to them) was delightful.
And finally, I would like to mention my good friend Tobias Rosenkranz, who has offered his support in any and all LaTeX related questions, at any and all times of the day (and night). He made this thesis look the part.

I am extremely grateful for this experience, and for all of your collective help, insight and support, and my gratitude goes to all of you!

Thank you!

Contents

CHAPTER 1

Introduction

In this introduction, the concept of the code compile unit software will be briefly motivated. The basic principles of how it is used will be presented. Some use cases in relation to Backstage will be introduced as well.
This thesis contains five chapters. The first chapter is this introduction. The second chapter describes the conception of the software, explaining the objectives in teaching it is meant to help achieve, as well as some design choices that were made along the way. The third chapter describes the implementation of the software in detail, going over third party software that has been used, as well as the application's main components, and finally a short guide on how to use the application, as well as on how to add new compilers to the system, allowing the user to compile their code in additional languages. The fourth chapter briefly analyzes some factors that influence performance, especially in a concurrent setting. The fifth and final chapters sketches some directions for future work that might improve the application and extend its use cases.

1.1 Motivation

Some background of how the project came to life, and a brief introduction into the established e-learning ecosystem it will be integrated into.

Active engagement with the teaching material is a tried and tested way of learning and understanding new concepts and oftentimes more effective than passively listening to a lecture [2]. However, the degree of engagement in large class lectures can often leave something to be desired.

Backstage 2 is an online platform for collaborative learning currently in developement by the PMS teaching and research unit. One of the goals of this platform is to further student engagement throughout the learning process.
The software project presented in this thesis is the Concurrent Code Compile Unit "co[co]^{2}nut" (*coco2* in the following). It integrates into the Backstage platform as an interactive code compile unit, to be used by students as well as teachers in and out of class. More information about specific use cases can be found in Section 1.2.

Particularly the use case of a large number of students solving an interactive coding exercise in class, and therefore potentially compiling their code all at the same time, poses some difficulties in a concurrent software setting. These difficulties and how coco2 approaches them are discussed in Section 2.2.
In a teaching context, using an online compile service can have benefits over simply installing the compiler on one's own system; for a very inexperienced student, the installation of compilers and runtime environments might already present a considerable hurdle. The online service and its integration into Backstage also allow for different kinds of data to be collected and used in learning analytics.

It should be mentioned as well that a software project for this purpose has already been developed in the context of a master thesis and has been used in teaching by PMS. However, it has some problems regarding the implementation details of concurrency as well as the general usability and extensibility. This version is based on *Compilebox*.[1] A more thorough comparison of the two versions is given in Section **??**.
This first version is simply called *coconut* and will be referred to *coco1* in this thesis for the sake of clarity.

1.2 Objective and Scope

An overview of possible use case scenarios for the software, specifications on what the input and output each symbolises, and some brief notes on what the software does not *do.*

The software offers an interface for sending a *compile request* from a client to a server and retrieving a *result*.
It should be noted that whenever we talk about "compile request", assume that we gen-

[1] https://github.com/remoteinterview/compilebox

erally mean by that a request to perform any interpretation or compilation on the source code, followed by the execution of the compiled or interpreted program. For example, for compiled languages like Haskell, if we say ”compile request”, we actually mean ”compile-and-then-run-request”. For languages that are not compiled, we mean ”interpret-and-run-request”.

So, for example for the Haskell request:

```
main = putStrLn "hello world!"
```

one would expect the result:

```
hello world!
```

1.2.1 Input and output semantics

The input is a *request*, consisting of source code to be compiled and run and the desired (programming or formal) language. As many programming languages offer or even heavily encourage[2] the source code to be organised into multiple files, *coco2* also supports this. Any command line arguments that are supposed to be parsed by the compiled program can also be specified. Information on how exactly this input should be formatted can be found in Section 3.3.

The output is a *result* of the compilation and execution of the source code. It is also encoded into the result whether or not the compilation and execution was successful or not. In any case, the output message is divided into an *output message* and an *error message*, representing what the compilation and execution would have written onto `stdout`, respectively `stderr`.

1.2.2 Use cases and direction

In conjunction with the e-learning platform Backstage 2, coco2 can be used for students and teachers to complete and review exercises in programming or pertaining to other formal languages such as predicate logic or turing machine code.

Backstage 2, as of now, has two main components, one of which is *Backstage courses*. Functionally, this is a web interface where slides and complementary material for lectures are put together by lecturers, hosted online during the time of the lecture, and can then be viewed and engaged in by the students. One example for supplementary material are *quizzes* run synchronously by the lecturer, i.e. all students are expected to input their answer into the system over a small time frame. Quizzes can for example be multiple choice questions, where statistics about the rate of correct and wrong answers are collected and can be shown and discussed during the lecture.

[2] Like Java, where each class has to be put in its own file.

An interesting application of coco2 is a Backstage quiz that is based upon a coding exercise the students have to complete during a lecture. With coco2, (and the online code editor *Knoala* that is integrated within Backstage 2) students can compile and test their code online. Also, by supplying some unit tests with the exercise, the code submissions could immediately be tested, and stats akin to the ones for the multiple choice quizzes, e.g. rate of students whose submitted code passes all unit tests etc., could be generated and shown during the lecture. This specific use case has implications for performance requirements, as discussed in Chapter 4.

Backstage's other main component is *Backstage projects*, a project based platform for the organization and scheduling of teaching materials and assignments. Students can also host and manage their source code for group software development projects. coco2 adds the feature to compile and execute program code directly on the platform, opening up many possibilities: The online compile service can be used for students during their home exercises to test out their code, providing them with useful feedback. There can be restrictions on assignments as well, tied to the submitted code being able to compile successfully or pass unit tests, decreasing the teachers' work load of reviewing the assignments.

1.2.3 Scope and limits

Even though coco2 can be used as described above, it should be noted that the software does not provide any functionality to test or analyze the compiled source code in any way. Any heuristics need to be applied by use of unit tests that are attached to the submitted source code or by external software.

