Rain Documentation

Release 0.3.0-pre

Stanislav Bohm, Vojtech Cima, Tomas Gavenciak

Jun 23, 2018

Contents

CHAPTER 1

Overview

Rain is an open-source distributed computational framework for large-scale task-based pipelines.

Rain aims to lower the entry barrier to the world of distributed computing and to do so efficiently and within any scale. Our intention is to develop a light yet robust distributed framework that features an intuitive Python API, straightforward installation and deployment with insightful monitoring on top.

Note: Despite that this is an early release of Rain, it is a fully functional project that can be used out-of-the box. Being aware that there is still a lot that can be improved and added, we are looking for external users and collaborators to drive our future work, both enthusiasts, from the industry and the scientific community. Talk to us online at Gitter or via email and let us know what your project needs and use-cases, submit bugs or feature requests at GitHub or even contribute with pull requests.

- **Dataflow programming.** Computation in Rain is defined as a flow graph of tasks. Tasks may be built-in functions, Python code, or an external applications, short and light or long-running and heavy. The system is designed to integrate any code into a pipeline, respecting its resource requirements, and to handle very large task graphs (hundreds thousands tasks).
- **Easy to use.** Rain was designed to be easy to deployed anywhere, ranging from a single node deployments to large-scale distributed systems and clouds ranging thousands of cores.
- **Rust core, Python API.** Rain is written in Rust for safety and efficiency and has a high-level Python API to Rain core infrastructure, and even supports Python tasks out-of-the-box. Nevertheless, Rain core infrastructure provides a language-independent interface that does not prevent adding support for other languages in the future.
- **Tasks in Python/C++/Rust** Rain provides a way to define user-defined tasks in Python, C++, and Rust.
- **Monitoring** Rain is designed to support both online and postmortem monitoring.

** *Get started now.* **

1.1 What is in the box

Rain infrastructure composes of a central **server** component and **governor** components, that may run on different machines. A governor may spawn one or more **executors** that are local processes that provides execution of an external code. Rain is distributed with Python executor. Rain also provides libraries for C++ and Rust for writing own specialized executors.

Users interacts with server via **client** applications. Rain is distributed with Python client API.

1.1.1 Python Client

- Task-based programming model.
- High-level interface to Rain core infrastructure.
- Easy definition of various types of tasks and their inter-dependencies.
- Python3 module.

1.1.2 Rain Core Infrastructure

- Basic scheduling heuristic respecting inter-task dependencies.
- Rust implementation enabling easy build, deployment, and reliable run.
- Distributed as all-in-one binary.
- Direct governor-to-governor communication.
- Basic dashboard for execution monitoring.

1.1.3 Executors

- Possibility to define own tasks in Python, C++, and Rust

1.2 Future directions

There are many things to improve and even more new things to add. To work efficiently we need to prioritize, and for that we need your feedback and use cases. Which features would you like to see and put to good use? What kind of pipelines do you run?

1.2.1 Better dashboard

Better interactive view on the current and past computation status, including post-mortem analysis. Which stats and views give you the most insight?

1.2.2 Better scheduler

While surprisingly efficient, the current scheduler is currently mostly based on heuristics and rules. We plan to replace it with an incremental global scheduler based on belief propagation.

1.2.3 Resiliency

The current version supports and propagates some failures (remote python task exceptions, external program errors) but other errors still cause server panic (e.g. governor node failure). The near-term goal is to have better failure modes for introspection and possibly recovery. The system is designed to allow building resiliency against task or governor failures via checkpoints in the task graph (keeping file copies). It is not clear how useful to our users this would be but it is on our radar.

1.2.4 Resources

Currently, the only resources supported are CPU cores. We are working on also supporting memory requirements, but other resources (GPUs, TPUs, disk space, . . .) should be possible with enough work and interest.

1.2.5 Stream objects support

Some tasks work in a streaming fashion and it would be inefficient to wait for their entire output before starting a consumer task. We plan to include streaming data objects but there are semantic and usage issues about resources, scheduling, multiple consumers and resiliency.

1.2.6 REST client interface

The capnp API is a bit heavy-handed for a client API. We plan to create a REST API for the client applications, simplifying API creation in new languages, and to unify it with the dashboard/status query API. External REST apis are convenient for many users and they do not seem to be a performance bottleneck.

1.2.7 Easier Deployment in cloud settings

The Rust binary is already one statically linked file and one python-only library, making distribution easy and running on PBS is already supported. We would like to add better support for cloud settings, e.g. AWS and Kubernetes.

1.3 What we do *NOT* want to do

There are also some directions we do NOT intend to focus on in the scope of Rain.

1.3.1 Visual editor

We do not plan to support visual creation and editing of pipelines. The scale of reasonably editable workflows is usually very small. We focus on clean and easy client APIs and great visualization.

1.3.2 User isolation and task sandboxing

We do not plan to limit malicious users or tasks from doing any harm. Use existing tools for task isolation. The system is lightweight enough to have one instance per user if necessary.

1.3.3 Fair user scheduling, accounting and quotas

When running multiple sessions, there is no intention to fairly schedule or prioritize them. The objective is only overall efficient resource usage.

1.4 Comparison with similar tools

TODO

1.5 Roadmap

https://github.com/substantic/rain/issues/26

CHAPTER 2

Quickstart

2.1 Introducing Rain Applications

Rain Applications are programs **defined on client side** and **executed on Rain infrastructure** using the Rain API. Rain automaticaly distributes execution of the applications in a distributed environment.

Rain Applications follows the paradigm of task oriented programming. Basic building blocks for every Rain App are tasks - generic abstraction units representing various kinds of computations ranging from native Python tasks to 3rd party software.

Rain tasks may be arbitrarily* chained together and so provide complex high-level functionality.

2.2 Writing your first Rain Application

This section demonstrate how to start Rain infrastructure locally and execute a simple "Hello world" application.

- **Start Rain infrastructue** Although, the components of Rain (server and governor(s)) can be started manually, in order to simplify this process, we provide "rain start" command to do it for you automatically. The following command starts server and one local governor. (Starting Rain infrastructure on distributed systems is described in *Starting infrastructure*.):

```
$ rain start --simple
```

- **Running "Hello World" example.** The following Python program creates a task that joins two strings (This example is more explained in Section tasks-and-objs.):

```
from rain.client import Client, tasks, blob

# Connect to server
client = Client("localhost", 7210)

# Create a new session
```

(continues on next page)

(continued from previous page)

```
with client.new_session() as session:

    # Create task (and two data objects)
    task = tasks.Concat((blob("Hello "), blob("world!"),))

    # Mark that the output should be kept after submit
    task.output.keep()

    # Submit all crated tasks to server
    session.submit()

    # Wait for completion of task and fetch results and get it as bytes
    result = task.output.fetch().get_bytes()

    # Prints 'Hello world!'
    print(result)
```

CHAPTER 3

User's Guide

3.1 Basic terms

Task is a basic unit of work in Rain, it reads inputs and produces outputs. Tasks are executed on computational nodes (computers where Rain governors are running). Tasks can be external programs, python functions, and basic built-in operations.

Data objects are objects that are read and created by tasks. Data objects are immutable, once they are created they cannot be modified. They are generic data blobs or directories with accompanying metadata. It is upto tasks to interpret the data object content.

3.2 Task definition and submission

Rain represents your computation as a graph of tasks and data objects. Tasks are not eagerly executed during the graph construction. Instead, the actual execution is managed by Rain infrastructure after an explicit submission. This leads to a programming model in which you first only **define** a graph and then **execute** it.

