Fachhochschule
Bonn-Rhein-Sieg

Fachbereich Informatik
Department Of Computer Sciences

Provenance-CSL
A Provenance Client Side Library

by
Roland Gude
Marius Oster

Contents

1 Introduction

Data management is a challenge in both scientific and technical environments. Therefore researchers have developed a special interest in this field. Modern approaches (i.e. Subversion, CVS) already offer authoring and versioning in distributed systems. However this might be insufficient in a vast number of scenarios, where not only the data resulting from a process, but also data which describes the process that generated those results is crucial.

For example, if a doctor needs to decide how to treat a patient, he must have access to the patient's data. Moreover he needs to know how the data was obtained (which tests were made), how old it is (when the tests were made), by whom it was provided (which doctor treated the patient in the past) and so on. This means that not only the patient's condition, i.e. his blood-pressure, needs to be documented, but also the process which led to the data. This process is called the Provenance of the data. Meta-data describing such a process is the process documentation [16].

There are many more scenarios where the process of acquiring the data might be almost as important as the data itself. The task of collecting a processes documentation might be quite easy in simple local systems, but becomes rather difficult in distributed environments.

The EU-Project *Provenance* [20] aims at the development of an open architecture which enables grid-applications to collect and collaborate process documentation in such environments.

The general design of the Provenance-architecture is built around a *Provenance store*. Applications, or actors, which are part of a process may record *P-Assertions*, which are the atomic units of the process documentation, on the Provenance store. They may query the Provenance store for certain P-Assertions as well.

In order to ease the process of developing applications which make use of Provenance, a client side library, which offers a simple API for interaction with Provenance stores, needs to be developed. This report describes the design of a Provenance-architecture, provides details of the API specification and presents the implementation of a Provenance client side library.

1.1 Motivation

The central point of the architecture of a Provenance system is the Provenance store. This store enables Provenance-aware applications (that is applications that are involved in a certain process) to store meta data describing process documentation. The Provenance store provides applications with a rather complicated and sophisticated API specification to cover all its functionality.

In order to ease the process of developing Provenance-aware application, a Provenance client side library needs to be developed. A client side library provides developers with an defined high-level API, which enables them to access and invoke webservices. The goal of the Provenance client side library is to reduce development costs of Provenance-aware applications. This is done by hiding irrelevant details of the communcation between a client and a Provenance store from developers of such applications.

Such a client side library needs to implement the defined protocols [10, 15, 22] and it needs to communicate with a Provenance store using a defined technology binding (like the SOAP binding defined in [17, 22])). It furthermore needs to supply an easy-to-use API for creating, storing and querying process documentation. PReServ, a implementation of a Provenance store, comes with a client side library for Java applications. Client side libraries for other programming languages do not exist. This more or less limits the programming languages in which Provenance-aware applications can be written without to much effort to Java. Since Python is a programming language which is widely used in scientific environments, a field where Provenance-awareness might be of great use, developing a client side library for Python applications suggests itself.

1.2 Related Work

1.2.1 Version Control Systems, Issue Trackers and Continous Integration Systems

Version Control Systems are strongly related to Provenance systems. The goal of a Version Control System is to track changes within a unit of data and to provide a version history. They are widely used in software engineering in order to manage different versions of the source code. They usually provide a mechanism to comment on the changes as well, thereby providing the possibility to find out who made certain changes for what reasons. Comments usually are submitted in free text and so it depends on the user if he provides the necessary information.

It is common to use an issue tracker to keep track of problems and features wich require code changes and to address an issue by providing the issue ID in the comment field of the version control system. Additionaly software engineers use continous integration systems to run tests and building the source code on a regular basis in order to identify issues.

The use of a Version Control System in combination with an issue tracker and a continous integration system provides what could be called the process documentation of software development process and thereby enables the identification of the provenance of a software application. However such a setup is highly specialized and not applicable in many research fields.

Version Control Systems, Issue Trackers and Continous Integration Systems are designed for distributed environments.