It further does not provide a code editor or any other user interface. Backstage 2 provides the online code editor Knoala which is based on the Ace Editor[3] and can be used together with coco2.

[3] https://ace.c9.io/

CHAPTER 2

Conception

This chapter provides some background on how the project was conceptualised. Its presumed didactical benefits are elaborated. Architectural considerations are given for where the project model differs from its successor, *coco1*.

2.1 Teaching Objectives

coconut2 *provides students with an accessible, automated way of receiving pertinent feedback for completed programming exercises.*

Giving effective feedback. In the way that teaching of students in higher education computer science courses is usually conducted, one step of the process is often providing the students with feedback on some of their work. The goal of feedback is to further the student's understanding of the material, mainly by making them aware of possible errors in their process.

In order to provide this feedback, the quality of the students' work needs to be assessed. Often, this is done by a human proficient in the topic of the work, such as a professor or a tutor. For certain kinds of submitted work, it is also possible to generate feedback automatically by a computer system. This automated feedback generation comes with both advantages and drawbacks. To fully discuss the comparison of automated feedback generation and feedback by humans would be widely out of scope for this thesis; instead, we briefly illustrate some of the advantages that automated feedback generation comes with in certain circumstances.

Incorporation of feedback into the work is generally easier for the student, if the feedback is given *immediately* (i.e. a few minutes or less after the work is submitted), while the student sill has the content of the work and its context present in their mind, and doesn't need to recall it from memory. In some cases, especially if the feedback is given regarding the processing of a task, immediate feedback can be observed to be more effective for learning. [1, p. 98]

In order for immediate feedback to be possible, a direct way of communication between student and teacher needs to be maintained. Realistically, for feedback given by humans, this means that immediate feedback can only be obtained at certain times and in certain settings, e.g. in a tutorial session. When the feedback is automatically generated, it is evidently much easier to make this feedback available to the student immediately after handing in their work, as long as the feedback generation process happens reasonably quick.

Personal feedback, i.e. feedback pertaining to the person completing the task themselves, is often observed to be a very ineffective type of feedback, and personal statements (even if they are positive) in addition to material feedback can even deteriorate the accompanying material feedback's effectiveness. [1, p. 91]
A human giving feedback might be tempted to add a personal comment to their feedback, even accidentally or unconsciously. For feedback that is automatically generated, it is obvious that personal feedback is easy to avoid.

Compilers as educational tools. Programming exercises are prevalent in the Computer Science curriculum: Obviously, they are helpful for learning a new programming language, but even if the goal is to understand a new computational concept, doing an

example implementation can be very beneficial.

For such coding exercises, it is not necessary to create new software for automatic feedback generation, because one already exists: For every conventional programming language, parsers and compilers exist. Parsers can find syntactic errors, and simply comparing one's expected result with the actual result of a compiled and executed program can provide insight into possible semantic errors.
The difficulty merely lies in effectively using the provided tools of the programming language: If a learner has the ability to compile their code, and is provided with unit and integration tests that are designed to point out common mistakes, they are effectively able to generate their own feedback.

This is where the collaborative online learning platform Backstage and *coco2* come in. While setting up a developing environment on one's own system is certainly necessary to effectively work on own projects. However, attaining proficiency with a programming language's sysem environment is arguably a learning goal in its own right. For absolute beginners, it might be beneficial to provide a web-based environment, thereby removing this additional hurdle.

There might also be people who, for whatever reason, might not be able or willing to install a compiler for a particular language on their own system: Shared computers, hardware constraints and compatibility issues are some examples. Compilers can be a powerful learning tool, and coco2 helps making them more accessible.

From a teaching standpoint, using Backstage and coco2 in favour of native compilers on students' own system also comes with advantages. By using the web based platform, supplying unit tests is made easy. Backstage also provides the possibility to gather data about the learners' mistakes, which can then be incorporated into the exercises and test cases for future iterations of the course.

2.2 Design Choices

Investigation into the behaviour of coconut v.1 (coco1) *and consideration of specific use cases led to the discovery of some technical challenges that arise in a concurrent setting.*

The already existing *coco1*, while generally solving the problem at hand and providing the same basic functionality that is described in 1.2, does not behave in the desired way under certain circumstances.

Cleanup of the Docker Containers. *coco1* creates a new Docker Container for each compile request. This Docker Container is basically a virtual machine (the Docker terminology is more thoroughly discussed in 3.1) that is persisted on the hard drive. However, *coco1* does not adequately handle the removal of these containers, leading to excessive usage of disk space after a while. The disposal of the Docker containers is part of *coco2*'s

Docker routine.

Timeout scheduling. As the compiled code cannot universally be expected to terminate on its own in finite time, it is necessary to enforce some kind of timeout scheduling. Because the question of whether or not a program will terminate is undecidable, this means that if the execution of the compiled code has not ended after a certain amount of time, it has to be terminated from the outside.

However, concurrent execution of a large number of processes with fixed timeout scheduling can lead to unjustified termination. In fact, it is easy to see that the average time $\widetilde{t}$ it takes to compute a request in a parallelised multi-core environment is dependent on the number of cores c, the number of requests n that are concurrently processed, and the time it takes to process a single request on a single core t, and further that $\widetilde{t}$ increases in relation to n.

$$\widetilde{t} = n \cdot \frac{t}{c}$$

Therefore, $\tilde{t}$ will exceed any fixed timeout limit if there are a large number of processes in the system at the same time. This means that if the number of processes in the system is large enough, every single process will be forcefully terminated before it can be fully executed.

Especially for use in large class lectures, scalability with regard to the number of requests coming in at the same time is important, and this problem needed to be addressed. *coco2* solves this problem by restricting the number of virtualised processes that run at the same time, and maintaining a queue for additional requests. This process is described in more detail in Section 3.2.

CHAPTER 3

Project Realisation

This chapter outlines the technical approach of the development, going over third party frameworks and libraries, as well as the implementation model and a quick usage reference.