Let us consider the following example, where two constant objects are created and merged together:

```
from rain.client import Client, tasks, blob

client = Client("localhost", 7210)  # Create a connection to the server
                                    # running at localhost:7210

with client.new_session() as session:  # Creates a session

    a = blob("Hello ")     # Create a definition of data object in the current session
    b = blob("world!")     # Create a definition of data object in the current session
    tasks.Concat([a, b])   # Create a task definition in the current session
                           # that concatenates input data objects

    session.submit()       # Send the created graph into the server, where the␣
↪computation
```

(continues on next page)

(continued from previous page)

```
                        # is performed.
    session.wait_all()  # Wait until all submitted tasks are completed
```

The graph composed in the session looks as follows:

When the graph is constructed, all created objects and tasks are put into the active session. In many cases, it is sufficient just to create one session for whole program lifetime, with one submit at the end. However, it is possible to create more sessions or built a graph gradually with more submits. More details are covered in Section *Sessions*.

3.3 Fetching data objects

Data objects produced by tasks are not transferred back to the client automatically. If needed, this can be done using the `fetch()` method. It returns `rain.common.DataInstance` that wraps data together with some additional information. To get raw bytes from `rain.common.DataInstance` you can call method `get_bytes()`.

In the following example, we download the result back to the Python client code. Expression `t.output` refers to the data object that is the output of task `t`:

```
from rain.client import Client, tasks, blob

client = Client("localhost", 7210)

with client.new_session() as session:
    a = blob("Hello ")
    b = blob("world!")
    t = tasks.Concat((a, b))
    t.output.keep()             # Tell server to keep result of task

    session.submit()            # Submit task graph

    result = t.output.fetch()   # Download result from the server
    print(result.get_bytes())   # Prints b'Hello world!'
```

By default, Rain automatically removes data objects that are no longer needed for further computation. Method `keep()` sets a flag to a given data object that instructs the server to keep the object until the client does not explicitly frees it. An object can be freed when the session is closed or when `unkeep()` method is called. Method `keep()` may be called only before the submit. Method `unkeep()` may be called on any "kept" object any time.

If method `fetch()` is called and the object has not been finished yet, the method blocks until the object is not finished. Note that this is the reason, why we did not use `wait_all()` in this example.

3.4 Inter-task dependencies

Naturally, an output of a task may be used as an input for another task. This is demonstrated by the following example. In the example, we use `tasks.Sleep(O, T)` that creates a task taking an arbitrary data object `O` and waits for `T` seconds and then returns `O` as its output. Being aware that such task is not very useful in practice, we find it useful as an intuitive example to demostrate the concept of task chaining:

```
from rain.client import Client, tasks, blob

client = Client("localhost", 7210)
```

(continues on next page)

(continued from previous page)

```
with client.new_session() as session:
    a = blob("Hello ")
    b = blob("world!")
    t1 = tasks.Sleep(b, 1.0)    # Wait for one second and then returns 'b'
    t2 = tasks.Concat((a, t1.output))
    t2.output.keep()

    session.submit()            # Submit task graph

    result = t2.output.fetch()  #  It will wait around 1 second
                                #  and then returns b'Hello world'
```

If a task produces only a single output, we can ommit `.output` and directly use the task as an input for another task. In our example, we can define `t2` as follows:

```
t2 = tasks.Concat((a, t1))
```

This shortened notation is used in the rest of the text.

3.5 More outputs

A task may generally create zero, one, or more outputs. All outputs are accessible via attribute `outputs`. That contains an instance of `rain.common.LabeledList`. It is an extension of a standard list (indexed from zero), that also allows to be accessed via string labels.

```
# The following task creates two outputs labeled "output1" and "output2" with
# an equivalent of 'cat data | tee output1 > output2'.
t = tasks.Execute(["tee", Output("output1")], stdout="output2", stdin=data)

t.outputs["output1"]  # Access to output "output1"
t.outputs["output2"]  # Access to output "output2"

# There is also some helper functions:
# Keep all outputs (equivalent to: for o in t.outputs: o.keep())
t.keep_outputs()

# After submit
# Fetch all outputs (equivalent to: [o.fetch() for o in t.outputs])
t.fetch_outputs()
```

If a task has more than one output or zero outputs, then accessing attribute `.output` throws an exception. Attribute `.outputs` is always availble independently on the number of outputs.

3.6 Object data types

Every data object represents either a single binary data blob or a directory. Since these two object *data types* behave very differently, they are distinguished and checked already when constructing the computation graph. The *data type* may be one of:

- 'blob' - Binary data block. May have a *Object content types* specified.
- 'dir' - Directory structure, see section *Directories*.

We consider developing other data object "modes", e.g. streams.

3.7 Object content types

Binary data objecs represent different type of data in different formats. The Rain infrastructure treats all data objects as raw binary blobs, and it is up to tasks to interpret them. Content type is a string identifier of the format of the data in tasks and clients. Python code also recognize some of content types and allows to deserialize them directly.

Currently recognized content types are:

- '' - Raw binary data, unknown or unspecified content type
- 'pickle' - Serialized Python object
- 'cloudpickle' - Serialized Python object via Cloudpickle
- 'json' - Object serialized into JSON
- 'cbor' - Object serialized into CBOR
- 'arrow' - Object serialized with Apache Arrow
- 'text' - UTF-8 string.
- 'text-<ENCODING>' - Text with specified encoding
- 'mime/<MIME>' - Content type defined as MIME type
- 'user/<TYPE>' - User defined type, <TYPE> may be arbitrary string

An object may have two different content-types: First, a type is specified when constructing the task graph. Second, the type may be set by the task executor dynamically (e.g. depending on some input data). If present, the latter is taken to be the actual content type and must be a sub-type of the former. Any type is considered a subtype of the unspecified type.

3.8 Constant data objects

Function `rain.client.blob()` serves for a creation of a constant data object. The content of the data object is uploaded to the server together with the task graph.

```
from rain.client import blob, pickled
blob(b"Raw data")  # Creates a data object with a defined content
blob(b"Raw data", label="input data")  # Data with a non-default label
                                       # (Default label is 'const')
blob("String data")  # Creates a data object from a string, the content type
                     # is set to 'text'
blob("[1, 2, 3, 4]", content_type="json")  # Data with a specified content type
blob([1, 2, 3, 4], encode="json")  # Serialize python object to JSON and set
                                   # content type to "json"
blob([1, 2, 3, 4], encode="pickle")  # Serialize python object by pickle
                                     # content type to "pickle"
pickled([1, 2, 3, 4])  # Short-cut for blob(..., encode="pickle")
```

3.9 Built-in tasks

The following tasks are supported directly by the Rain governor:

Concat **(`rain.client.tasks.Concat`)** Concatencates inputs into one resulting blob.

Load, LoadDir **(`rain.client.tasks.Load`, `rain.client.tasks.LoadDir`)** Creates data object from an external file or direftory. (Note: The current version does not support tracking external resources; therefore, this operation "internalizes" the file, i.e. it makes a copy of it into the working directory.)

Store **(`rain.client.tasks.Store`)** Saves data object to a filesystem. The data are saved into local file system of the governor on which the task is executed. This task is usually used for saving files to a distributed file system, hence it does not matter which governor performs the task.

Sleep **(`rain.client.tasks.Sleep`)** Task that forwards its input as its output after a specified delay. Mostly for testing and benchmarking.