Common Version Control Systems are:

- Concurrent Versioning System (CVS) (centralized) [6]
- Subversion (SVN) (centralized) [29]
- Bazaar-NG (bzr) (decentralized) [2]

Common Issue Trackers are:

- Bugzilla (centralized) [4]
- Mantis (centralized) [14]
- Bugs Everywhere (be) (decentralized) [3]

Common Continous Integration Systems are:

- Anthill [1]
- CruiseControl [5]
- DamageControl [7]

1.2.2 Logging Systems

Modern software systems usually take advantage of a logging system in order to create logs of activities and events throughout the system. This is usually done by storing messages which contain a timestamp and a description of the occured event or the current activity. Most logging system allow the definition of loglevels which enables a classification of the log messages or can indicate its severity (i.e. *debug*, *info* or *error*).

Logging systems can not be used to indicate associations or relations between multiple events or activities. Thus it is not known by a logging system which log messages have an impact on each other. Therefore the capabilities of a logging system can be compared to those provided by Interaction P-Assertions and Actorstate P-Assertions (see 3.1.2).

Logging systems are a common part of many of today‘s programming languages standard libraries. Furthermore several projects exist which aim at the development of sophisticated logging systems.

Examples for logging systems are:

- The python *logging* module (Part of Python standard library) [23].
- The java *java.util.logging* package (Part of Java standard library) [12].
- Apache Log4j (Java logging system) [13].

1.2.3 Other Provenance systems

Provenance of an entity is used as measurement for its identity, value or trustworthyness in several research fields and has thus been addressed multiple times. For example the provenance of a piece of fine art is used as a help to determine its value. Several Provenance systems are discussed in Chapter 10 of [9].

2 Methods and Materials

This chapter briefly describes the used technologies, methods and materials which had a significant impact on the project.

2.1 Python

Python [25] is a highly dynamic, object oriented script and programming language. It is well known for its easy, clear and expressive syntax and its useful built-in datastructures. Because of these properties it is easy to learn and software written in Python usually has a very good maintainability. Therefore it is widely used in scientific environments where programmers usually are not computer scientists. Software engineers use Python not only for rapid prototyping or development of applications but also enable other applications to be extended with python scripts.

Python programs are organized in packages, modules and classes. A package is the directory in which a module, which is a file, resides. Every package can contain subpackages and modules. Modules can contain multiple classes. Unlike other object oriented languages, Python does not force the definition of classes.

Python was the only programming language that has been used during the development of Provenance-CSL.

2.2 SOAP

SOAP[1] is a protocol for data exchange based on the eXtended Markup Language (XML). It can be used for remote procedure calls (RPC) and is the most widely used protocol for communication with web services. SOAP communication is message based, i.e. one side sends a request and the other side answers with a result. Even though it is independent from any transport protocol, SOAP messages are usually exchanged using the Hypertext Transfer Protocol (HTTP).

A SOAP message consists out of a top-level container element, called the envelope. The envelope contains an (optional) head element and a body element. Furthermore the used namespaces are defined in the envelope. The head can be used to state meta information like used encryptions. The body contains the payload of the message, like which method to call and which parameters to supply, or the return values of a method call.

The SOAP-bindings of the Provenance protocols have been used to communicate with Provenance stores.

2.3 Zolera SOAP Infrastructure

The Zolera SOAP Infrastructure (ZSI) [27] is an implementation of SOAP version 1.1 [28]. A special feature of ZSI is that it comes with a Web Service Definition Language (WSDL) compiler

[1] SOAP has been an acronym for Simple Object Access Protocol. Since the protocol is not simple at all and can not only be used for object access, SOAP is not longer considered to be an acronym.

wsdl2py, which generates Python stubs for the client side of a web service[2]. Since the Provenance protocols are defined using WSDL, this was an important feature.

ZSI has been used to generate Python code from the WSDL definition of the Provenance protocols. It has also been used for all SOAP communication.

