TRY REPY!

A WEB-BASED DEVELOPMENT AND EXECUTION ENVIRONMENT FOR RESTRICTED PYTHON

handed in by
Lukas Pühringer

BACHELOR THESIS

Branch of Study: Mediainformatics
Faculty of Computer Science at the University of Vienna

Advisors:
Albert Rafetseder, Bakk. (FH), MSc; Prof. Dr. K. Tutschku
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht.

Toulouse, October 14, 2011

Lukas Pühringer

Abstract

Try Repy is a web-based software development and execution environment for restricted Python, a programming language used in the Seattle Network Testbed. This document shows how Try Repy fits into current network research by discussing its practical and its theoretical backgrounds, namely Seattle/Internet testbeds and virtualization techniques. Additionally it lists Try Repy’s features and explains how they can be used for developing and evaluating network experiments in Seattle. Furthermore, it describes how Try Repy allows examining virtualization strategies, primarily considering isolation and scheduling. Rounding out the picture, this document reveals yet unexplored potentials of the presented software, by proposing some enhancements for Try Repy’s components.
Contents

1 Introduction 1
   1.1 Seattle 1
   1.2 Try Repy 2

2 Virtualization 3
   2.1 Sandboxes and Seattle 4
   2.2 Sandboxes and Try Repy 5

3 Design 6
   3.1 Used Technology and Deployment 7
      3.1.1 Repy and Seattle 7
      3.1.2 Web Interface 8
   3.2 Architecture 9
      3.2.1 Server 9
      3.2.2 Client 10
   3.3 Extensibility 10

4 Try Repy Use Guide 10
   4.1 Installation and Usage 11
   4.2 Web Interface 11
   4.3 Known Issues 13

5 Future Work 14
   5.1 Try Repy as Development Environment 14
      5.1.1 Threads 14
      5.1.2 TCP/UDP Listeners 15
      5.1.3 Preprocessing 15
      5.1.4 User Interface 15
   5.2 Try Repy as Research Subject 16

6 Resume 16

7 Acknowledgments 17
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Seattle components</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Try Rey Deployment</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Try Rey Server Architecture</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Try Rey JavaScript Architecture</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Try Rey Web Interface</td>
<td>21</td>
</tr>
</tbody>
</table>
1 Introduction

The key aspect of this document is the description of Try Repy, a web-based software development and execution environment for restricted Python, which was implemented in the course of a bachelor seminar in the year 2011 at the research group of Future Communication [2]. Repy [3], a restricted variant of Python, is the programming language used within the Seattle Network Testbed [4]. Given that Seattle is heavily used at the research group of Future Communication, for research as well as for teaching, the chief requirement for Try Repy was to develop an interface which facilitates the creation and evaluation of arbitrary network experiments. At the same time, the modular design of Try Repy provides a framework for future experiments with different virtualization mechanisms within the software itself.

This document is structured as followed. The Introduction will give an insight into the characteristics and capabilities of the Seattle Network Testbed and its language Repy [3]. This preface will pave the way to a presentation of Try Repy. Section 2 will discuss virtualization concepts as a general technique as well as in Seattle and Try Repy, which are both virtualized environments. In order to enable a better understanding of the presented software and facilitate further development, Try Repy’s design is described from a high-level point of view in section 3. The usage of Try Repy as a software developing environment will be treated in section 4 describing in detail the necessary installation steps and giving an overview of the web frontend’s features and peculiarities. Equally important, there will be a list of propositions for further developments of the presented software in 5 discussing known issues that exist and pleasant features that have not been implemented so far.

1.1 Seattle

Try Repy was built for and on top of Seattle [4], which is a community-driven network testbed, initially developed at the University of Washington. Typical use cases include network measurement, peer-to-peer applications [5], web measurement [6] and distributed hashtables [7]. The system is based on resource donations by users and institutions, and in consequence runs primarily on end-user systems. This requires contained virtual machines (VMs) to avoid compromisation of user system’s security and performance. The components of Seattle are depicted in Fig. 1. The most granular component in the Seattle testbed is the VM. The user, who runs experiments in the VM usually controls it through a clearinghouse [8], an entity in the testbed that accounts resource donations and accordingly grants access to vessels. At this moment, the only language to run experiments with is restricted Python – Repy [3]. A vessel can define restrictions concerning resource consumption and other privileges.