Execute **(`rain.client.tasks.SliceDirectory`)** Run an external program with given inputs, parameters and resources. See `rain.client.Program` if you execute a program repeatedly with different data.

MakeDirectory **(`rain.client.tasks.MakeDirectory`)** Tasks that creates a directory combining the inputs under given paths.

SliceDirectory **(`rain.client.tasks.SliceDirectory`)** Tasks that extracts a file or subdirectory from a directory object.

```
# This example demonstrates usage of four built-in tasks
from rain.client import tasks, Client, blob

client = Client("localhost", 7210)

with client.new_session() as session:

    # Create tasks opening an external file
    data1 = tasks.Load("/path/to/data")

    # Create a constant object
    data2 = blob("constant data")

    # Merge two objects
    merge = tasks.Concat((data1, data2))

    # Sleep for 1s
    result = tasks.Sleep(merge, 1.0)

    # Write result into file
    tasks.Store(result, "/path/to/result")

    session.submit()
    session.wait_all()
```

(Examples for the directory-related tasks are in section *Directories*)

3.10 Running external programs

3.10.1 Task `tasks.Execute`

The whole functionality is built around built-in task `rain.client.tasks.Execute`. When a program is executed through `rain.client.tasks.Execute`, then a new temporary directory is created. This directory will be removed at the end of program execution. The current working directory of the program is set to this directory.

The idea is that this directory is program's sandbox where input data objects are mapped and files created in this directory may be moved out as new data objects when computation completes. Therefore, in contrast with many other workflow systems, programs in rain should not be called with absolute paths in arguments but use relative paths (to stay in its working directory). Governors try to avoid unnecessary data object replication in the cases when a data object is used by multiple tasks that run on the same governor.

If the executed program terminates with a non-zero code, then tasks fails and content of standard error output is written into the error message.

The simple example looks as follow:

```
tasks.Execute("sleep 1")
```

This creates a task with no inputs and no outputs executing program "sleep" with argument "1". Arguments are parsed in shell-like manner. Arguments can be also specified explicitly as a list:

```
tasks.Execute(("sleep",  "1"))
```

Command may be also interpreted by shell, if the argument `shell=True` is provided:

```
tasks.Execute("sleep 1 && sleep 1", shell=True)
```

3.10.2 Outputs

Files created during task execution or task standard output can be used as the output of `rain.client.tasks.Execute`. The following example calls program `wget` that downloads web page at https://github.com/ and saves it as `index.html`. The created file is forwarded as the output of the task.

```
from rain.client import Client, task, Output

client = Client("localhost", 7210)

with client.new_session() as session:
    t = tasks.Execute("wget https://github.com/",
                      output_paths=[Output("index", path="index.html")])
    t.output.keep()

    session.submit()
    result = t.output.fetch().get_bytes()
```

The class `rain.client.Output` allows to configure the outputs. The first argument is the label of the output. The argument `path` sets the path to the file used as output. It is a relative path w.r.t. the working directory of the task. If the path is not defined, then label is used as path; e.g. `Output("my_output")` is equivalent to `Output("my_output", path="my_output")`. The Output instance can be also used for specification of additional attributes such content type or size hint. Please see the class documentation for more details.

If we do not want to configure the output, it is possible to use just a string instead of instance of `Output`. It creates the output with the same label and path as the given string. Therefore we can create the previous task as follows:

```
t = tasks.Execute("wget https://github.com/", output_paths=["index.html"])
```

The only difference is that label of the output is now "index.html" (not "index").

Of course, more than one output may be specified. Program `wget` allows to redirect its log to a file through `--output-file` option:

```
t = tasks.Execute("wget https://github.com/ --output-file log",
                  outputs_paths=["index.html", "log"])
```

This creates a task with two outputs with labels "index.html" and "log". The outputs are available using standard syntax, e.g. `t.outputs["log"]`.

Outputs can be also passed directly as program arguments. This is a shortcut for two actions: passing the output path as an argument and putting output into `output_paths`. The example above can be written as follows:

```
t = tasks.Execute(["wget", "https://github.com/", "--output-file", Output("log")],
                  output_paths=["index.html"])
```

The argument `stdout` allows to use program's standard output:

```
# Creates output from stdout labelled "stdout"
tasks.Execute("ls /", stdout=True)

# Creates output from stdout with label "my_label"
tasks.Execute("ls /", stdout="my_label")

# Creates output through Output object, argument 'path' is not allowed
tasks.Execute("ls /", stdout=Output("my_label"))
```

3.10.3 Inputs

Data objects can be mapped into the working directory of `rain.client.tasks()`. The simplest case is to use a data object directly as arguments for a program. In such case, the data object is mapped into randomly named file and the name is placed into program arguments. Note that files are by default mapped only for reading (and proctected by setting file permissions). More options of mapping is described in *Mapping data objects onto filesystem*.

```
from rain.client import Client, task, blob

client = Client("localhost", 7210)

with client.new_session() as session:
    data = blob(b"It is\nrainy day\n")

    # Maps 'data' into file XXX where is a random name and executes
    # "grep rain XXX"
    task = tasks.Execute(["grep", "rain", data], stdout=True)
    task.output.keep()

    session.submit()
    print(task.output.fetch().get_bytes())  # Prints b"rainy day"
```

For additional settings and file name control, there is `rain.client.Input`, that is a counter-part for `rain.client.Output`. It can be used as follows:

```
from rain.client import Client, task, Input

...

# It executes a program "a-program" with arguments "argument1" and "myfile"
# and while it maps dataobject in variable 'data' into file 'myfile'
my_data = ... # A data object
```

(continues on next page)

(continued from previous page)

```
task = tasks.Execute(["a-program", "argument1",
                      Input("my_label", path="myfile", dataobj=my_data)])
```

The argument `input_paths` of `rain.client.tasks.Execute` serves to map a data object into file without putting its filename into the program arguments:

```
# It executes a program "a-program" with arguments "argument1"
# and while it maps dataobject in variable 'data' into file 'myfile'
tasks.Execute(["a-program", "argument1"],
              input_paths=[Input("my_label", path="myfile", dataobj=my_data)])
```

The argument `stdin` serves to map a data object on the standard input of the program:

```
# Executes a program "a-program" with argument "argument1" while mapping
# a data object on the standard input
tasks.Execute(["a-program", "argument1"], stdin=my_data)
```

3.10.4 Factory `Program`

In many cases, we need to call the same program with the same argument set. Class `rain.client.Program` serves as a factory for `rain.client.tasks.Execute` for this purpose. An instance of `Program` can be called as a function that takes data objects; the call creates a task in the active session.

```
from rain.client import Client, blob, Program, Input

rain_grep = Program(["grep", "rain", Input("my_input", path="my_file")], stdout=True)

client = Client("localhost", 7210)

with client.new_session() as session:
    data = blob(b"It is\nrainy day\n")

    # Creates a task that executes "grep rain my_file" where dataobject in variable
    # 'data' is mapped into <FILE>
    task = rain_grep(my_input=data)
```

`Program` accepts the same arguments as `execute`, including `input_paths`, `output_paths`, `stdin`, and `stdout`. The only difference is that in all places where data object could be used, `Input` instance (without `dataobj` argument) has to be used, since `Program` defines "pattern" indepedently on a particular session.

3.11 Python tasks

In addition to built-in tasks, Rain allows to run additional types of tasks via executors. Rain is shipped with Python executor, that allows to execute arbitrary Python code.