2.4 Python Enterprise Application Toolkit

The Python Enterprise Application Toolkit (PEAK) [18] is a collection of Python modules which adds useful features for component based design to Python. Its subpackages *importutils* and *pyprotocols* have been used to enable lazy loading and automated protocol adaption.

2.4.1 Lazy Loading

Lazy loading is a technique that allows the importing or loading of a library on demand. This is useful if an application takes advantage of a significant number of libraries. If those libraries do not support lazy loading in such a scenario, the application would be forced to load all libraries at startup which might be needed during the execution process. This can take quite a long time and is one reason why some applications need a long time to start. If the applications libraries support lazy loading, the libraries won't be loaded before the moment they are needed, thus scattering the loading time over the whole runtime of the application. Therefore libraries that are not needed in a particular run will not be loaded during that run. This technique can significantly reduce startup time of an application.

A prominent example for lazy loading is the Eclipse IDE [8]. A common Eclipse installation consists of more than 150 plugins. If all those plugins were loaded at startup and loading of each plugin would only take one single second, the complete startup process of Eclipse would take more than two or three minutes.

The *importutils* package of PEAK allows to define modules as *lazy modules*. It is completely compatible with the normal Python importing mechanism.

2.4.2 Protocols

Unlike other object oriented programming languages, Python makes no use of anything like interfaces. In some object oriented programming languages (i.e. Java [11]), interfaces are used to describe the methods a class has to provide in order to implement an interface. As an alternative concept, *PyProtocols* [24] introduces protocols and protocol adaption to Python.

Protocols are used to describe the behaviour of objects by defining which methods have to be supported and which members (i.e. variables, types) have to be provided in order to support the object. A really valuable feature of PyProtocols is the automatic adaption mechanism, which allows automated adaption from one datatype *d1* which supports a protocol *p1* to support another protocol *p2* if an adapter from *p1* to *p2* has been defined.

An example of this adaption mechanism in the Provenance context would be the following: Provenance records usually have a sink and a source. Both are complex types which usually contain a URL. By defining an adapter from strings which match the URL pattern to the complex type behind sink and source, it is now possible to use strings whenever the complex type is expected. In that case PyProtocols will automatically convert the strings to the expected complex type. This technique eases the usage of the generated code and the library.

For instance if a developer wants to use the Provenance-CSL to store messages defined by his internal data types on a Provenance store, all he has to do is to define an adapter from his data type to the corresponding P-Assertion interface of the CSL. If he wants to record something on the store now, he can send the designated information to the store by simply supplying the Provenance-CSL's recording API with instances of his datatype. By defining adapters for the return types to his own data types, he would also be able to receive his own internal data types from a Provenance store using Provenance-CSL.

[2]It also features the *wsdl2dispatch* compiler which generates Python stubs for the server side. However this was not needed by the project.

2.5 PyUnit

PyUnit [26] is a unittesting framework for Python, similar to JUnit, the standard unittesting framework for Java. PyUnit allows the creation and aggregation of unittests. It is part of the Python standard library since Python 2.1.

Unit tests are small tests which usually run a small part of some code and check if the results match an expectation. For instance unittests could check if a method returns the correct values for several inputs or if it raises the correct exception on misuse.

It is common to write unittests as a form of specification for a softwares behaviour before the software itself is written. This principle is called the *Test First* principle.

PyUnit is used to test all utility functions for correct behaviour with several correct and incorrect input parameters. It is used to test the implementation of the Provenance store service client.

3 Technical Approach

3.1 Provenance Design

Provenance is designed using a service-oriented approach. In a service-oriented architecture (SOA) clients typically invoke services, which may themselves act as clients for other services. The running of an application programmed in a SOA style requires the execution of a certain workflow (i.e. the invocation of a set of services in a certain order). The workflow of an application programmed in a SOA style (external workflow) and the workflow of a specific service (internal workflow) are two of the three different pieces of information documentation the Provenance distinguishes.

Figure 1 illustrates the scope of a Provenance system. All elements of a Provenance system will be explained in the following.