3.1 Third Party Software Used

Some third party frameworks and libraries were used in development. It is discussed how and why they were used. Specific terminology that might be unfamiliar to the reader is explained.

3.1.1 Docker

Docker basics. Docker is an open-source software for virtualising applications in isolated units, called *containers*. Docker provided means for maintaining *images*, in that way, configurations for running such containers can be reused. Many base configurations that come with runtime-environments and compilers for different languages can be found in the official Docker repositories.[1]

Docker containers can be used to run applications inside virtualised environments, thereby ensuring security of the host system. For an online compile service, this security is crucial, as compilation and execution of user code in turing-complete, file-IO-proficient languages pose a risk to the integrity of the system.

Compared to conventional virtual machines, Docker containers are relatively lightweight; they do not introduce a full operating system to the machine but instead run directly on the Linux kernel, while still providing security by strictly separating kernel name spaces and network stacks.[2]

A Docker container can be started from the terminal via the `run` command, for example:

```
$ docker run ubuntu echo "hello world"
```

In order to further customise the images, one can create a Dockerfile, where a host mount volume, a working directory, and multiple commands to be executed can be specified. Usually, specifications for new Docker images will use existing images as a baseline:

```
/* Dockerfile.txt */
# base your image on ubuntu
FROM ubuntu
# install additional packages
RUN apt-get install curl
# run an application
CMD curl "http://haskell.org/hoogle"
```

From this Dockerfile, a new Docker image can be crated, using the `build`-command, and then run:

```
$ docker build -f files/Dockerfile.txt -t fancy_new_image
$ docker run fancy_new_image
```

It is possible to *mount* a directory of the host file system to a directory of the Docker container's file system, making the contents of the directory accessible to the Docker con-

[1] https://hub.docker.com
[2] https://docs.docker.com/engine/security/security/

tainer, and persisting any artifacts created by the Docker container inside this directory, even after the container stops.

```
$ docker run -v /files/mountpoint fancy_new_image
```

For further reference, see the Docker documentation.[3]

Docker Client and Docker Daemon The Docker Client is the process that allows the user to input commands (such as `docker run`). These commands are then translated into http API calls by the Docker Client, and relayed to the Docker Daemon, which handles communication with the host operating systems and creation of mounted directories.

Integration into the project. *coconut2* uses a designated Docker image for each supported language. Preferably, Docker images from the official Docker repositories were chosen, as those are likely to be kept up-to-date by the Docker team.

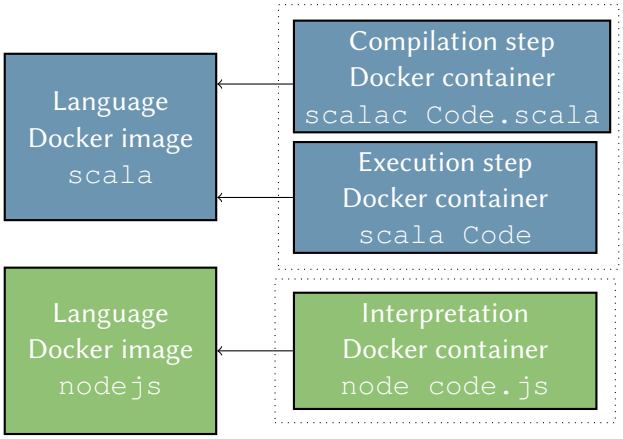


Figure 3.1: Relation between compile requests, docker containers, docker images. A docker container of a certain colour represents an instance of the docker image of the same colour. Dotted rectangles represent one compile request.

It would be natural to simply run a Docker container for each compile request, based on the image of the language the user wants to compile their code in. However, due to complications with the Docker Scala API, it was not possible to relay multiple commands to be executed in one container without creating a designated Docker file, i.e. building a designated Docker image. As building a Docker image is a relatively time consuming process, this idea was discarded.

Instead, each command is executed in its own Docker container. Data is exchanged between these containers by mounting them to a place in the host file system. This is also where the user code is transferred into the container.
This abstraction could also enable sequencing of processing steps that do not depend on

[3] https://docs.docker.com/

the same Docker image, i.e. that are executed in different run time environments, which might be a direction for future work.

3.1.2 MongoDB

MongoDB is an open source database program that uses JSON-like documents with schemas, meaning it is a NoSQL or non-relational database. For reference, see the MongoDB documentation.[4]
Compile responses are stored inside a MongoDB database together with their request data and metadata. Because this request and user data can later be used in learning analytics, it is stored persistently.
MongoDB in particular was chosen because it ensures that most recently updated items are kept in working memory, so that it can also be used to effectively fetch the result to a certain user's requested computation shortly after this computation was completed.

3.2 Implementation

An overview over coconut2*'s implementation model and its main components, their functionality and the way they interact with each other.*

A compile request enters the system via the *Server*'s REST API. The actual compilation and execution of the user code is virtualised by use of Docker containers. The *DockerBeetles* together with their *DockerBeetleKeeper* serve as intermediary agents between the REST API and the creation and running of these containers.
DockerBeetles are independent and stateless objects that handle the creation and destruction of Docker containers. One DockerBeetle represents one compilation request. After a Beetle is done with its work, it writes its result into the *CocoLog*, associated with some meta data.
The DockerBeetleKeeper is the central control unit for the DockerBeetles. It maintains a queue of request from which the Beetles are generated and synchronises access to critical system variables.
A simplified illustration of the system architecture can be viewed in Figure 3.2.

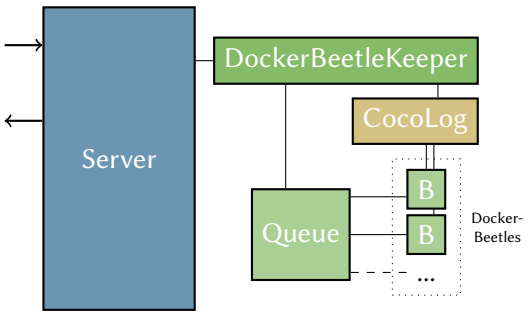


Figure 3.2: An overview of the software's main components.