3.11.1 Decorator `@remote`

Decorator `rain.client.remote()` turns a python function into a Rain task. Let us consider the following example:

```
from rain.client import Client, remote

@remote()
def hello(ctx):
    return "Hello world!"

client = Client("localhost", 7210)

with client.new_session() as session:
    t = hello()                 # Create a task
    t.output.keep()
    session.submit()
    result = t.output.fetch()
    print(result)               # Prints b'Hello world!'
```

The decorator changes the behavior of the decorated function in a way that calling it no longer executes it in the client but creates a task that executes the function in a python executor. Governor starts and manages executors as necessary, there is no need of any action from the user.

The decorated function should accept at least one argument. As the first argument, the context of the execution is passed to the function. Context enables some actions within the task. It is a convention to name this argument as `ctx`.

3.11.2 Inputs

Decorated function may take more parameters than `ctx`; these parameters define inputs of the task. By default, they can be arbitrary Python objects and they are serialized via `cloudpickle`. If the decorated function is called with a data object, it is invokend with `rain.common.DataInstance` that contains data defined by the object:

```
from rain.client import Client, remote, blob

@remote()
def hello(ctx, data1, data2):
    return data1 + data2.get_bytes()

client = Client("localhost", 7210)
with client.new_session() as s:

    # Create data object
    data = blob("Rain!")

    # Creates a task calling function 'hello' in governor
    t = hello(b"Hello ", data)

    t.output.keep()
    s.submit()
    s.wait_all()

    # Prints b'Hello Rain!"
    print(t.output.fetch().get_bytes())
```

In remotely executed Python code, Rain data objects are replaced with actual data instances. All occurences of data objects are replaced, even those encapsulated in own data structures:

```
class MyClass:

    def __init__(self, my_data):
```

(continues on next page)

(continued from previous page)

```
        self.my_data = my_data


@remote()
def my_call(ctx, input):
    # If we assume a call of this function as below,
    # we obtain an instance of MyClass where attribute 'my_data'
    # is list of instances of DataInstance
    return b""

...

my_instance = MyClass([blob(b"data1"), blob(b"data2"), blob(b"data3")])
task = my_call(my_instance)
```

Note: It is possible to pass also generators as arguments to remote functions, and it works as expected. However, Rain has to include all data objects occuring in related expressions as task dependencies. Therefore, you may create more dependencies then expected. To avoid this problems, we recommend to evaluate generators before passing to remote functions, especiialy if it is a filtering kind of generator.

All metadata of data objects (including content type) are passed to the data instances occuring in remote functions. Therefore, it is possible to call method `load()` on data instances to deserialize objects according to their content types:

```
@remote()
def fn1(ctx, data):
    # Load according content type. Throws an error if content type is not provided
    loaded_data = data.load()
    ...

# Automatically call load() on specific argument
@remote(inputs={"data": Input(load=True)})
def fn2(ctx, data):
    ....

# Automatically call load() on all arguments
@remote(auto_load=True)
def fn3(ctx, data):
    ....

# Example of calling:
data = blob([1,2,3,4], encode="json")
fn1(data)
```

The second case uses `rain.client.Input` to configure individual parameters. It can be also used for additional configurations, like data-object size hints for Rain scheduler, or content type specification:

```
# The following function asks for a dataobject with content type "json" as
# its argument. If the function is called the following happens:
# 1) If the input dataobject has content type "json", it is passed as it is
# 2) If the input dataobject has no content type (None), then content type "json"
     is set as the object content type
# 3) If the input dataobject has content type different from "json", the task fails
```

(continues on next page)

(continued from previous page)

```
@remote(inputs={"data": Input(content_type="json")})
def fn1(ctx, data):
    pass
```

3.11.3 Outputs

By default, it is expected that a remote function returns one data object. It may return an instance of `bytes` or `str` that will be used as content of the resulting data object. If an instance of bytes is returned then the content type of resulting object is `None`, if a string is returned then the content type is set to "text". A remote function may also return a data instance, when you want to set additional attributes of data object. More outputs may be configured via `outputs` attribute of remote:

```
@remote()
def fn1(ctx):
    return b"Returning bytes"

@remote()
def fn2(ctx):
    return "Returning string"

# Configuring more unlabaled outputs
@remote(outputs=3)
def fn3(ctx):
    (b"data1", b"data2", b"data3")

# No output
@remote(outputs=0)
def fn4(ctx):
     pass

# Configuring labeled outputs
@remote(outputs=("label1", "label2"))
def fn5(ctx):
     return {"label1": b"data1", "label2": b"data2"}

# Set content types of resulting objects
@remote(outputs=(Output(content_type="json"), Output(content_type="json"))
def fn6(ctx):
    return ("[1, 2, 3]", "{'x': 123}")

# Automatically encode resulting objects
@remote(outputs=(Output(encode="pickle"), Output(encode="json"))
def fn7(ctx):
    return ([1, 2, 3], {"x": 123})
```

3.11.4 Debug stream

Method `debug` on the context allows to write messages into debug stream that can be found in task attribute "debug" and it is also part of an error message when the task fails.

```
@remote()
def remote_fn(ctx):
    a = 11
```

(continues on next page)

(continued from previous page)

```
    b = a + 10
    ctx.debug("Variable a = {}", a)
    ctx.debug("Variable b = {}", b)
    raise Exception("Error occured!")

# When this task is executed, you get the following error message:
#
# Exception: Error occured!
#
# Debug:
# Variable a = 11
# Variable b = 21
```

3.11.5 Type hints

If you are using sufficiently new Python (>=3.5), you can use type hints to define outputs and inputs, e.g.:

```
@remote
def test1(ctx, a : Input(content_type="json")) -> Output(encode='pickle', label='test_
↪pickle');
    pass
```

3.12 Resources

In the current version, the only resource that can be configured is the number of cpus. This following example shows how to request a a specific number of cpus for a task:

```
# Reserve 4 CPUs for execution of a program
tasks.Execute("a-parallel-program", cpus=4)

# Resere 4 CPUs for a Python task
@remote(cpus=4)
def myfunction(ctx):
    pass
```

3.13 Attributes 'spec' and 'info'

Most of the information about the tasks and data objects falls into two categories:

- The user-created specification data (*spec*).
- The information about the task execution and object computation (*info*).

These are stored and transmitted separately. Once the objects and tasks are submitted, the spec is immutable. The info is initially empty and is set by the governor (and in part by the task executor). When a task or object is finished, info is also immutable.

The data is transmitted as JSON, attributes with values `None`, empty strings and empty lists may be omitted when encoding.

A client may ask for info attributes of any task/object as long as session is open; "keep" flag is not necessary. Attributes are not updated automatically, `fetch()` or `update()` has to be called to update attributes.

3.13.1 Error, debugn and user

The task info and object info share `error` attribute. When non-empty, the task is assumed to have failed. You may specify `error` of an object to indicate the error more precisely, but it usually indicates a failure of the generating task. Note that empty `error` is assumedto mean success even if explicitly present.

The `debug` attribute is intended for any log messages from Rain or the user, especially for internal and external debugging. General node progress is normally not logged here as it is contained in the Rain event log. This is the only attribute that is not immutable once set and may be appended to.

Both task and object info and spec have a `user` dictionary intended for any JSON-serializable data for any purpose. The keys prefixed with _ are used internally in testing and development.