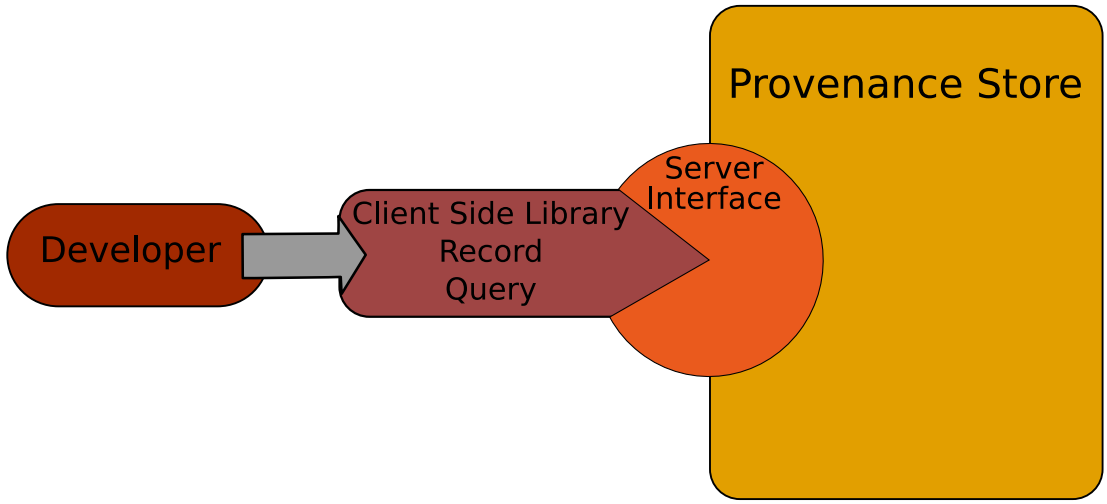

Figure 1: Scope of a Provenance system. According to: [21]

3.1.1 Provenance Store

The participants of a Provenance recording process are called actors. The software component which is storing the process documentation is called the *Provenance store*. A set of protocols for interaction between actors and Provenance Store have been defined in [15, 10]. Actors may store and query process documentation from the Provenance store using those protocols.

PReServ [19] is an implementation of a Provenance store which is currently in development and utilizes the Provenance SOAP technology binding [17]. The PReServ Provenance Store is a

web service which allows storage and querying of P-Assertions using SOAP message encapsulated web service calls. The interfaces are defined with WSDL.

3.1.2 P-Assertions

P-Assertions are the elementary unit of Provenance. A p-assertion is an assertion that is made by an actor and pertains to a process [16]. In other words, all P-Assertions pertaining to one workflow form the process documentation of the workflow. There is a distinction between three different types of P-Assertions [16].

- *Interaction P-Assertions*: An assertion of a received or sent identifiable messages contents.
- *Relationship P-Assertions*: An assertion that one message sent by the actor is the effect of another message (cause) received by the actor.
- *Actor State P-Assertions*: An assertion describing an internal message of an actor. The message may be the cause for further messages, but it may not be the effect of another message.

3.1.3 Lifecycle

The Provenance lifecycle consists out of four phases.

1. *Creation*: Actors create P-Assertions.
2. *Recording*: P-assertions are being recorded at the store.
3. *Querying*: Users or Applications Query P-Assertions from the store.
4. *Managing*: It might be needed to handle distribution and change management at the Provenance Store.

3.1.4 Client Side Library

The Provenance-CSL can be used by developers of Provenance-aware applications. It provides developers with an easy-to-use interface, that enables them to create, store and query P-Assertions. Thereby the CSL hides irrelevant details in the communication between the application and the Provenance store, thus reducing development costs of Provenance-aware applications.

The CSL implements the defined protocols [10, 15] and communicates with a Provenance store using a defined technology binding (like the SOAP binding defined in [17]).

3.2 API Specification

This section briefly describes the API of Provenance-CSL.