[4] https://docs.mongodb.com/

An incoming compile request, as discussed in Section 1.2, consists of the serialised *source code files* (one of them marked as the ”main” file, or entry point to the program), as well as the *programming or formal language* the code is to be interpreted as. This information is the *request data*. The Server also collects some *request meta data* information that is mostly pertinent to the Backstage-specific context from which the compile request originates. Together, request data and request meta data make up a *CocoRequest* object.

3.2.1 DockerBeetleKeeper

The DockerBeetleKeeper is a case object. Its main jobs are to maintain the compile request queue and to ensure that no more than the allowed number of beetles are running at the same time. This maximum number of concurrently alive beetles is derived from the number of CPU threads (the need for this restriction is discussed in Section ??).

```
1  def request(req: CocoRequest, time_stamp: Long): String = synchronized {
2    val id = DockerBeetleID(
3               time_stamp,
4               Fun.unique_number
5               )
6    if (beetles_running >= MAX_ALIVE_BEETLES) {
7      request_queue.enqueue((req, id))
8    } else {
9      beetles_running = beetles_running+1
10     compile(req, id)
11   }
12   id.toString
13 }
```

Code Block 3.3: DockerBeetleKeeper’s method that handles incoming requests.

Synchronising the queue. Once a new CocoRequest enters the system, it is relayed to the DockerBeetleKeeper’s method `request` (see Code Block 3.4), which pairs it with a *DockerBeetleID*. This ID’s use is twofold: It serves as a unique identification number and encodes a time stamp of when the request was orignially received. From there, this pair is either enqueued into the request queue, or, if there are less beetles currently running than allowed, the `compile` method is instead called right away, where a new DockerBeetle is created and compilation is started.
The `request` method is *synchronised* (simply by wrapping it in Scala’s this.synchronized. This is because by accessing the semaphoric `beetles_running` variable (l. 6-9 in Fig. 3.4), the method body qualifies as a *critical section*.

Handling a compile request. In the `compile` method, an attempt is made to turn the request data into a DockerBeetle. This transformation will fail if the specified *language* is not known to the system. (Information about how new languages are added can be found in Section 3.3.) The result of the `compile` method is a *RunResult* wrapped inside a scala Future,[5] enabling the concurrent computation of multiple compile requests.

[5] https://docs.scala-lang.org/overviews/core/futures.html

```
1  def compile(req: CocoRequest, id: DockerBeetleID): Future[RunResult] =
2    req.data match {
3      case CocoRequestData(mainfilename, files, arg, language) =>
4          cocolibrary(language) match {
5              case Some(cocoLanguage) =>
6                  DockerBeetle(mainfilename, files, arg,
7                     cocoLanguage, id, req.meta)()
8              case None =>
9                  Future(RunFailure("", "sorry I do not speak " +
10                 language, req.meta, System.currentTimeMillis(),
11                 System.currentTimeMillis()))
12     }
13   }
```

Code Block 3.4: DockerBeetleKeeper's method that creates new DockerBeetles.

3.2.2 DockerBeetles

A DockerBeetle object is the represents one compile request in the system. The DockerBeetle controls creation and termination of the Docker containers where the user code compilation and execution takes place: This includes setting up the containers' working environment (i.e. creating and deleting the mount directory on the host system). It also passes compilation and execution results to the CocoLog, once execution is completed. When all of this is finished, it alerts the DockerBeetleKeeper to dequeue the next waiting request, thereby ensuring livelihood of the system.

```
1  case class DockerBeetle(
2                       mainfile: String,
3                       files: List[CocoFile],
4                       arg: String,
5                       language: CocoLanguage,
6                       id: DockerBeetleID,
7                       metaData: CocoRequestMeta,
8                       HOSTWORKDIR: String = Main.config.dockerplayground
9                      ) { /* ... */ }
```

Code Block 3.5: DockerBeetle's constructor.

Creation of new DockerBeetles. A DockerBeetle is created with all of the data that makes up a CocoRequest (files, language, arguments, meta data) as well as the generated ID and the working directory its docker containers will be mounted to (see Code Block 3.5). When it is created in the DockerBeetleKeeper's `compile` method (l. 6-7, Code Block 3.3), its `apply` method is immediately called as well. This method represents the complete DockerBeetle life-cycle, based on chained-together scala Futures. The DockerBeetle is therefore functionally stateless, meaning that it has no mutable state that is visible from the outside.

Life-cycle of the DockerBeetle. The `apply` method (Code Block 3.6) mainly consists of the aforementioned strung together Future wrapped computations. The basic procedure

consists of creating the mount directory, writing all of the input source files to this directory, creating and running the docker containers, and finally deleting the mount directory recursively, logging the result and informing the DockerBeetleKeeper that the next waiting request can be dequeued and "beetled".
The methods `create_mount`, `write_file` and `delete_recursively` called in ln.2, ln.4 and ln.10 respectively are merely file-IO handling methods that are based on `java.io`.
The `run` method (called in ln.7) handles the sequenced creation and destruction of the

```
1  def apply(): Future[RunResult] = {
2    create_mount()
3      flatMap { _ =>
4        val file_writings = files.map(x => write_file(HOSTWORKDIR + UNIQUENAME, x))
5        Future.sequence(file_writings)
6    } flatMap { _ =>
7        run_list(docker_commands)
8    } andThen {
9        case _ =>
10         deleteRecursively(work_dir)
11   } andThen {
12       case Success(runres) =>
13         log_and_dequeue_next(runres)
14       case Failure(e: TimeoutException) =>
15         log_and_dequeue_next(RunFailure("", s"Timeout after
               ${Main.config.timeout} seconds.", metaData, id.birthday,
               System.currentTimeMillis()))
16       case Failure(e: Exception) =>
17         log_and_dequeue_next(RunFailure("", s"There was an exception other than
               a timeout: ${e.getMessage}.", metaData, id.birthday,
               System.currentTimeMillis()))
18   }
19 }
```

Code Block 3.6: DockerBeetle's apply method.

appropriate docker containers and will be described in detail a bit further down.
The method `log_and_dequeue_next` (called in lns.13, 15, 17) calls a synchronised procedure in the DockerBeetleKeeper, making sure that access to the `beetles_running` variable and the writing to the log are synchronised.