3.13.2 Task spec and info

Task spec (::`rain.common.attributes.TaskSpec` in Python) has the following attributes:

- `id` - Task ID tuple, type `rain.common.ID`.
- `task_type` - The task-type identificator (e.g. "executor/method").
- `config` - Any task-type specific configuration data, JSON-serializable.
- `inputs` - A list of input object IDs and labels as ::`rain.common.attributes.TaskSpecInput` * `id` - Input object ID. * `label` - Optional label.
- `outputs` - List of output object IDs.
- `resources` - Dictionary with resource specification.
- `user` - Arbitrary user json-serializable attributes.

Task info (::`rain.common.attributes.TaskInfo` in Python) has the following attributes:

- `error` - Error message. Non-empty error indicates failure.
- `start_time` - Time the task was started.
- `duration` - Real-time duration in seconds (floating-point number).
- `governor` - The ID of the governor that executed this task.
- `debug` - Debugging log, usually empty.
- `user` - Arbitrary json-serializable objects.

3.13.3 Data object spec and info

Data object spec (::`rain.common.attributes.ObjectSpec` in Python) has the following attributes:

- `id` - Object ID tuple, type `rain.common.ID`.
- `label` - Label (role) of this output at the generating task.
- `content_type` - Specified content type name, see *content type*.
- `data_type` - Object data type, `"blob"` or `"dir"`.
- `user` - Arbitrary user json-serializable attributes.

Data object info (::`rain.common.attributes.ObjectInfo` in Python) has the following attributes:

- `error` - Error message. Non-empty error indicates failure.

- `size` - Final size in bytes (approximate for directories).
- `content_type` - Content type after execution. Note that this must be a sub-type of `spec.content_type`.
- `debug` - Debugging log, usually empty.
- `user` - Arbitrary json-serializable objects.

3.13.4 Python API

In the client, the attributes are available as `spec` and `info` on `rain.client.Task` and `rain.client.DataObject`.

An example of fetching and querying the attributes at the client:

```
with client.new_session() as s:
    task = tasks.Execute("sleep 1")
    s.submit()

    s.wait_all()

    # Download recent attributes
    task.update()

    # Print name of governor where task was executed
    print(task.info.governor)
```

In the python executor and remote tasks, the object attributes are available on the input `rain.common.DataInstance`, the task attributes on the execution context (::`rain.executor.context.Context`).

An example of remote attribute manipulation:

```
@remote()
def attr_demo(ctx):
   # read user defined attributes
   foo = ctx.spec.user["foo"]

   # setup new "user_info" attribute
   ctx.info.user["bar"] = [1, 2, foo]

   # Write some debug log
   ctx.debug("Running at governor", ctx.info.governor)
   return b"Result"

with client.new_session() as session:
    task = attr_demo()
    task.spec.user["foo"] = 42
    session.submit()
    session.wait_all()
    task.update()

    # prints: [1, 2, 42]
    print(task.info.user["bar"])

    # prints the debug log
    print(task.info.debug)
```

3.14 Waiting for object(s) and task(s)

Waiting for a completion of a single task/object is done using the `wait()` method directly on awaited task or data object. Multiple tasks/objects can be awaited at once using the `wait` method with a set of tasks/obejcts on session:

```
with client.new_session() as session:
    a = blob("Hello world")
    t1 = tasks.Sleep(a, 1.0)
    t2 = tasks.Sleep(a, 2.0)
    session.submit()

    t1.wait()  # This blocks until t1 is finished, independently of t2
    t2.output.wait()  # Waits until a data object is not finished

    # The same as two lines above, but since we are doing it at once, it is
    # slightly more efficient
    session.wait([t1, t2.output])
```

Note: Note that in the case of `wait()` (in contrast with `fetch()`), object does not have to be marked as "kept".

3.15 Directories

Rain allows to use directories in the similar way to blobs. Rain allows to create directory data objects that can be passed to `tasks.Execute()`, remote python code, and other places without any differences. There are only two specific features:

- If a directory dataobject is mapped to a file system it is mapped as directory (not as a file as in the case of blobs).
- If a directory is viewed as raw bytes (e.g. method `get_bytes` on data instance), tar file is returned.

A data type of an object (blob/directory) is a part of the task graph and has to be determinated during its construction. To specify it in places where `Input` and `Output` classes are used, there are classes `rain.client.InputDir` and `rain.client.OutputDir`.

```
from rain import

from rain.client import Client, tasks, blob, OutputDir, directory

client = Client("localhost", 7210)

with client.new_session() as session:

    # Creates a directory object from client's local file system
    # Recursively collects all files and directories in /path/to/dir
    d = directory("/path/to/dir")

    # Create blob data objects
    data1 = blob(b"12345")
    data2 = blob(b"67890")

    # Task that creates a directory from data objects
    d2 = tasks.MakeDirectory(tasks.make_directory([
         ("myfile.txt", data1),  # Map 'data1' as file 'myfile.txt' into directory
```

(continues on next page)

(continued from previous page)

```
        ("adir", d),  # Map directory 'd' as subdir named 'adir'
        ("a/deep/path/x", data2),  # Map 'data2' as a file 'x'; all subdirs on path␣
↪is created
    ])

    # Task taking a file from a directory data object
    d3 = tasks.SliceDirectory(d2, "a/deep/path/x")

    # Task taking a directory from a directory data object
    # This is indicated by  '/' at the end of the path.
    d3 = tasks.SliceDirectory(d2, "a/deep/")

    # Taking directory as outpout of task.execute
    tasks.Execute("git clone https://github.com/substantic/rain",
                  output_paths=[OutputDir("rain")])
```

3.16 Mapping data objects onto filesystem

Rain knows two methods of maping a data objects onto filesystem.

- **write** - creates a fresh copy of data objects is created on filesystem that can be freely modified. Changes of the file is *not* propagated back to data object.
- **link** - symlink to the internal storage of governor. The user can only read this data. This method may silently fall back to 'write' when governor has no file system representation of the object.

Task `rain.client.tasks.Execute()` maps files by **link** method. It can be changed by `write` argument of `Input`:

```
# Let 'obj' contains a data object

# THIS IS INVALID! You cannot modified linked objects
tasks.Execute("echo 'New line' >> myfile", shell=True,
              input_paths=[Input("myfile", dataobj=obj)])

# This is ok. Writable copy of 'obj' is created.
tasks.Execute("echo 'New line' >> myfile", shell=True,
              input_paths=[Input("myfile", dataobj=obj, write=True)])
```

Data instance has methods `write(path)` and `link(path)` that performs the mapping to a given path. They can be used on both in executor and in client. Let us note that in the current version **link** in the client always falls back to **write**. Example:

```
@remote()
def my_remote_function(ctx, input1):
    input1.write("myfile")  # Writes data into 'myfile' that can be edited
                            # without change of the original object
    input1.link("myfile2")  # Creates a read-only file system representation
                            # of data object
```

Warning: Read-only property in linking method is forced by setting up file rights. Therefore, as far you do not change permissions of files/directories, you are proctected against accidental modifications of data objects. If you

change permissions or content of linked data objects, the behavior is undefined. Let us remind that Rain is designed only for execution of trusted codes. Obviously this kind of isolation is **not** a protection against malicious users.

3.17 Sessions

3.17.1 Overview

The client allows to create one or more sessions. Sessions are the environment scopes where application create task graphs and submit them into the server. Sessions follows the following rules:

- Each client may manage multiple sessions. Tasks and data object in different sessions are independent and they may be executed simultaneously.
- If a client disconnects, all sessions created by the client are terminated, i.e. running tasks are stopped and data objects are removed. (Persistent sessions are not supported in the current version)
- If any task in a session fails, the session is labeled as failed, and all running tasks in the session are stopped. Any access to tasks/objects in the session will throw an exception containing error that caused the problem. Destroying the session is the only operation that does not throw the exception. Other sessions are not affected.

