Open source framework for interactive data exploration in server based architecture

D5.5 v1.0

WP5 – Visual Analytics: D5.5 Open source framework for interactive data exploration in server based architecture

Dissemination Level: Public

Lead Editor: Janez Demsar

Date: 30 April 2015

Status: Final

Description from Description of Work:

T5.4. Interactive data exploration in server based architecture

We will develop techniques to split the processing between the database and the server (typically a supercomputer or a cloud) on one side and the client on the other, with both parties connected on a non-local network. The setup will allow for interactive data analysis of large volume data without the significant lags or artefacts that are common in solutions based on video streaming. The transferred data will suffice for the common operations performed on the visualization (e.g. zooming in, rotating, selecting subsets etc.), with this part of processing done on the client. Data transfer will be organized so that the user gets a preview of reasonable quality (for instance through a representative subsample of data) even before all data points have been transferred. Transferring the final stages of rendering the data to client will reduce the server load and ensure that the visualizations remain responsive when servers are experiencing high load.

Contributors:

Anze Staric, UL
Janez Demsar, UL

Internal Reviewer(s):

UNIMAN: Geoffrey Ndu
2ndQ: Mark Wong

Version History

Version	**Date**	**Authors**	**Sections Affected**
0.1	Apr 25, 2015	A Staric, J Demsar	Initial draft
0.2	Apr 29, 2015	J Demsar	Minor fixes in various sections
1.0	Apr 30, 2015	J Dix	Final version

Table of Contents

List of Figures

1. Summary

We defined and implemented a framework for data exploration in which the data is kept on a remote server and is processed there, while at the same time it allows the user to interact with the tool in a similar fashion as with the existing desktop tools. In particular, we developed a remote version of Orange in which the code is executed on the server and returns proxy objects that mimic the functionality of the corresponding objects of the standard Orange. Proxy objects can be used in further computation, thus allowing for “semi-lazy evaluation”, in which only the actions that involve showing actual results to the user require waiting for the computation to complete.

This document describes the implemented framework. We start with a short description of the architecture, followed by more detailed description of the API and technical details.

2. Overview

Remote Orange is a python package that allows the execution of python objects on a remote server. The following example trains a Logistic Regression model on iris data set on a remote server, predicts the class of the first instance in the training set and prints it out.

```
from orangecontrib import remote
with remote.server('127.0.0.1:9465'):
    import Orange
    from Orange.classification.logistic_regression import \
        LogisticRegressionLearner

iris = Orange.data.Table('iris')
logreg = LogisticRegressionLearner()(iris)
print(logreg(iris[0]))
```

The `with` statement provides a context in which the standard Orange module is substituted by the remote Orange on the specified server. Among the last three lines, the first two return proxy objects immediately, and only the last returns actual data (a string) and requires computation to finish.

Unlike other python packages for remote execution such as xmlrpc and Pyro, Remote Orange automatically generates proxy files for complete packages and transmits them to the client. Execution is asynchronous: method calls on remote objects yield new proxies that are used to access the results as they become available.

By retaining the same API when accessing remote libraries, this approach does not break the code completion in modern Python IDEs (e.g. PyCharm). IDE provides code completion for local version of the library, but since the API is the same, IDEs suggestions are relevant.

3. Architecture

The architecture of a system using Remote Orange for remote execution of tasks is shown in Figure 1. It consists of clients, application server, workers and data storage.