Building and running the Docker Containers. DockerBeetle's `run_list` and `run` methods handle the synchronised execution of the relevant shell commands that create (and afterwards, dispose of) the docker containers for each compilation step. Every compilation step is executed inside its own Docker container.
These Docker containers are created with respect to a *mount directory*, an *image* and a

```
1 s"docker run --name $UNIQUENAME$i -w /app -v $work_dir_path:/app $image $comm"
```

Code Block 3.7: Interpolated String that assembles Docker shell commands.

command. The mount directory is the directory on the host where source code files and compiled files are stored between container runs. The image corresponds to the selected

language the code is to be compiled in. The command is the command that is executed inside the Docker container, and these are for example commands that run the compiler for the desired language, (like `ghc --make filename.hs`) or execute the compiled user program (like `./filename` or `java filename.java`). They also receive a name that is composed of the unique name that is given to this particular DockerBeetle, corresponding to the request, and a number that indicates what step of the compilation sequence it represents.

```
1  // run a single command. retrieve a run result in a future
2  def run(command: String, number: Int): Future[RunResult] = {
3
4    // logging the results from stdout and stderr
5    var err: ArrayBuffer[String] = ArrayBuffer.empty[String]
6    var ok: ArrayBuffer[String] = ArrayBuffer.empty[String]
7    val logger: ProcessLogger = ProcessLogger(str => {ok += str}, str => {err +=
         str})
8
9    val process = command.run(logger) // start asynchronously
10   val q = Future(Await.result(Future(process.exitValue()), Main.config.timeout))
11   q.map {
12     // encodes success or failure depending on process exit code
13     case 0 => RunSuccess(ok.mkString("\n"), err.mkString("\n"), metaData,
           id.birthday, System.currentTimeMillis())
14     case _ => RunFailure(ok.mkString("\n"), err.mkString("\n"), metaData,
           id.birthday, System.currentTimeMillis())
15   } andThen {
16     case Success(foo) =>
17       // remove the finished container
18       s"docker rm $UNIQUENAME$number".!
19     case Failure(e: TimeoutException) =>
20       // stop running container, then remove it
21       val r = s"docker stop -t 0 $UNIQUENAME$number".run()
22       Future(r.exitValue()) onComplete { _ =>
23         s"docker rm $UNIQUENAME$number".run()
24       }
25   }
26 }
```

Code Block 3.8: DockerBeetle's run method.

Temporal synchronization. To synchronise the execution of the compilation steps, (i.e. making sure the compilation of the source code happens before the execution) the `run_list` method applies a recursion based approach of stringing together scala Futures. It ensures that the compilation steps occur in the correct order, and that each step is only performed if the preceeding step was successful (e.g. no attempt will be made to execute a compiled program if the compilation has not been successfully completed).
The `run` method (shown in Code Block 3.8) represents the creation and disposing of of a single Docker container.
Its parameters, a String and a number, represent a specific Docker command and its index in the sequence of compilation steps (e.g. the command for compiling the source code might have the index 0, then the execution of the compiled program would have the index 1).
These indices are relevant in order to distinguish the two or more docker containers that

are created in the process: Distinguishing these containers allows for a new container to be created before the last one has been completely disposed of, enabling more efficient parallelization.
The `scala.sys.process` package includes an implicit conversion from String to the trait `ProcessBuilder`, giving it a `run` method, (called in Code Clock 3.8 ln. 9) which then creates a new process by interpreting the string as a Linux shell command, and executing it concurrently.

Data synchronization. The subsequent creation of multiple Docker containers pertaining to a single request also necessitates the synchronization of access to and persistence of some data. This is achieved by creating a directory on the host system which each of the subsequent containers are mounted to. After the final container terminates, this directory is removed from the host. (See the IO-related method calls in Code Block 3.6.)

Timeout scheduling. Each compilation and execution step of the user's code has to be terminated after a fixed amount of time, because there is no guarantee that a given user program will terminate on its own in a finite time interval. The system uses a configurable central timeout value for each of the steps, meaning that each step of the compilation and execution process is granted a fixed amount of time to reach termination on its own. After this interval, the compilation or execution step is interrupted and the subsequent steps are not performed.
This timeout scheduling is handled by the DockerBeetles. In Code Block 3.8 ln.10, the result of the computation of the process exit value wrapped in a Future is awaited, and wrapped in *another* Future; thereby executing its computation asynchronously, but with a fixed time limit.

```
1 case class RunSuccess(std_out: String, std_err: String, meta: CocoRequestMeta,
2       start_time: Long, end_time: Long) extends RunResult
3 case class RunFailure(std_out: String, std_err: String, meta: CocoRequestMeta,
4       start_time: Long, end_time: Long) extends RunResult
```

Code Block 3.9: RunResult's constructors

Logging and returning results. The output that is written to `stdout` and `stderr` inside each of the Docker containers is assumed to be the request's result. As of now, it is not possible to retrieve any additional files that were created by the user's compiled program. (See 5.2 for a discussion of this direction for future work.)
This is done by passing a `ProcessLogger` object to the called `run` method of the `ProcessBuilder` assumed string command. This logger simply appends anything piped to `stdout` and `stderr` respectively to a String variable (Code Block 3.8 lns. 5-7). After the Docker container terminates, the `RunResult` object is assembled (lns. 13-14). This can either be a `RunSuccess` or a `RunFailure`, depending on the exit code of the process run inside the Docker container. Both the error and the regular output channels are relayed (this might be useful in case there are some errors in an otherwise successful process,

like compiler warnings). The meta data and time stamp that have been collected from the request are also stored in this result object.
At this point the result also receives a new time stamp, signifying the time at which the request's processing has been finished. Thereby a `RunResult` object consists of the String piped by the process to `stdout`, the String piped by the process to `stderr`, meta data, a starting time stamp, and an ending time stamp (see Code Block 3.9).
The returned result is also added to the `CocoLog`, together with the request data and meta data.