A vessel runs on a node, which is basically a computer, whose resources where
donated by a participant of the Seattle network. It is possible to host multiple vessels, managed by a hypervisor called the Seattle node manager on one computer. In order to execute experiments in a vessel, a user needs to acquire the vessel. Thereafter, the experimenter uses an experiment manager to execute the experiment on one or multiple previously acquired vessels. There are different command shell and scriptable service managers to interact with a vessel.

**Workflow of Experiment Deployment** In order to deploy an experiment on a Seattle vessel, the following steps are required:

1. download and install Seattle
2. write Repy code
3. save the code file to the local Seattle directory
4. acquire remote vessel(s)
5. connect to the remote vessel(s)
6. upload and run the Repy program
7. check the status of the executing program
8. download the output of the experiment

Acquiring remote resources (vessels) is done via a clearinghouse, e.g. SeattleGENI, whereas interaction with the remote resources requires a service manager, e.g. Seash. Try Repy ties in at this point, trying to facilitate the Repy development and experiment deployment process.

**1.2 Try Repy**

Try Repy is first and foremost a software development and execution environment, to write and run Repy experiments in a Seattle vessel. It is based on a server-client architecture, providing a web interface as development environment and a server as execution environment.

From a client perspective, there is the web interface, providing a source code editor together with other handy features that facilitate code development.

The server, running itself in a Seattle vessel, imposes a system to manage multiple contained sandboxes. Additionally, the server deals with the communication between the web interface and a user’s sandbox, i.e. it receives user-submitted code and delivers the output of the code execution back to the web interface.

How Try Repy is used to write and evaluate experiments is described in detail in section 4. Meanwhile, the important and inherent subject of virtualization is discussed in the correspondent section 2 revealing the already mentioned second
perspective on Try Repy, that is, its potential for exploring different virtualization strategies.

2 Virtualization

Virtualization is a chief subject regarding Try Repy. On the one hand, Try Repy itself is deployed in Seattle, which is a virtualized environment. On the other hand, Try Repy provides virtualized and safe environments to execute user submitted code therein. Such an environment is usually referred to as sandbox or virtual machine (VM) [7], which will be discussed in more detail later.

The term virtual refers to “the quality of effecting something without actually being that something” [11]. From a technological point of view, this means to abstract a portion of the given resources and present them to the resource-requesting entity as if it would access the real resource. On the lowest level of abstraction, we are talking about physically existent resources like CPU-power, memory, storage or a physical communication medium. As will be seen in Try Repy, virtualization can have an arbitrary amount of abstraction levels, where each level pretends to provide the actual resource that has been requested.

Virtualization is an ubiquitous topic in computer science and has basically existed ever since IBM introduced a system of pseudo-machines managed by a hypervisor to be used in their mainframes, in the 1960s. On these mainframes, only the hypervisor had direct access to the physical hardware, while the pseudo-machines got the resource provided by the hypervisor, neither knowing about each other, nor about the hypervisor. [12]

Why Virtualization? In [12], the following main reasons for virtualization are discussed: From an economical, and also from an ecological point of view, it is of course better to use the existing physical infrastructure instead of purchasing new one, especially when the existing infrastructure still offers unused capacity. Virtualization allows to fully exploit the given hardware by deploying several virtual entities in it.

Additionally, virtualization addresses security. As already mentioned, virtualized environments are often called sandboxes, because they basically form an enclosed safe environment that cannot immediately access and therefore can not compromise the host operating system below. If a sandbox crashes, it usually can be replaced easily by a new sandbox, or by a previously saved working instance of the sandbox, without any effect on the hosting system.

How does Virtualization work? Usually, the resource providing layer, i.e. a hypervisor, defines interfaces for the offered resources. Before calls to these interfaces
are passed on to the resources’ “real” interfaces, they can be arbitrarily manipu-
lated, according to the specifications of the virtualizing environment. The manip-
ulation refers to two major paradigms of virtualization – Isolation and Schedul-
ing.