- **provenance** - Base package. It is only used as a collection of all the other parts.
- **provenance.api** - This package contains the package users should utilize for their application. It is a collection of all parts of the Provenance-CSL which might be useful for its users. Importing them via provenance.api results in lazy loading of those parts. Usually users should only import from provenance.api.
- **provenance.interfaces** - This package contains the interface-definitions for all types used in Provenance-CSL. Users will need this solely if they wish to define new adapters. It can be imported with *from provenance.api import interfaces* or with *from provenance.interfaces import api as interfaces*. The first version will result in a lazy loading of the package.

- **provenance.adapters** - This package contains predefined adapters for several datatypes and interfaces. It can be imported with *from provenance.api import adapters* or *from provenance.adapters import api as adapters*. Both statements are equivalent. None results in lazy loading.
- **provenance.serverAPI** - This package contains the datatypes and stubs which are generated by *wsdl2py*. Users will usually not work with this package directly. It can be imported with *from provenance.api import serverAPI* or *from provenance.serverAPI import api as serverAPI* nevertheless. The first method results in lazy loading.
- **provenance.utils** - Collection of utility functions which help with the generation of data which can be recorded on a Provenance store. If users do not create their own adapters for their data types, they should use these functions to create the correct datatypes for recording. It can be imported with *from provenance.api import utils* or *from provenance.utils import api as utils*. The first method results in lazy loading.
- **provenance.client** - Contains the implementation of the Provenance store service client, which is the interface to the Provenance store and allows to store data on it or query it from there. It can be import with *from provenance.api import client* or *from provenance.client import api as client*. The first method results in lazy loading.

3.3 Implementation

This chapter describes the current state of the implementation.

3.3.1 Client Side Library

The client side library currently supports recording of P-Assertions on a Provenance store using the Provenance protocols of version 0.25 [22]. Querying and the concept of P-Headers have not been implemented yet.

The current implementation features a complete set of utility functions for easy creation of P-Assertions and records and everything that is necessary for that.

All interfaces which are necessary for recording have been defined as well as interfaces for the result types. Several interfaces have been defined to be context sensitive (i.e. is an Endpoint used as a sink or as source).

Adapters for all wsdl2py-generated types to the appropriate recording interfaces have been defined as well as adapters for a wide range of simple Python data types (like strings, lists and dictionaries) to support several recording interfaces. Adapters for the results of recording operations to the appropriate interfaces have been defined as well.

3.3.2 Testsuite

A suite of unittests for several parts of the client side library has been developed in order to ensure functionality of the library itself. The following test modules which can be run by themselves or using a PyUnit-Testrunner have been developed.

- **provenance.utils.test**: Contains testcases for the *utils* package. The defined factory functions are tested using multiple sets of valid and invalid parameters. The return values are checked for type, protocol and the containing data. The tests also check whether invalid parameters result in the expected exceptions.

  By using parameters which must be adapted by PyProtocols, some of the adapters are tested by this package as well. Even though this is a nice feature, it does not ensure that all adapters behave in the expected way. Testcases for the adapters would be needed to achieve this as well.

  The testsuite for the *utils* covers all current utilities sufficiently[3].

[3]Since there is always something more to test, it can not be said that the testuite is complete

- **provenance.client.test**: Contains testcases for the *client* package. It is tested whether the Serviceclient class supports the required interfaces. Further tests are made for the *record* and *query* functions of the client. These tests require an actual Provenance store installation to be available and some configuration as well (the URL of the store must be provided). These tests record testdata on a given Provenance store and thereby ensure that the complete SOAP communication is fully functional.

4 Results

The current state of the library features a set of utility functions for creation of records (including all types of P-Assertions) and allows recording them on a Provenance store. All unittests for the utils package and the recording testcases for the client pacage can be run without failures and error. Querying has not been implemented so far.

Unittests have shown that it is possible to communicate with a PreServ Provenance store version 0.3 and store records on it without problems. Since it is not possible to query the recorded P-Assertions back, it was necessary to browse the stored data manually. This showed that the records have been stored on the servers. However we were not able to query them back even when we used the Java-Client Side library. This might be the case because of wrong search expressions.