3.2.3 Server

The coco2-server uses a REST-API in order to handle incoming compile requests. Requests are sent via a POST-request, to which a response that consists of a single string of characters is given immediately. This string of characters is called the *id* of the request (not to be confused with the internal DockerBeetleID). The result can be retrieved by a parametrised GET-request after compilation is done. This functionality all lies on the server's compile-route. (Figure 3.10.)

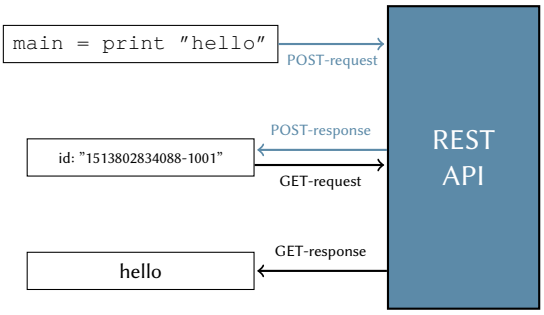


Figure 3.10: Illustration of how the REST API is used to send compile requests and retrieve the corresponding result, using an ID system.

The server is based on akka-http.[6] Apart from the aforementioned compile route, it provides a route called explain, where a documentation page can be viewed. Further, the server handles JSON-conversions and deals with the authentication.

Authentication and meta data collection. As of now, the authentication is directly tied to the meta data that is collected by the software and depends on the specific authentication system that Backstage 2 uses.[7] Modularising the authentication and meta data collection might be a goal for the future when another authentication system is meant to be used, or when there is different meta data to be collected.
Both for sending in requests and for retrieving results via the compile route, a valid authentication is needed. The explain route only has GET functionality and works without authentication.
The `CocoServer` decodes the authentication token using a JSON web token signing algorithm.

[6] https://doc.akka.io/docs/akka-http/current/
[7] https://www.keycloak.org/index.html

Relaying incoming requests to the DockerBeetle. When a POST request to the compile route comes in, it immediately receives a time stamp by the server, in order to measure it's time spent in the system. Afterwards it is relayed to the DockerBeetleKeeper for further processing.

Returning results to the user. For GET requests to the compile route, the server firstly checks if the id parameter has a valid form. If it does not, it responds with an error message. If the id is valid, it is looked up in the `CocoLog`. If a result has been logged under this id, it is returned to the user as a response. If no result has been logged for this particular id, a message informs the user how many requests are currently queued, and prompts them to try again shortly.

Cancelling requests. The user can also send a DELETE request parametrised with an id in order to cancel a previously sent request. If there is a request with this id currently queued, it is removed from the queue and will not be fulfilled. There is no way to cancel a request for which the compilation process has already started. The needed functionalities for interrupting a working DockerBeetle from the outside would be anti-idiomatic to its stateless and self contained nature and this architectural argument outweighs the benefit of saving a (rather small since bounded by a timeout value) amount of computation time.

3.3 Usage

A short guide on how to use the software, both for the server and the client side.

3.3.1 Setting up the server

Creating a configuration file. Under `src/main/resources/application.conf`, place a configuration file with the following keys. See Figure 3.11 for an Example.

`dockerplayground`	Path to the parent directory of the Docker container mounting directories. It is important that the Docker Daemon has writing permissions for this location.
`secret`	The shared secret key used by the JSON web token signing algorithm that is used to decode the authentication tokens.
`timeout`	This value determines the amount of time after which a running Docker container gets terminated. Scala's duration syntax can be used here.
`serverport`	The port on which the REST server listens on.
`resourcesdirectory`	Path where the custom language compilers are stored. (See Section 3.4 for further information.)
`hyperthread`	This value should be equal to the number of physical CPU cores the application has access to. This field can be omitted, in which case the Java Runtime estimates the number of available processors, however this will most likely lead to suboptimal performance on machines that support multi threading.

When the application is started via a compiled .jar file (usually created with `sbt assembly`) and not from the sbt console, it is important to call `sbt clean` after each change to the configuration file.

```
1 dockerplayground="/user/local/home/vdbrausegrund/dockerplayground/"
2 secret="arwZa3o7Y43Dz5e148nk"
3 timeout=20 seconds
4 serverport=8008
5 resourcesdirectory="/home/vdbrausegrund/customlangs/"
6 hyperthread=8
```

Figure 3.11: An example configuration file.

Starting the application. Running the application with `sbt run` or a similar command will automatically start a REST server listening on the port specified in the configuration file. When starting the application, any number of the following flags can be provided to modify the application's behaviour.

`-INIT` Initialization: Pulls all remote Docker images, builds all local Docker images. This flag should be set if the Docker images have been wiped recently.

`-DEBUG` Debug Mode: Authentication is no longer validated, however, there still needs to be an authentication header present, it will simply be regarded as valid, no matter its content.

`-DIE` Software can be shutdown from terminal. Useful for testing.

3.3.2 Sending in requests and retrieving results using the REST-API

The server uses a REST API to process requests. This section explains the format of the expected requests and the given responses.

```
{"mainfilename":"main.hs",
 "files": [{"name":"main.hs",
            "content":"main = print 07734"},
           {"name":"notmain.hs",
            "content":"main = print 80085"}],
 "arg":"a string with arguments",
 "language":"haskell"}
```

Figure 3.12: An example payload for a compile request.

Request format In order to request some user code to be compiled, a POST request should be sent to the server's compile route. The payload needs to be a JSON object with the following keys. (See Figure 3.12 for an example.)

`mainfilename`	String	Name of the main file, or entry point file.
`files`	[Object]	An Array of JSON objects that represent files, formatted as follows:
`name`	String	The name of the user code file
`content`	String	The content of the user code file
`arg`	String	The argument that should be supplied to the user program when it is executed. If no arguments are desired, this key still needs to be present. Its associated value should be the empty string.
`language`	String	The language in which the user code should be compiled or interpreted.