**Isolation** Isolation in virtualization can be seen from two different points of view. On the one side, isolation should happen between concurrently existing virtualized entities. That is, no entity needs to know about the co-existence of another entity and least should they interfere with each other. On the other side, a virtualized environment should be isolated regarding a direct resource access in order to avoid compromisation of the hosting system, as discussed above.

**Scheduling** Scheduling in virtualization refers to sharing the available resources between the virtualized entities in a controlled manner. There is a wide range of different scheduling schemes, each depending on the desired results. Scheduling can be fairness-based or involve priorities. It is also possible to implement certain guarantees for the available resources. As already mentioned, it is possible to construct one virtualization layer over another one, to an almost arbitrary extent, limited only by physical capacities. In the same manner, scheduling can be applied on each layer. An exemplary performance analysis of a system of layered schedulers was done in [13]. Both isolation and scheduling issue will be illustrated now in more detail regarding the implementation of Seattle and Try Repy as examples.

### 2.1 Sandboxes and Seattle

This subsection gives a brief summary of Seattle’s sandbox design, which is described in detail in [7]. The core components of the Seattle sandbox are:

- a small self-contained sandbox kernel
- an encasement library
- security layers
- standard libraries
- the Repy programming language
- untrusted user code

The chief aim of the Seattle sandbox is to ensure that a security failure in the standard library code has a minimal security impact on the sandbox containment. This is achieved by keeping the privileged code – the sandbox kernel, which has access to the operating system – to a minimum regarding size as well as complexity, and deliberately placing a lot of sensitive portions of standard libraries in isolated
components. These isolated components have the same permissions as untrusted user code. Hence, a flaw in non-kernel code would not allow an attacker to escape the sandbox. As a consequence, changing or extending the libraries has no security impact on the kernel, which makes the system very robust and scalable.

The isolated components, which are layered on top of the kernel, are called security layers. Additionally Seattle introduces an encasement library, which monitors the security boundary between two consecutive security layers.

To load and execute the code of a security layer, the sandbox kernel uses virtual namespaces. Virtual namespaces provide a code safety validation function, which checks if the code is Repy-conform, and a function to evaluate the safe code. Security layers are instantiated on top of each other, always passing on a potentially manipulated set of capabilities to the next security layer. Each security layer can only use the capabilities granted by its instantiating security layer beneath. The last security layer on top of all other layers will contain the user code. Hence, in order to use its granted capabilities, the user code has to traverse the entire security layer stack. In addition to the capabilities, a capability contract is passed on instantiation of a security layer. This contract is used to check the valid usage of a security layer’s capabilities. The entire set of capabilities available in the kernel concern network functions, virtual namespace calls, file I/O calls, a lock object to deal with race conditions, functions to provide information about the environment, thread related calls and a call to return random bytes.

2.2 Sandboxes and Try Repy

Try Repy is built on top of Seattle. It uses Seattle’s context and virtual namespace object to build encapsulated sandboxes.

**Context**

The context in Seattle is a Python dictionary, which stores the entire Repy API and the shared variables, and all newly defined functions in a Repy program. Hence, when a Repy program tries to access a function or a shared variable, it does so via its context dictionary.

In Try Repy, a copy of the current context is stored at the very beginning of the program execution. This happens after the functions that are required by all sandboxes are defined, but before the system functions, i.e. the web server’s functions, are defined. Later, when a sandbox is created – this happens, when a client requests Try Repy for the first time – another copy of the previously stored context copy is passed to the sandbox. Consequently, every sandbox has its own context of API functions, shared variables and newly defined functions, which cannot interfere with any other sandbox’s context, nor with the system’s, respectively the web server’s context.

On sandbox creation, each sandbox partially overrides the API functions stored in its context. Thus, a user-submitted program does not actually call the API
function but an abstraction, which wraps the real API function. This is a basic instrument of virtualization. Wrapping the functions makes it possible to e.g. gather accounting information, perform scheduling and make sure that the function call of one sandbox does not interfere with the calls of any other.

Repy, in its current state of development, has neglected the question of scheduling, but has already implemented abstractions for some important function calls, like file and network operations. So far the abstractions are used for accounting and isolation purposes. For example, there is a set of abstractions for Repy’s file operations, which allows every sandbox to work in its own file namespace. That is, when a user accesses the files in a vessel, he cannot access (read/write/list) other users’ files.