3.17.2 Active session

Rain client maintains a global stack of sessions and `with` block moves a session on the top of the stack and removes it from the stack when the block ends. The session on the top of the stack is called *active* session. The following example demonstrates when a session is active:

```
from rain.client import Client, tasks, blob

client = Client("localhost", 7210)

# no session is active
with client.new_session() as a:

    # 'a' is active

    with client.new_session() as b:
        # 'b' is active
        pass

    # 'b' is closed and 'a' is active again

# 'a' is closed and no session is active
```

Tasks and data objects are always created within the scope of active session.

Note: Which session is active is always a local information that only influences tasks and data objects creation. This information is not propagated to the server. Submitted tasks are running regardless the session is active or not.

3.17.3 Closing session

Session may be closed manually by calling method `close()`, dropping the client connection or leaving `with` block. To suppress the last named behavior you can use the `bind_only()` method as follows:

```
session = client.new_session()

with session.bind_only():
    # 'session' is active
    pass

# 'session' is not active here; however it is NOT closed
```

Once a session is closed, it is pernamently removed from the session stack and cannot be reused again.

Note: The server holds tasks' and objects' metadata (e.g. performance information) as long as a session is alive. If you use a long living client with many sessions, sessions should be closed as soon as they are not needed.

3.17.4 Multiple submits

The task graph does not have to be submmited at once; multiple submmits may occur during the lifetime of a session. Data objects from previous submits may be used while constructing a new new submit, the only condition is that they have to be marked as "kept" explicitly.

```
with client.new_session() as session:
   a = blob("Hello world")
   t1 = tasks.Sleep(a, 1.0)
   t1.output.keep()

   session.submit()  # First submit

   t2 = tasks.Sleep(t1.output, 1.0)

   session.submit()  # Second submit
   session.wait_all()  # Wait until everything is finished

   t3 = tasks.Sleep(t1.output, 1.0)

   session.submit()  # Third submit
   session.wait_all()  # Wait again until everything is finished
```

Let us remind that method `wait_all()` waits until all currently running task are finished, regardless in which submit they arrived to the server.

CHAPTER 4

Writing Own Executors

This section covers how to write a new executor, i.e. how to create a program that introduces new tasks type to Rain. A governor spawns and stops executors as needed according tasks that are assigned to it. Each tasks always specifies what kind of executor it needs.

There are generally two types of executors: **Universal executors** and **Specialized executors**. The universal one allows to execute an arbitrary code and specialized offers a fix of tasks that they provide.

The current version of Rain supports universal executor for Python. This is how `@remote()` decorator works. It serializes a decorated function into a data object and creates a task that needs Python executor that executes it.

For languages where code cannot be simply transferred in a portable way, Rain offers **tasklibs**, a libraries for writing specialized executors. The current version provides tasklibs for C++ and Rust. A tasklib allows to create a stand-alone program that know how to communicate with governor and provides a set of functions.

This sections shows how to write new tasks using tasklibs for C++ and Rust and how to create run this tasks from client.

Note: Governor itself also provides some of basic task types, that are provided through a virtual executor called **buildin**. You may see this "executor" in dashboard.

4.1 Rust tasklib

The documentation for writing executor in Rust can be found at https://docs.rs/rain_task/. Registration of an executor into a governor and using client API are same for all executors (*Registration in governor* and *Client API*).

4.2 C++ tasklib

Note: C++ tasklib is not fully finished. It allows to write basic task types, but some of more advanced features (e.g. working with attributes) are not implemented yet.

4.2.1 Getting started

The following code shows how to create an executor named "example1" that provides one task type "hello". This task takes one blob as the input, and returns one blob as the output.

```
#include <tasklib/executor.h>

int main()
{
  // Create executor, the argument is the name of the executor
  tasklib::Executor executor("example1");

  // Register task "hello"
  executor.add_task("hello", [](tasklib::Context &ctx, auto &inputs, auto &outputs) {

      // Check that we been called exactly with 1 argument.
      // If not, the error message is set to context
      if (!ctx.check_n_args(1)) {
          return;
      }

      // This is body of our task, in our case, it reads the input data object
      // inserts "Hello" before the input and appends "!"
      auto& input1 = inputs[0];
      std::string str = "Hello " + input1->read_as_string() + "!";

      // Create new data instance and set it as one (and only) result
      // of the task
      outputs.push_back(std::make_unique<tasklib::MemDataInstance>(str));
  });

  // Connect to governor and serve registered tasks
  // This function is never finished.
  executor.start();
}
```

4.2.2 Building

To compile the example we need to creating following file structure:

- myexecutor
  - myexecutor.cpp – Source code of our example
  - CMakeFile.txt – CMake configuration file. The content is below.
  - tasklib – Copy of tasklib from Rain repository (located in `rain/cpp/tasklib`)

Content of `CMakeFile.txt` is following:

```
cmake_minimum_required(VERSION 3.1)
project(myexecutor)

add_subdirectory(tasklib)

add_executable(myexecutor
               myexecutor.cpp)

target_include_directories(myexecutor PUBLIC ${CBOR_INCLUDE_DIRS} ${CMAKE_CURRENT_
↪SOURCE_DIR}/src)
target_link_libraries (myexecutor tasklib ${CBOR_LIBRARIES} pthread)
```

Now, we can build the executor as follows:

```
$ cd myexecutor
$ mkdir _build
$ cd _build
$ cmake ..
$ make
```

4.3 Registration in governor

When you write your own executors, you have to registrate them in the governor. For this purpose, you have to create a configuration file for governor.

As an example, let us assume that we want to register called "example1".

```
[executors.example1]
    command = "/path/to/executor/binary"
```

The configuration is in TOML format. If we save it as `/path/to/config.toml` we can provide the path to the governor by starting as follows:

```
rain governor <SERVER_ADDRESS> --config=/path/to/config.toml
```

or if you are using "rain start":

```
rain start --simple --governor-config=/path/to/config
```

More about starting Rain can be found at *Starting infrastructure*.

4.4 Client API

This section describes how to call own tasks from Python API.

Each task contains a string value called `task_type` that specifies executor and function. It has format `<EXECUTOR>/<FUNCTION>`. So far we have created (and registered) own executor called `example1` that provides task `hello`. The task type is "example1/hello'.

The followig code creates a class `Hello` that serves for calling our task:

```
from rain.client import Task


class Hello(Task):
    """ Task takes one blob as input and puts b"Hello " before
        and "!" after the input. """

    TASK_TYPE = "example1/hello"

    def __init__(self, obj):
         # Define task with one input and one output,
         # Outputs may be a (labelled) list of data objects or a number.
         # If a number is used than it creates the specified number of blob outputs
         super().__init__(inputs=(obj,), outputs=1)
```

This class can be used to create task in task graph in the same way as tasks from module `rain.client.tasks`, e.g.:

```
with client.new_session() as session:
    a = blob("Hello world")
    t = Hello(a)
    session.submit()
    print(t.output.fetch().get_bytes())  # prints b"Hello WORLD!"
```

CHAPTER 5

Installation, Running & Deployment

Rain Distributed Execution Framework consists is an all-in-one binary. Rain API is a pure Python package with a set of dependencies installable via pip.

5.1 Binaries

Rain provides a binary distribution for Linux/x64. The binary is almost fully statically linked. The only dynamic dependancies are libc and sqlite3 (for logging purpose).