Besides the described payload, and a header declaring the content type, it is required that a valid authentication header is present. This header should be of the form:

```
Authorization: Bearer <Authentication String>
```

A list of the available languages is displayed as part the documentation page of the server. Any of the aliases that are listed there for a given language can be used as the `language` key of a request.

Response and result format The immediate response to a POST compile request will contain a JSON object with only two keys: `id` and `status`. (With the latter being either "running" or "queued".)
If a GET request is sent to the compile route, with this ID as a route parameter, the response will again be a JSON object. If the associated compilation has not yet been completed, it will contain the same two keys as the response to the POST request: `id` and `status`.
If the compilation has been completed, the `state` will be "finished", and the JSON response object will additionally contain a key `result`, which in turn contains the following keys:

`success`	Boolean	Whether or not the compilation was deemed successful. If either compilation or execution of the code yields an exit code that is not 0, this value is false, otherwise true.
`stdout`	String	The output of all compilation and execution steps that were piped to stdout, concatenated together into one string.
`stderr`	String	The error and warning messages of all compilation and execution steps that were piped to stderr, concatenated together into one string.
`start`	Integer	Time at which the request entered the system, in the epoch format.[8]
`end`	Integer	Finishing time of the last compilation or execution step, in the epoch format.

[8] https://www.unixtimestamp.com/index.php.

3.4 Adding new languages

The presented software is a framework that is meant to be extended in different ways. One way of extending the framework that was specifically planned for when designing the application is the process of adding new languages that user code can be compiled in. Other kinds of extension might require more work, and can be read about in Chapter 5.
New languages can be added to the system by extending the `CocoLinguist` object. This object maintains the languages known to the system, represented by lists of `CocoLanguage` objects. By adding new elements to these lists, new languages are added to the system.

```
1    case class CocoLanguage(image: String,
2                            aliases: List[String],
3                            command_builders: List[CocoRequestData =>
                                 Command])
```

Figure 3.13: Representation of languages.

The `CocoLinguist` object maintains two lists of `CocoLanguages`, one for such languages where the Docker image is pulled from the remote Docker image repository (called `pre-built-languages`) and one for languages where the configuration for the Docker image, as well as the compiler, is stored on the local machine (this list is called `custom languages`).
In order to add a pre-built language, it is enough to create a corresponding `CocoLanguage` object. To add a language that is virtualized in a custom Docker image, you also need to add a Dockerfile (and any compilers and runtime environments you want to use) to the `custom_languages` directory inside the resources directory specified in the application's configuration file.

```
1    CocoLanguage(
2      image = "haskell",
3      aliases = List("hs"),
4        (data: CocoRequestData) => IndependentCommand(s"ghc --make
             ${data.mainfilename}"),
5        (data: CocoRequestData) =>
             FileDependentCommand(s"./${data.mainfilename.stripSuffix(".hs")}
             ${data.arg}", data.mainfilename.stripSuffix(".hs")))
6      )
```

Figure 3.14: Example `CocoLanguage` object for the language Haskell.

A `CocoLanguage` object is built by supplying the name of the image (in case of a pre-built language that is to be pulled from a remote repository, this name needs to be equal to the image name that is used in the remote repository, so that Docker can find it), any number of aliases you want to add (so that the language doesn't always have to be referenced by the remote image name, in case that name is undesired for some reason), and a list of command builders.

A command builder is a function that takes all of the request data as an input, and yields a `Command` object. A `Command` object can either be an `IndependentCommand` or a `FileDependantCommand`: Independent commands are always executed, file dependent commands are only executed if a certain file exists. This is a useful distinction, because there are instances in which certain steps of the compilation and execution process are dependent not only on the success of the preceding step, but on a specific artifact being created by the preceding step.
Consider as an example the `CocoLanguage` object that represents the Haskell language. (Code Block 3.14). The first command builder represents the compilation command. This is an independent command because at least one source code file must be present in any valid `CocoRequest`, meaning that there is always something to compile. The second command builder represents execution of the compiled code, which yields a file dependent command: There might be valid, compilable code, that does not yield any executable files (e.g. example any Haskell program that does not contain a main function).

CHAPTER 4

Performance Analysis

For a single request, *coco1* and *coco2* yield compilation and execution results in approximately the same time. coco2 produces slightly more overhead due to the need for two HTTP requests instead of one, and the use of a designated Docker container for each step. However, coco1 processes compile requests sequentially instead of in parallel, leading to poor scalability of the application, which makes it unsuitable for use in large class lectures. coco2 parallelises the processing of its requests, meaning that its performance can always be improved by allowing it to run on more CPU cores, as long as the bottleneck of writing data to the hard drive that emerges from the frequent creation and destruction of Docker containers is avoided by using a virtual RAM drive.

4.1 Performance Improvements with a RAM drive

Instructing Docker to write its images and containers into RAM instead of virtual memory can increase performance drastically by softening the I/O bound.

Language: Haskell (compiled and executed, two steps)
User Code Input: `main = print 11`

containers in RAM	mounts in RAM	# of requests at the same time 16	32	64	128
✗	✗	13	34	66	125
✗	✓	13	32	66	124
✓	✗	7	15	27	53
✓	✓	7	14	26	53

Language: node.js (interpreted, one step)
user code input: `console.log(11);`

containers in RAM	mounts in RAM	# of requests at the same time 16	32	64	128
✗	✗	10	15	31	62
✗	✓	10	15	31	62
✓	✗	2	5	9	19
✓	✓	2	5	9	19

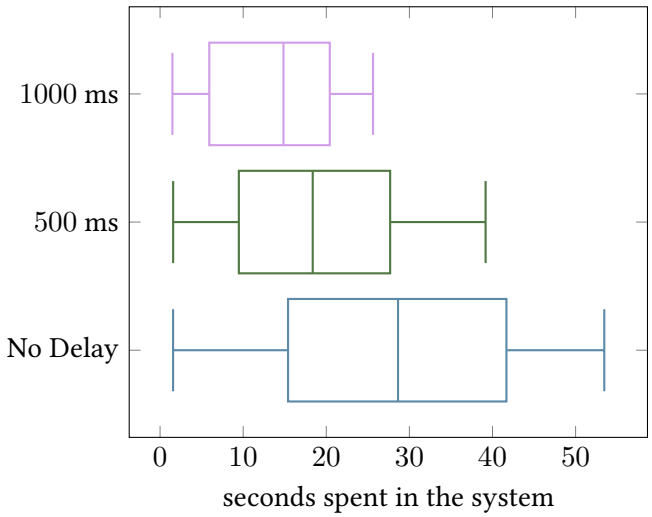
Figure 4.1: Average time spent in the system for user code compilation requests, in seconds, depending on whether or not the Docker Container itself or its host mount volume were mounted on a RAM drive.