Other abstractions are mainly used for thread accounting, because so far there is no other way to find out the status of the threads started by a user-submitted program. So Try Repy introduces a thread accounting system that registers, respectively deregisters, every started thread in the abstraction of the thread generating function.

And of course, there is an abstraction used to log the output of a user-submitted program.

**Virtual Namespace** The virtual namespace \[1\] is provided by the Repy API \[3\] and offers a function to evaluate a Repy program in a specified context (the vessel’s context on a Try Repy Sandbox context). Hence, when a user submits a piece of code, the code is evaluated in a newly generated virtual namespace, using the sandbox’s context.

### 3 Design

This section will give an overview of Try Repy’s design from a high-level perspective. The major documentation of the code is the code itself. Since Seattle-specific Codestyle and Documentation Guidelines \[15\] \[16\] were strictly applied in the course of developing Try Repy, the code and its comments should be easy to read.

Hence, this section will only explain in general the used technologies, their functionality, and where they are deployed in order to work together. Additionally, some ideas on how to extend the given architecture will be proposed.

Basically, Try Repy uses a server-client architecture. That is, it consists of a server listening for incoming HTTP requests in order to serve a web interface to an arbitrary amount of users. The first request of a user instantiates a new sandbox for this user. Each further request of the same user performs operations in the user’s sandbox.
3.1 Used Technology and Deployment

3.1.1 Repy and Seattle

As depicted in Fig. 2 and already explained in previous sections, Try Repy runs on a Seattle node in one of the node’s vessels. Since only Repy code can be executed in vessels, Try Repy, or at least all the server parts, are written in Repy [17] as well.

Web Server The main program of Try Repy is the web controller. It registers a listener for incoming HTTP requests in order to serve the web interface, it instantiates the sandboxes and allows communication between the web interface and the user’s sandbox.

Sandbox The sandbox is the place where a user Repy program is executed. The sandbox provides logging facilities, thread management and most importantly, it provides transparent abstractions of the Repy API in order to ensure isolation between multiple sandbox instances.

Seattle Standard Library Try Repy uses the following scripts of Seattle’s Standard Library [18]:

- httpserver.repy
- urllib.repy
- random.repy
- base64.repy

Httpserver and urllib facilitate the web controller’s work, random is used to randomly generate user IDs and base64 is used to encode filenames within Try Repy’s file abstraction.

VirtualNamespace Virtual namespaces [14] are a chief technology in Seattle, also provided by the Repy API [9]. A virtual namespace is a safety-checked environment for Repy code, which provides an evaluate method. The virtual namespace is the innermost entity of the sandbox, where the user-submitted code is actually evaluated.
File Abstraction  The file abstraction is just one of several abstractions of the Repy API used in Try Repy. The file abstraction is described explicitly, because of two reasons. On the one hand, it is a showcase for other abstractions, used in the sandbox in order to provide isolation between sandboxes. On the other hand, because the web controller itself also uses the file abstraction as a handy tool to manage the web files (HTML, CSS, JS).

Regarding isolation between sandboxes, the file abstraction introduces a user ID property for each file. That is, every sandbox, i.e., user, has its own file space, which no other sandbox has access to.

In vessels directory hierarchies are not possible by default. With Try Repy’s file abstraction directory hierarchies can be simulated. These logical hierarchies are used by the web controller upon an HTTP GET request. From a vessel’s perspective all files are still saved in one directory, but the file abstraction can match a requested file path with its file name in the vessel. This is very useful while developing, because all web files can be stored in a reasonable directory structure and reference each other considering this structure. When Try Repy is deployed, a script flattens out the directory hierarchy to make it vessel conform and encodes the file names in a matter, so the file abstraction can match the encoded file names with their original paths.

3.1.2 Web Interface

The web interface is the user’s interface to develop and evaluate Repy code. A detailed description of all its features is found in section 3.