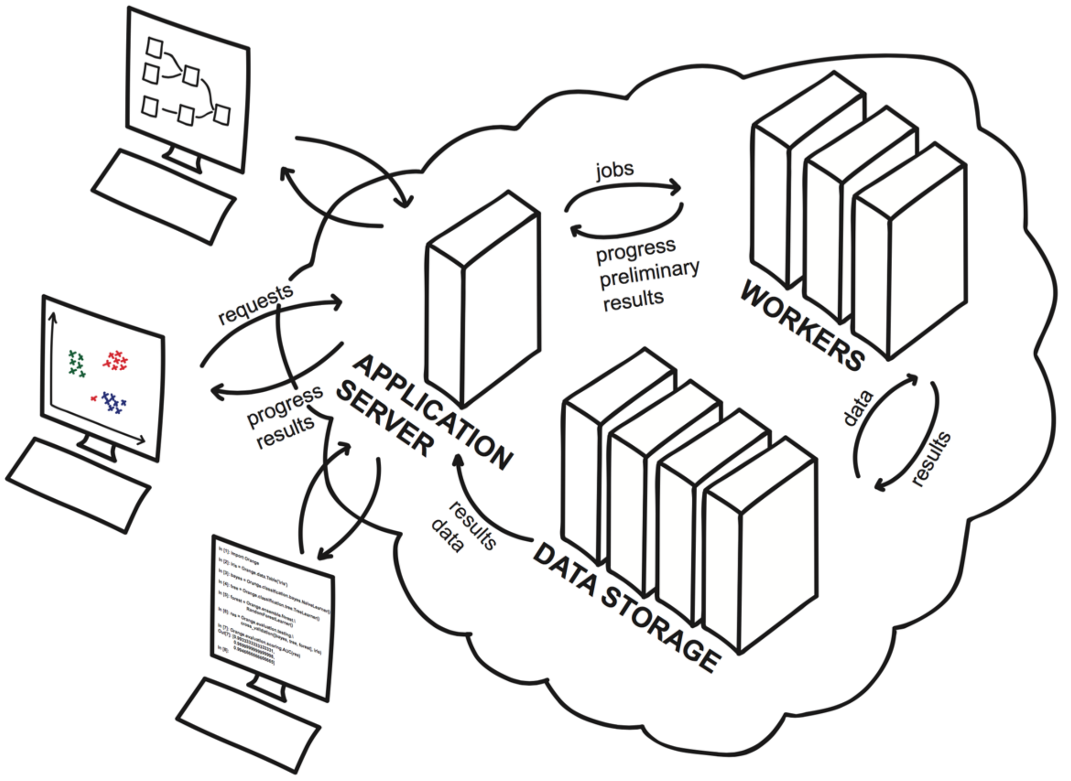

Figure 1. Architecture for remote execution using Remote Orange. Clients communicate with Application server using HTTP requests to upload data, request execution, query for execution status and download intermediate or final results. Application server schedules execution of remote tasks on workers, which perform the actual computation in parallel.

Data and results are transferred to/from workers when needed/available.

Application server is a HTTP server which provides a RESTful (Representational State Transfer) API. It receives requests from clients and acts on them. GET requests are used to query for status, execution state and final results. They are synchronous and block if the data asked for is not yet available. POST requests are used to upload Python objects and schedule or abort the execution of computational tasks. Each POST requests returns a unique id, which is later used to query the status or fetch results from the server. No client code is executed on the application server, execution is delegated to the workers instead.

Workers execute the code related to the computational tasks received from the application server. This includes creation of objects, access to their members and methods calls. When a result of one computational tasks is used as an input to another, workers delay the execution until all results are available. Intermediate results during the computation and final results, when the task completes, are stored in data storage.

Data storage stores data related to the remote objects. Data storage distinguishes between data and state objects. Data objects are either uploaded from the client or are results of the computational tasks executed on workers. When used as inputs to the submitted computational tasks, they are transmitted to workers where their value is used in execution. State objects are intermediate results of the computation and are stored whenever save_state is called in a long-running computation. They are accessible by the application server but expire once the computation completes.

Clients that access this architecture can be of different types, from scripts and GUI programs (Orange), to the interactive shell in the python interpreter, which can be used to trigger remote execution. Access to the application server is abstracted in a virtual python package containing proxy classes for available modules. These proxies provide the same API as if the library is used locally.

4. Implementation

4.1 Proxies

`Proxy` is a python class with several members and methods (Figure 2). When a new `Proxy` instance is created, a POST request is made to the server scheduling the creation of the matching remote object. The unique id returned by the server is stored in attribute `__id__`.

Methods `get`, `get_state`, and `ready` trigger GET requests to the server and return result, the last saved state or execution status of the job with the id stored in proxy. Abort is a destructive operation and triggers a POST requests the aborts the computation.

Calls to other methods/members of the proxy on the local machine schedule the corresponding call on the server using a POST request. The id that the request returns is used to construct a new proxy object, which is returned to the caller.

```
class Proxy:
    __id__ = uuid.uuid1()
    # uid corresponding the remote task id

    def get():
        # Retrieve the final results.
        # Block until the result is available.

    def get_state():
        # Retrieve the last saved computation state.
        # Computation needs to manually save state using
        # save_state function.

    def ready():
        # Return True if result is available, False otherwise

    def abort():
        # Abort the execution of the task. If the task has not been
        # scheduled yet, its execution will not start at all.
```

Figure 2. API of the Proxy class

There are two types of proxy objects in Remote Orange. A named proxy is generated by traversing the package and inspecting the class members. Named proxies have all the methods of the original class and provide code completion in interactive interpreters such as ipython. Client can obtain the contract containing the definitions of all available proxy objects from the server using a GET request for resource 'contract'. Anonymous proxies are used for results of the method calls, when the type of results is unknown (as the actual call has not yet been performed). They do not perform any validation and blindly schedule all method calls on the server.

When the server is queried for the contract, it returns an instance of `RemoteModule` (Figure 3). This object is generated when the server starts by traversing the packages that the servers is configured to execute. Description of each module and class is stored inside the `RemoteModule` instance. When this instance is downloaded to the client and unpickled, a dynamic module is created for each module in the description and a named proxy is added to the corresponding module for each class in the description.

```
class RemoteModule:
    # Stores a description of a python package in a format
    # that can be pickled and transferred between computers

    descriptions = {}
    # A dictionary mapping names to module descriptions

    modules = {}
    # A dictionary mapping names to proxy modules/objects

    def __init__(module, exclude=()):
        # Create a new RemoteModule for given module.
        #
        # exclude: a list of string prefixes. All modules and objects
        # that begin with one of these will not be included in the
        # RemoteModule

    def create_modules():
        # Create modules from stored descriptions
        #
        # Called from __init__ and after unpicling, as
        # dynamic modules cannot be pickled

    def __import__(name):
        # import function that returns a proxy object for name.
        # Raises ImportError when a proxy for name does not exist.
```

Figure 3. API of the RemoteModule object

When the remote server is used using remote.server context handler, like in the introductory example, python's `__import__` function is temporarily replaced with the `RemoteModule`'s. The latter yields remote proxies with modules/classes the `RemoteModule` instance is aware of. When modules/classes that are not available on remote server are imported inside this context manager, an `ImportError` is raised.