Latest release can be found at https://github.com/substantic/rain/releases. It can be downloaded and unpacked as follows:

```
$ wget https://github.com/substantic/rain/releases/download/v0.3.0-pre/rain-v0.3.0-
↪pre-linux-x64.tar.xz
$ tar xvf rain-v0.3.0-pre-linux-x64.tar.xz
```

Installation of Python API:

```
$ pip3 install rain-python
```

5.2 Build via cargo

If you have installed Rust toolchain, you can use cargo to build Rust binaries and skip manual download:

```
$ cargo install rain_server
```

Note that you still have to install Python API through pip:

```
$ pip3 install rain-python
```

5.3 Build from sources

For building from sources, you need Rust and SQLite3 (for logging) and Capnp compiler (for compiling protocol files) installed on your system.

```
# Example for installation of dependencies on Ubuntu

# Installation of latest Rust
$ curl https://sh.rustup.rs -sSf | sh

# Other dependencies
$ sudo apt-get install capnproto libsqlite3-dev
```

For building Rain, run the following commands:

```
$ git clone https://github.com/substantic/rain
$ cd rain
$ cargo build --release
```

After the installation, the final binary can be found `rain/target/relase/rain`.

Installation of Python API:

```
$ cd python
$ python setup.py install
```

5.4 Starting infrastructure

5.4.1 Starting local governors

The most simple case is running starting server and one governor with all resources of the local machine. The following command do all work for you:

```
$ rain start --simple
```

If you want to start more local governors, you can use the following command. It starts two governors with 4 assigned cpus and one with 2 assigned cpus:

```
$ rain start --local-wokers=[4,4,2]
```

5.4.2 Starting remote governors

If you have more machines that are reachable via SSH you can use the following command. We assume that file `my_hosts` contains addresses of hosts, one per line:

```
$ rain start --governor-host-file=my_hosts
```

Let us note, that current version assumes that assumes for each host that Rain is placed in the same directory as on machine from which command is invoked.

If you are running Rain inside PBS scheduler (probably if you are using an HPC machine), then you can simple run:

```
$ rain start --autoconf=pbs
```

It executes governor on each node allocated by PBS scheduler.

Note: We recommended to reserve one CPU for server unless you have long runnig tasks. This reservation can be done through cgroups, or CPU pinning.

Another option (with less isolation) is to use option `-S`:

```
$ rain start -S --governor-host-file=my_hosts
```

If a remote machine is actually localhost (and therefore runs Rain server) then `--cpus=-1` argument is used for the governor on that machine, i.e. the governor will consider one cpu less on that machine.

5.4.3 Starting governors manually

If you need a special setup that is not covered by `rain start` you can simply start server and governors manually:

```
$ rain server                     # Start server
$ rain governor <SERVER-ADDRESS>   # Start governor
```

5.5 Arguments for program *rain*

5.5.1 Synopsis

```
rain start --simple [--listen=ADDRESS] [--http-listen=ADDRESS]
         [-S] [--runprefix=CMD] [--logdir=DIR] [--workdir=DIR]
         [--governor-config=PATH]
rain start --autoconf=CONF [--listen=ADDRESS] [--http-listen=ADDRESS]
         [-S] [--runprefix=CMD] [--logdir=DIR] [--workdir=DIR]
         [--governor-config=PATH] [--remote-init=COMMANDS]
rain start --local-governors [--listen=ADDRESS] [--http-listen=ADDRESS]
         [-S] [--runprefix=CMD] [--logdir=DIR] [--workdir=DIR]
         [--governor-config=PATH]
rain start --governor-host-file=FILE [-S] [--listen=ADDRESS]
         [--http-listen=ADDRESS]
         [-S] [--runprefix=CMD] [--logdir=DIR] [--workdir=DIR]
         [--governor-config=PATH] [--remote-init=COMMANDS]

rain server [--listen=LISTEN_ADDRESS] [--http-listen=LISTEN_ADDRESS]
            [--logdir=DIR] [--ready-file=<FILE>]
rain governor [--cpus=N] [--workdir=DIR] [--logdir=DIR]
            [--ready-file=FILE] [--config=PATH] SERVER_ADDRESS[:PORT]
rain --version | -v
rain --help | -h
```

5.5.2 Command: start

Starts Rain infrastructure (server & governors), makes sure that everything is ready and terminates.

–simple Starts server and one local governor that gains all resources of the local machine.

–autoconf=CONF Automatic configuration from the environment. Possible options are:

- *pbs* - If executed in an PBS job, it starts server on current node and governor on each node.

–local-governors=RESOURCES Start local with a given number of cpus. E.g. –local-governors=[4,4,2] starts three governors: two with 4 cpus and one with 2 cpus.

–governor-config=PATH Path to governor config. It is passed as –config argument for all governors.

–governor-host-file=FILE Starts local server and remote governors. FILE should be file containing name of hosts, one per line.

The current version assumes the following of each host:

- SSH server is running.
- Rain is installed in the same directory as on the machine from which that `rain start` is executed.

-S Serves for reserving a CPU on server node. If remote governor detects that it is running on the same machine as server then it is executed with `--cpus=-1`.

The detection is based on checking if the server PID exists on the remote machine and program name is "rain".

–listen=(PORT|ADDRESS|ADDRESS:PORT) Set listening address of server. Default is 0.0.0.0:7210.

–http-listen=(PORT|ADDRESS|ADDRESS:PORT) Set listening address of server for HTTP (dashboard). Default is 0.0.0.0:8080.

–runprefix Set a command before rain programs. It is designed to used to run analytical tools (e.g. –runprefix="valgrind –tool=callgrind")

–logdir=DIR The option is unchanged propagated into the server and governors.

–workdir=DIR The option is unchanged propagated into governors.

–remote-init=COMMAND Commands executed on each remote connection. For example: `--remote-init="export PATH=$PATH:/path/bin"`.

5.5.3 Command: server

Runs Rain server.

–listen=(PORT|ADDRESS|ADDRESS:PORT) Set listening address of server. Default is 0.0.0.0:7210.

–logdir=DIR Set logging directory of server. Default is /tmp/rain/logs/server-<HOSTNAME>-PID.

–ready-file=FILE Create file containing a single line "ready", when the server is fully initialized and ready to accept connections.

5.5.4 Command: governor

Runs Rain governor.

SERVER_ADDRESS[:PORT] An address where a server listens. If the port is omitted than port 7210 is used.

–config=PATH Set a path for a governor config.

–cpus=N Set a number of cpus available to the governor (default: 'detect')

- If 'detect' is used then the all cores in the machine is used.
- If a positive number is used then value is used as the number of available cpus.

- If a negative number -X is used then the number of cores is detected and X is subtracted from this number, the resulting number is used as the number of available cpus.

–listen=(PORT|ADDRESS|ADDRESS:PORT) Set listening address of governor for governor-to-governor connections. When port is 0 then a open random port is assigned. The default is 0.0.0.0:0.

–logdir=DIR Set the logging directory for the governor. Default is `/tmp/rain/logs/governor-<HOSTNAME>-<PID>/logs`.

–ready-file=FILE Creates the file containing a single line "ready", when the governor is connected to server and ready to accept governor-to-governor connections.

–workdir=DIR Set the working directory where the governor stores intermediate results. The defautl is `/tmp/rain/work/governor-<HOSTNAME>-<PID>`

> **Warning:** Rain assumes that working directory is placed on a fast device (ideally ramdisk). Avoid placing workdir on a network file system.