The web interface is basically a collection of HTML, CSS and JavaScript files that lie in the Try Repy file namespace and are served to the user’s web browser by the web controller on HTTP GET request, using the Try Repy file abstraction. The JavaScript part of the web interface has three main functions:

- Navigation and Usability
- Communication with the web server via Ajax
- Incorporating the Ace Editor

Ace  Ace [19], a successor of the Mozilla Skywriter (Bespin) Project, is a JavaScript Library, that enables the incorporation of a standalone code editor on a webpage. It is used in many different projects, mainly within the Cloud9 IDE [20], which is a complete IDE running in a browser and living in the cloud.

In order to find out more about Ace, visit the links of the Ace or Cloud9, provided in the References list, or have a look at section 4 which describes the functionalities of Ace, that where adapted for Try Repy.
3.2 Architecture

This subsection will explain the previously introduced major components of Try Repy in more detail using UML diagrams. The following subsections will not describe all implemented functions and variables in detail, but will discuss those parts that are important for an understanding of the principles of Try Repy.

3.2.1 Server

As seen in Fig. [3] three main components were implemented on the server side.

**tr_webcontroller.repy** The web server provides global variables and several functions to dispatch HTTP requests coming from the web interface. Each HTTP request opens a TCP connection, which receives a separate thread and has access to the global variables and functions.

One of the global variables is a copy of the Repy context, to be passed to each sandbox on instantiation. The context, as mentioned earlier, is a dictionary with references to the Repy API. The second global variable is a dictionary, which stores all sandboxes that have been instantiated in the web server’s lifetime, under an according user ID. The third global variable is a lock to avoid race conditions, i.e. when multiple threads in the web controller try to access the dictionary of sandboxes.

Concerning the functions, the bigger part handles the different kinds of HTTP requests. According to the request, the web controller serves a file (HTML, CSS, JS), a user ID, the log of a sandbox, the evaluation of a code, or the current output of a code evaluation in order to provide real-time feedback to the user. The other functions are helper functions for the HTTP communication and the user management.

**tr_sandbox.repy** The sandbox is a class with different groups of attributes. Each sandbox has its own ID which is called user, and the copy of the Repy context, which makes it possible for the user-submitted code to access the Repy API. Furthermore, there are attributes for the log of a sandbox, for the output of a current user code execution and for the thread management.

When a sandbox is instantiated, the context_wrap() method is called. This method overrides the sandbox’s context with the abstractions for isolation and future scheduling purposes. There are different abstractions for different subjects, e.g. there are abstractions for network operations and there is the already mentioned file abstraction. These abstractions in turn wrap the real Repy API calls.
The file abstraction is a collection of wrapping functions, that extend the Repy API file operations, adding a user property and a namespace translation.

### 3.2.2 Client

On the client side, the dynamic part consists of the Ace JavaScript library and a managing Try Repy JavaScript file, depicted in Fig. [1].

**tr_repy.js** When the web server is loaded for the first time, the function `init_ace()` is called. This function handles the entire Ace incorporation. Another collection of functions handles the Ajax communication between client and server. While submitted code is being evaluated on the server, anew submission is blocked. For this purpose there is a register and a deregister function for the code submission key command, provided by Ace. Then there is a toggle function for navigation purposes and a function that makes HTML elements when clicked inserted to the editor. The remaining functions concern the file insertion feature, discussed in more detail in section [2].

### 3.3 Extensibility

The preceding paragraphs discussed the current implementation of Try Repy, but this is not necessarily the only way of using the implemented modules. The web interface could easily be exchanged or extended by another interface. The modular design, as seen in Fig. [2] would allow to use the same sandbox implementation providing another server instead of the web server that manages the sandboxes and calls the sandbox operations. A possible interface could be a telnet interface. At the same time, the given web interface with its managing web server does not need to necessarily use the Repy sandbox. It could also provide its interface to any other language’s sandbox.

### 4 Try Repy Use Guide

The following section describes the usage of Try Repy as a software development environment with built-in execution capabilities. The description includes the necessary steps to run the Try Repy web server in a local Seattle repository, as well as a guide listing all features provided by the Try Repy web interface. Refer to [3] for Seattle installation instructions.
4.1 Installation and Usage

As described in section [2], the Try Repy software consists of several Repy files, providing the web server, as well as a directory containing static web files (HTML, JavaScript, CSS). In order to run Try Repy, some conversions are required. The web controller script has to be preprocessed and turned into a complete Repy script. Additionally, the file names of the static web files have to be converted to work within Try Repy's namespace. The shell script `tr_build.sh` takes care of these conversions and copies all necessary files into the build directory.