5 Conclusion and Discussion

The development of the library has been quite successful this far. The basic architecture has been defined and seems to be well suited for the task. Significant parts of the protocol have already been implement and seem to work, even though there are some problems left.

The development of the library seemed much easier in the beginning because the top layers of the Protocol definitions are quite straight forward and easily comprehendable. Unfortunately this can not be said for a lot of things which are buried deep down inside the protocol definitions. Complex types are used for things which could easily be expressed using simple boolean values, the naming of data types is sometimes missleading or used multiple times with different meanings. From our point of view the protocols details are overly complex. Appart from that, the Java Client Side library, which was supposed to be used as a reference for the Python implementation, is not well documented. Because of that it was not possible to use it as a reference at all.

Moreover a new version of PReServ was released in the course of the development of the Provenance-CSL. Several modules were added and changed in the WSDL definition of the new release, so it was focused more on the adaption of the new version rather than developing new functions for an old version of PReServ.

Even though it was not possible to deliver a full featured library in the available time, the delivered library can be used as a strong basis for further development.

References

[1] Anthill - continous integration system.
http://www.anthillpro.com/html/default.html.

[2] Bazaar-ng - decentralized version control system.
http://bazaar-vcs.org/.

[3] Bugs everywhere - issue tracker.
http://www.panoramicfeedback.com/opensource/.

[4] Bugzilla - issue tracker.
http://www.bugzilla.org/.

[5] Cruisecontrol - continous integration system.
http://cruisecontrol.sourceforge.net/.

[6] Concurrent versioning system.
http://www.nongnu.org/cvs/.

[7] Damagecontrol - continous integration system.
http://opensource.thoughtworks.com/projects/damagecontrol.jsp.

[8] Eclipse.org homepage.
http://www.eclipse.org/.

[9] P. Groth, S. Jiang, S. Miles, S. Munroe, V. Tan, S. Tsasakou, and L. Moreau. An architecture for provenance systems. 2006.

[10] P. Groth, S. Miles, V. Tan, J. Ibbotson, and L. Moreau. The p-assertion recording protocol. August 2006.

[11] Java tutorials, programming concepts - interfaces.
http://java.sun.com/docs/books/tutorial/java/concepts/interface.html.

[12] Java.util.logging package guide.
http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/.

[13] Apache log4j.
http://logging.apache.org/log4j/docs/.

[14] Mantis - issue tracker.
http://www.mantisbt.org/.

[15] S. Miles, L. Moreau, P. Groth, V. Tan, S. Munroe, and S. Jiang. Provenance query protocol. August 2006.

[16] S. Munroe, P. Groth, S. Jiang, S. Miles, V. Tan, J. Ibotson, and L. Moreau. Overview of the provenance specification effort. October 2006.

[17] S. Munroe, P. Groth, S. Jiang, S. Miles, V. Tan, J. Ibotson, and L. Moreau. A soap binding for provenance p-headers. August 2006.

[18] Pyton enterprise application toolkit homepage.
http://peak.telecommunity.com/.

[19] Preserv homepage.
http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/SoftWare.

[20] Provenance-homepage.
http://twiki.gridprovenance.org/.

[21] The open provenance specification. http://www.gridprovenance.org/openSpecification.

[22] Provenance protocols version 0.25.
http://www.pasoa.org/schemas/version025/ProvenanceService.wsdl.

[23] Python logging module.
http://docs.python.org/lib/module-logging.html.

[24] Peak pyprotocols homepage.
http://peak.telecommunity.com/PyProtocols.html.

[25] Python homepage.
http://www.python.org/.

[26] Pyunit homepage.
http://pyunit.sourceforge.net/.

[27] Python webservices project homepage.
http://pywebsvcs.sourceforge.net/.

[28] The soap 1.1 specification.
http://www.w3.org/TR/soap.

[29] Subversion - version control system.
http://subversion.tigris.org/.