CHAPTER 6

Examples

6.1 Distributed cross-validation with libsvm

```
# ============================================================
# This example creates a simple cross-validation pipeline
# for libsvm tools over IRIS data set
#
# Requirements:
# 1) Installed svm-train and svm-predict
#    (libsvm-tools package on Debian)
# 2) IRIS data set in CSV format, e.g.:
#    https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/
↪iris.csv
# ============================================================

import os
from rain.client import Client, tasks, Program, Input, Output, remote

THIS_DIR = os.path.dirname(os.path.abspath(__file__))
DATA_FILE = os.path.join(THIS_DIR, "iris.csv")
CHUNKS = 3


# Convert .csv to libsvm format
@remote()
def convert_to_libsvm_format(ctx, data):
    lines = [line.split(",") for line in data.get_str().rstrip().split("\n")]
    lines = lines[1:]  # Skip header
    labels = sorted(set(line[-1] for line in lines))

    result = "\n".join("{} 1:{} 2:{} 3:{} 4:{}".format(
        labels.index(line[4]),
        line[0], line[1], line[2], line[3]) for line in lines)
    return result
```

(continues on next page)

(continued from previous page)

```
def main():

    # Program: SVM train
    # svm-train has following usage: svm-train <trainset> <model>
    # It reads <trainset> and creates file <model> with trained model
    train = Program(("svm-train", Input("data"), Output("output")))

    # Porgram: SVM predict
    # svm-predict has following usage: svm-predict <testdata> <model> <prediction>
    # It reads files <testdata> and <model> and creates file with prediction and
    # prints accuracy on standard output
    predict = Program(("svm-predict", Input("testdata"), Input("model"), Output(
↪"prediction")),
                      stdout=Output("accuracy"))

    # Connect to rain server
    client = Client("localhost", 7210)
    with client.new_session() as session:

        # Load data - this is already task, so load is performed on governor
        input_data = tasks.Load(DATA_FILE)

        # Convert data - note that the function is marked @remote
        # so it is not executed now, but on a governor
        converted_data = convert_to_libsvm_format(input_data)

        # Using unix command "sort" to shuffle dataset
        randomized_data = tasks.Execute(("sort", "--random-sort", converted_data),␣
↪stdout=True)

        # Create chunks via unix command "split"
        chunks = tasks.Execute(("split", "-d", "-n", "l/{}".format(CHUNKS),␣
↪randomized_data),
                               output_files=["x{:02}".format(i) for i in range(CHUNKS)]).
↪outputs
                               # Note that we are taking "outputs" of the task here ==> ^
↪^^^^^^^

        # Make folds
        train_sets = [tasks.Concat(chunks[:i] + chunks[i+1:]) for i, c in␣
↪enumerate(chunks)]

        # Train models
        models = [train(data=train_set) for train_set in train_sets]

        # Compute predictions
        predictions = [predict(model=model, testdata=data) for model, data in␣
↪zip(models, chunks)]

        # Set "keep" flag for "accuracy" output on predictions
        for p in predictions:
            p.outputs["accuracy"].keep()

        # Submit and wait until everything is not completed
        session.submit()
```

(continues on next page)

(continued from previous page)

```
        session.wait_all()

        # Print predictions
        for p in predictions:
            print(p.outputs["accuracy"].fetch().get_str())


if __name__ == "__main__":
    main()
```

CHAPTER 7

Python API

The Rain Pyton API consists of two domains that observe the workflow graph differently, although the concepts are similar and some classes are used in both contexts.

- Code run at *the client*, creating sessions and task graphs, executing and querying sessions. There, the tasks are only created and declared, never actually executed.
- Python code that runs inside remote Pyhton tasks on the governors. This code has access to the actual input data, but only sees the adjacent data objects (input and output).

7.1 Client API

7.1.1 Client

One instance per connection to a server.

7.1.2 Session

One instance per a constructed graph (possibly with multiple submits). Tied to one `Client`.

7.1.3 Data objects

Tied to a `Session`.

7.1.4 Tasks

Tied to a `Session`.

7.1.5 Attributes

7.1.6 Input and Output

These are helper objects are used to specify task input and output attributes. In particular, specifying an `Output` is the preferred way to set properties of the output `DataObject`.

7.1.7 Builtin tasks and external programs

Native Rain tasks to be run at the governors.

7.1.8 Data instance objects

Tied to a session and a `DataObject`. Also used in *Remote Python tasks*.

7.1.9 Resources

Note: TODO: Describe and document task resources.

7.1.10 Labeled list

7.2 Remote Python tasks

API for creating routines to be run at the governors. Created by the decorating with `remote` (preferred) or by `Remote`.

Whe **specifying** the remote task in the client code, the relevant classes are `Remote`, `Input`, `Output`, `RainException`, `RainWarning`, `LabeledList` and the decorateor `remote`.

Inside the running remote task, only `RainException`, `RainWarning`, `LabeledList`, `DataInstance` and `Context` are relevant.

The inputs of a `Remote` task are arbitrary python objects containing a `DataInstance` in place of every `DataObject`, or loaded data object if `autoload=True` or `load=True` is set on the `Input`.

The remote should return a list, tuple or `LabeledList` of `DataInstance` (created by `Context.blob()`), `bytes` or `string`.

CHAPTER 8

Contributors's Guide

We welcome contributions of any kind. Do not hesitate to open an GitHub issue or contant us via email; this part of documentation is quite sparse.

8.1 Scripts

- `utils/checks/stylecheck.sh` – Runs code style checks (flake8 & dry cargo fmt)
- `utils/checks/fullcheck.sh` – Runs stylecheck.sh + all available tests
- `utils/dist/make_release.py` – Compiles release binary of Rain, creates Python package and publishes release on GitHub

8.2 Testing

Rain contains two sets of tests:

- Unittests (in Rust)
- Integration tests (in Python)

8.2.1 Python tests

Python tests are placed in `/rain/tests/pytests`.

To execute them simply run `py.test-3` in the root directory of Rain. The logs are stored in `rain/tests/pytests/work`.

Important notes:

- Make sure you are running Python 3 py.test.

- Working directory `rain/tests/pytests/work` is fully cleaned before every test! Therefore, if you want to see logs, make sure that no other test is executed after the test you want to see. See options `-x` and `-k` of py.test-3
- By default, Python tests run with rain binary located in `rain/target/debug/` directory. This path can be modified using RAIN_TEST_BIN environment variable.

8.3 Dashboard

Rain Dashboard is implement in JavaScript over NodeJs. However, we do not want to have Node.js as a hard dependency when Rain is built from sources. Therefore, compiled form of Dashboard is included in Rain git repository. Neverthless, if you want to work on Dashboard, you need to install Node.js.

8.3.1 Installation

(We assume that you have already installed Node.js.)

```
cd dashboard
npm install
```

8.3.2 Development

For development, just run:

```
npm start
```

It starts on Rain dashboard on port 3000. Now you can just edit dashboard sources, **without** recompiling Rain binary. Dashboard in the development mode assumes, that http rain server is running at localhost:8080. If you need, you can change the address in `dashboard/src/utils/fetch.js`, but do not commit this change, please.

8.3.3 Deployment

All Dashboard resources (including JS source codes) are included into Rain binary. When Rain is compiled, files in `dashboard/dist` are read. To generate `dist` directory from actual dashboard sources, you need to run:

```
cd dashboard
sh make_dist.sh
```

Then you need rebuild Rain (e.g. `cargo build`). When you finish work on dashboard, do not forget to include files in `dist` into repository.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search