$ cd /path/to/tryrepy/
$ ./tr_build.sh

Once `tr_build.sh` has been executed successfully, the shell script `tr_run.sh` copies the contents of the build directory to your local `seattle_repy` directory and registers the Try Repy web server on the specified port. Try Repy requires a restriction file, called `restrictions.tryrepy` to lie in the `seattle_repy` directory, permitting the usage of the serverport.

$ tr_run.sh /abspath/to/seattle_repy <serverport>

Webinterface available on:
http://<ip>:<port>

Navigate then to the address with a web browser of your choice and you will be served the Try Repy web interface. The client side of Try Repy was successfully tested with Firefox 6.0.2, Safari 5.1, Chrome 14.0.835.186 and Opera 11.50 on Mac OSX 10.6.8. It should work on other platforms as well.

4.2 Web Interface

Whenever the Try Repy web interface is called, a new sandbox is instantiated at the server. The interface is depicted in Fig. [5] and its features are discussed subsequently.

Editor The main part of Try Repy’s web interface is the integrated Ace editor. The editor is always displayed on the left side of the web frontend while the right side changes upon click in the navigation bar. The editor is used to write Repy code to it. By clicking the button `submit code` beneath the editor, the contents of the editor window are sent to the Try Repy server, and hence are evaluated there in the according sandbox. Additionally, the editor’s contents can be submitted by pressing the keys `cmd + return`, respectively `ctrl + return`. Code submission
is locked until the submitted code has been evaluated. The button clear code irreversibly deletes all code written to the editor.

**Callargs** Using the input line beneath the editor window, one can append a space separated list of call arguments. In Repy, a call argument $i$ is found in the global list `callargs[i].`

**Standard Output** As discussed in subsection 4.3 Try Repy uses `log()` instead of `print` to write to the standard output. Therefore, the standard output section in Try Repy displays all `log()` statements in your Repy code. Additionally it displays uncaught exceptions. The standard output reloads dynamically during the evaluation in order to give a near-real time feedback. Upon code submission, the web interface automatically displays the standard output section.

**Session Log** The session log retrieves and displays the entire log of a sandbox’s lifetime. That is, every submitted code and its according output, combined with a timestamp, which illustrates the elapsed time between sandbox creation and code submission.

**Insert Files** The insert files section provides inserting one or multiple files from the local file system into the editor window without using copy and paste. Files can be inserted either at cursor position or at the top of the editor window. Additionally, one of three different file delimitation styles can be applied. One is multiple `#` characters and the `<filename>` before and after the inserted file, the second just writes `#<filename>` before and after the file and the third style applies no delimitation at all.

**Special Characters** Due to tedious keyboard layouts on mobile devices, the section special characters provides a clickable table of often used characters. Upon click, a character is inserted into the editor at cursor position.

**Code Snippets** This section offers multiple Repy code snippets and is especially interesting for Repy novices. The snippets are inserted into the editor by click. Amongst others, there are two snippets that demonstrate a UDP conversation in Repy. One snippet implements a listener, which listens for incoming UDP messages and the other one implements a sender, which sends UDP messages. To demonstrate the transmission, each snippet has to be run in a different sandbox. Use different browser windows/tabs.
**Editor Options**  The editor options concern the appearance of the editor such as the editor’s theme, line numbering and font size.

**Read Me**  In this section, you find a short version of the use guide.

### 4.3 Known Issues

In order to use Try Repy one has to be aware of the subsequent items that differ from common Repy development. Possible solutions to some of the issues are sketched in section [3].

**log()**  In Repy, one can write to standard output by using the `print` statement. Within Try Repy the `print` statement does not work. Instead of `print`, one has to call `log()`, passing the characters to be printed as arguments to the function. In RepyV2 [21], `log()` will be generally used instead of `print`.

**Stop Execution**  Currently there is no way to stop execution prematurely within a virtual namespace. Since Try Repy uses virtual namespaces to evaluate the submitted code, the execution can not be stopped by the user. The only way to do this would be shutting down the entire server, which of course would kill all other existing sandboxes.