When a Docker container is started, the software that is needed to run it (its operating system, system tools, libraries, and settings, as well as any runtime environments or compilers needed for user code compilation) is persisted to a designated place in the file system of the host system. Each individual docker container that is set up therefore necessitates a significant number write operations to the hard disk. This is evidently a bottle neck of the application, because disk operations cannot be parallelised, and leads to a classic I/O bound of the applications performance.

This problem can be mitigated by using a RAM drive as the location where the Docker containers are stored. The volatility of such a RAM drive does not pose a problem, because the Docker containers are only needed for the time it takes to complete a single request. Using a RAM drive for the container's storage location did, in fact, yield a performance increase of about a factor of $2 - 3$ (see Figure 4.1). Placing the host mount volume of the Docker containers (recall, that is where the user code is transmitted into the containers' file system) on the RAM drive was also tested and did not yield a significant improvement, probably due to very small file sizes.
These tests were performed on a machine with two CPU cores and 4 GB of RAM, using a RAM drive 2 GB in size. With more CPU cores, higher parallelisation could be achieved, and an even bigger performance increase would be expected.

4.2 Performance Dependency on Frequency of Incoming Requests

Data shows that even a short delay between incoming requests results in a significantly reduced average waiting time for each request, compared to requests coming in in bulk at the exact same time.



Language: Haskell (compiled and executed, two steps)
User Code Input: `main = print 11`

Figure 4.2: A box plot of the time that was spent in the system by 64 requests that were sent in with varying intermittent delays.

The data acquired in 4.1 is based on different numbers of multiple requests entering the system at the exact same time. However, in a realistic setting, even for a large number of users, especially in the teaching context, one would expect at least some delay between the incoming requests. Even for a large number of students solving the same exercise in

the same time frame, the exact times at which they would submit their solutions would be different.

As can be seen in Figure 4.2, even a short delay between incoming requests lowers each request's time spent in the system significantly. Specifically, a delay as little as 500 ms decreased the median time spent in the system by nearly a full second.

CHAPTER 5

Conclusion and Future Work

coconut2 allows for concurrent user code compilation, and its scalability makes it suitable for use in large class lectures.
This chapter discusses directions for future work, including some additional features that could be useful in learning, and the possible challenges that might arise during the development of these features.

5.1 Ways to Further Optimise Performance

Use of pre-compiled artifacts. Especially in the context of teaching, code compilation could encompass redundant tasks. Programming exercises may be supplied with unit tests, these unit tests do not change between students, and therefore do not need to be recompiled for every student attempting to solve the exercise.
There are many languages where this approach would be fathomable, in Java, for instance, compiled `.class` files of the same name are interchangeable.
This would necessitate the possibility of a differentiation between main user code files, and precombiled files provided by A meaningful abstraction for this would need to be conceptualized, and implemented. This would also most likely entail changes to the REST API.

Non-virtualised execution. Especially in the context of formal and didactical languages, there are some programs that do not necessarily need to be virtualised for security. For example, the virtual Turing Machine language does not contain any functionality for writing to the filesystem or interfering with the system at all. The potential "danger" of non-terminating programs can be mitigated by terminating the created process after the given timeout threshold, a system which is already implemented.
Therefore, for these languages, one could conceptualise a `NonDockerBeetle`, similar to the `DockerBeetle`, that directly executes the compilation and execution commands on the machine itself, without using Docker as an intermediary at all. One possible challenge to this approach is the correct set-up of the working directory, as this is currently mostly handled by Docker. This could lead to better performance.

5.2 Adding new features

Exploring other possible use cases of coconut2 that could be covered by changing the software and adding new features.

Rich output. Currently, coco2 only supports strings of text as an output format. While code that creates additional output, such a rendered image file, can be compiled and executed, there is currently no way to transmit these files back to the user.
It would be possible to persist any created files on the host system simply by omitting the deletion step. As the directories created by a specific compilation request is already tied to that request by its identifier, retrieval would also be possible without much change to the software. However, transmitting files other than text with the response would necessitate rather big changes to the API. Also, the issue of data storage limit would need to be addressed.

Interactive REPL Many programming languages don't only support compiling a full executable program, but also offer an interactive REPL (Read–Eval–Print Loop). This is very useful for testing small scripts, and can especially aid in getting first grasp of the syntax and basic functionalities of a newly learned language. Evidently, REPLs are useful tools for learning and teaching.

For the extension of coco2 to support an interactive REPL, two approaches seem feasible. One possible approach, that would require the least change to the software, would be to create an entire new REPL session on the server, for every new user input. This way, no interactive connection between the client and the server would be necessary. However, this would make supporting the evaluation of any time-dependent expressions very difficult.
Another approach would be to maintain a connection between the client and the server, and sending the user's input to the same interactive REPL session, until the user decided to exit the REPL. This could be achieved, for example, by using web sockets. However, this would necessitate larger changes to the existing software, specifically the API.

Bibliography

[1] John Hattie and Helen Timperley, *The power of feedback*, Review of educational research **77** (2007), no. 1, 81–112.

[2] Michael Prince, *Does active learning work? a review of the research*, Journal of engineering education **93** (2004), no. 3, 223–231.