4.2 Data Storage

Data storage is managed by two classes, `ResultManager` and `StateManager`.

```
class ResultsManager:
    def register_result(id):
        # Notify the manager that a computation producing
        # a result with id has been scheduled.

    def set_results(id, result):
        # Store result and notify waiting parties that the
        # result is available

    def get_result(id):
        # Retrieve result with given id. If result is not
        # available, blocking until it is ready.
        #
        # Raises KeyError if result is not available and
        # has never been registered with this
        # ResultsManager.

    def has_result(id):
        # Return True if result is available

    def awaiting_result(id):
        # Return True if result has been registered with this manager
```

Figure 4. API of the ResultsManager class.

`ResultsManager` (Figure 4) is responsible for storing and retrieving the results. The implementation of `get_result` method must be able to block the execution while the result is not available. Stored results are immutable, so they can be cached on workers as long as they are used in new computations.

StateManager (Figure 5) manages the temporary storage of intermediate results produced by long running computations. These results are usually downloaded to the client using proxies `get_state` method and are not needed any more when the computation completes. Method `save_state` may be called often so its implementation should be fast, while avoiding concurrency issues with the `get_state` method.

```
class StateManager:
    # State manager is responsible for saving
    # and retrieving execution state.
    #
    # Implementations needs to define the following methods:

    def save_state(state, id):
        # Save execution state

    def get_state(id):
        # Get the last saved execution state
        # Return None, no state has been saved

    def delete_state(id):
        # Remove all saved execution states for given id.
        # Called when computation is done.
```

Figure 5. API of the StateManager class.

4.3 HTTP API

The client and the server communicate using HTTP protocol. Fetching the status of the computation and downloading of intermediate and final results uses GET commands, while POST commands are used for data upload, scheduling of tasks and aborting of execution.

GET requests

GET object/{id}

Get pickled result of the computation with id \{id\}. Wait until result is available.

GET state/{id}

Get pickled last saved state of the computation with id {id}.

GET status/{id}

Get pickled status of the computation {id}.

POST requests

POST requests are used for uploading objects and sending commands that result in computational tasks. Which of these modes will be used depends on the content_type of the request.

content_type='application/octet-stream'

POST data is expected to be a pickled python object. Object is stored in ResultsManager with a new id. This id is returned in response.

content_type='application/json'

POST data should deserialize into one of the available commands (Figure 6).

```
# Create - create a new remote object
{'create': {'module': 'path.to.module',
            'class_': 'ClassName',
            'args': ['list', 'of', 'args'],
            'kwargs': {'dict': 'of', 'keyword': 'args'}}}

# Call - call a method on remote object
{'call': {'object': {'__jsonclass__': ['Promise', 'id']},
          'method': 'method_name',
          'args': ['list', 'of', 'args'],
          'kwargs': {'dict': 'of', 'keyword': 'args'}}}

# Get - return a member of remote object
{'call': {'object': {'__jsonclass__': ['Promise', 'id']},
          'member': 'member_name'}}

# Abort - abort exection of a remote command
{'abort': {'id': 'id_of_remote_object'}}
```

Figure 6. Supported commands and their JSON representation

5. Demonstration

Remote Orange provides one of the two mechanisms for remote execution implemented within AXLE. The other, which is used in most other tasks, is execution on the database server itself. Both allow the user to run data analytic methods without transferring the data from the remote location. In comparison with the execution on the database server, Remote Orange is useful for computations that are not practical for SQL or extension modules (e.g. when SQL queries or similar extensions would be too cumbersome or slow, where C extensions are considered unsafe), or when the required functionality exists in Python and its use within the database would require reimplementing it.

To demonstrate the usefulness of the implemented modules within the context of task description (gradual improvement of visualization with processing on the remote server and minimal data transfer), we implemented iterative PCA on the remote server and the corresponding widgets in Orange.

We uploaded a video of a live demo to https://youtu.be/g6u0iRtiqkE. The widget SQL Table connects to the database and selects a table with 100 million rows. Note that although the connection is shown as localhost, the port is actually forwarded to another computer. The PCA widget uses a remote server (in this case on the same computer as the database) to iteratively compute PCA. The results of the computation are continuously updated and sent to the widgets Data Table that shows the principal components and the Heat map that shows a discrete scatter plot (heat map) along the first two axes in the projection.

The window at the bottom right is the server log and shows how the PCA widget polls the state from the remote PCA.

6. Source code and licenses

All functionality presented in this report is already included in the working version of Orange, available on its website (http://orange.biolab.si/orange3/). The source code is released under the BSD license (except for the GUI part, which is under GPL due to its dependency on PyQt), and available on Github (https://github.com/biolab/orange3).