**exitall()**  In Repy, the `exitall()` function is used to terminate all threads of an executing program. In Try Repy, plain `exitall()` would not only terminate all threads of an executing program in a sandbox, but all threads in all sandboxes and all server threads as well. Therefore, a user’s `exitall()` call is caught in the sandbox and it has no effect.

**Thread Exceptions**  Usually an exception in one thread would terminate the entire executing program. Giving the restrictions for user thread handling, an uncaught exception in Try Repy does not affect potential other healthy threads. It only terminates the thread where the exception occurred, and writes the output to Try Repy’s standard output and to the sandbox log. Other healthy threads, if there exist any, will continue execution.
TCP/UDP Listeners  One major missing feature in Try Repy sandboxes concerns the isolation of TCP/UDP listeners. In Repy, when a TCP or UDP listener is registered and waits for a connection respectively for incoming packets, it can be overridden by another listener registered on the same IP address and port. The last handler to be registered will be the one who establishes a TCP connection, respectively who will receive the UDP packets. Within a Try Repy sandbox, this genuine Repy behavior is deliberately adapted. Unfortunately, the overriding of listeners between Try Repy sandboxes and even between Try Repy sandboxes and the Try Repy web server is possible as well. This should be really fixed.

Ace Bugs  Try Repy uses the Ace JavaScript framework in order to provide the text editor. Since Ace is still work in progress, there might occur several bugs concerning the editor. One known issue is the missing compatibility of Ace with Apple’s iPad. Apparently, Ace’s text editor, which is actually an HTML text area, cannot bind the iPad’s keyboard so far.

5 Future Work

From a user’s perspective Try Repy needs fixes for the above mentioned known issues. Of course, the user interface can be ameliorated to an arbitrary extent in order to facilitate software development with Try Repy. Furthermore, there is the possibility to use Try Repy as subject of research itself regarding virtualization techniques. Some ideas for further development concerning both construction sites are discussed subsequently.

5.1 Try Repy as Development Environment

5.1.1 Threads

As mentioned, Python and consequently Repy’s thread manipulation mechanisms are very restricted. Therefore, most of the current flaws of Try Repy are somehow related to the difficulties of dealing with separate threads: There is no exiting all threads in one sandbox manually or when a thread exception is raised, without impact on threads in other sandboxes and the system threads of the Try Repy web server. So far, these issues are not really taken care of. One possible work-around would incorporate the context of a sandbox. Basically, the context of a sandbox is a Repy dictionary containing all globally available variables, respectively functions. Since it is not possible to manually stop threads, a way to do so anyhow would be to convince the threads stopping on their own, e.g. by raising exceptions in these
particular threads. This could be managed by overriding or deleting the sandboxes context. Hence, when a healthy thread calls a variable or a function, it would no longer find it in the context dictionary, and therefore naturally raise an exception. This would of course entail a valid recreation of the sandbox’s context for successive code submissions and their execution. In return, it would enable an `exitall()` call within the submitted Repy code or as clickable command on the Try Repy web interface, dedicated only to the according sandbox. Additionally, it would make it possible to abort the entire program when one thread raises an uncaught exception.

### 5.1.2 TCP/UDP Listeners

In order to deal with the overriding TCP/UDP listener problem, it would suffice to establish a globally accessible variable in the web controller that registers and deregisters all network listeners. Additionally, one would have to extend the existing sandbox’s network abstraction to refuse the registration of a listener at a certain IP, port and protocol, if it already exists in another sandbox or if the web server is using it.

Actually, this concern is heavily related to the resource scheduling challenges mentioned in subsection 5.2 and should be definitely treated regarding the great picture.

### 5.1.3 Preprocessing

Repy provides a preprocessor program [22], that makes it possible to use `include` – a surrogate for Python’s `import` – statements in Repy programs. Basically, the preprocessor takes an input Repy file and searches for all dependent Repy files indicated with an `include` statement in order to append them to the output Repy file. It would be useful to have such a functionality of preprocessing on the Try Repy web interface. Of course, the Try Repy preprocessor would have to be extended in order to check if the Repy files to include are valid to be used in Try Repy. That is, `print` statements would have to be replaced with `log()` statements. So far, Try Repy already provides an interface to insert locally stored files into the editor window.

Nevertheless, in the future Repy preprocessing will be phased out by the `dylink` module [23], which will dynamically include files as necessary.

### 5.1.4 User Interface

Further nice features, concerning the user interface, that have not been implemented yet, are for example a custom Repy syntax highlighter maybe with integrated syntax checking. So far, Try Repy uses the Ace Python syntax highlighter.
In the course of implementing a Repy syntax highlighter, one could append syntax references that suggest links to Python and Repy documentation sites, whenever a particular statement is written to Try Repy’s editor. These desires, as well as the Try Repy web interfaces platform compatibility, e.g. iPad support, are strongly related to the used JavaScript Framework Ace. Since Ace is under constant development, it is recommended to check out its progress and update the Ace build used in Try Repy.

5.2 Try Repy as Research Subject

From a network researcher’s point of view, the major interest in Try Repy as a research subject lies surprisingly within its network facilities. As described in section 3, Try Repy has a modular layout, which separates the user interface entirely from the web server, which is again separated from the sandboxes, where each user can execute his or her code. The sandbox already provides multiple abstractions of different Repy functions. One set of abstractions is related to Repy’s network operations. Extending these abstractions would enable to experiment with different scheduling mechanisms for the network resources provided below. Of course, this is practicable for different layers, or so to say on different levels of abstraction. In terms of highest granularity, scheduling could be applied between different threads of an executing program that run in one sandbox. On the next level, the scheduling would happen between each sandbox. And then again, a scheduling mechanism could portion the resources, distinguishing between system resources, i.e. all resources that are occupied by the web server, and sandbox resources that are required by an executing program in any sandbox.

To do so, two things are needed: On the one side, custom abstractions of the network resources calling functions on each level, and on the other side a globally accessible space, where the abstracted functions can register, respectively ask for the required resources.

6 Resumé

This thesis presented Try Repy, a web based software development and execution environment to be used for Restricted Python, the programming language used within the Seattle Network Testbed. As Try Repy is based on virtualization, an overview of virtualization concepts and how the presented piece of software fits there were also given. In addition to Try Repy’s features for Repy software developers, it was explained how Try Repy can be used as a subject of research itself, regarding virtualization.

Try Repy is hosted on the Seattle SVN at the University of Washington and licensed under the MIT license. The Ace build is licensed under tri-license.
MPL/LGPL/GPL [25]. If you are interested in using Try Repy, or want to take part in further development, check out the repository [27].

7 Acknowledgments

Above all, the author wants to thank his bachelor advisor, Albert Rafetseder, who gave some very valuable directions [28], and supported the author patiently throughout the development of Try Repy and the writing of this thesis. Thanks go to Florian Metzger and Christian Schwartz as well, who first had the idea of using the Ace JavaScript Framework for a web-based development environment for Repy, and of course to Kurt Tutschku, who gave the author the opportunity to work together with a lot of interesting people at the research group of Future Communication. Since Try Repy is based on Seattle [4] and Ace [19], the author also wants to thank those involved, especially for their well documented code and quick and helpful responses. Also to mention are Try Haskell [29], and Try Ruby [30], who unwittingly donated the name of the presented software.
Figure 1: Seattle components [I]
Figure 2: Try Repy Deployment
The web controller is the main program, which dispatches HTTP requests from the web interface and manages the sandboxes.

```
mycontext["clean_user_context"]
mycontext["user_dict"]
mycontext["lock_user_dict"]
```

```
serve()
serve_file()
serve_output_buffer()
serve_code()
serve_log()
serve_user()
make_response()
exist_user()
read_and_decode_form_data()
```

The file abstraction provides functions to perform file operations within the Try Repy namespace.

```
.is_user_file()
tr_listdir()
tr_open()
tr_removefile()
```

```
user
context
start_time
log
log_lock
tmp_logtime
tmp_output_buffer
lock_output_buffer
thread_count
thread_lock
thread_accounting_lock
my_log()
context_wrap()
evaluate_reply()
read_output_buffer()
write_output_buffer()
get_log()
thread_register()
thread_deregister()
```

Figure 3: Try Repy Server Architecture
Figure 4: Try Repy JavaScript Architecture

Figure 5: Try Repy Web Interface
References


