university of groningen / faculty of science and engineering / computing science

Bachelor Thesis

Architecture of the Provenance Holder: A Provenance Component for Collaborative Scientific Workflows

Author:
Karlo Mirkovic

Primary Supervisor:
prof. dr. D. (Dimka) Karastoyanova

Secondary Supervisor:
Dipl.-Inf. Ludwig Stage

Contents

List of Figures

1 Abstract

In the last years, scientific workflows have emerged as a fundamental abstraction for structuring and executing scientific experiments in computational environments. Due to the trustworthiness and reproducibility of information, provenance, sometimes referred to as *lineage*, has a tremendous impact on the quality and efficiency of workflows and therefore Workflow Management Systems (WfMSs). Working within scientific collaborative groups, scientists need to be able to reproduce each other's results at every step of the workflow. However, current SWfMSs offer little to no provenance information regarding the execution or adaptation of the aforementioned collaborative scientific workflows.
This project addresses the design of the low-level architecture of Provenance Holder, which is a component of the ChorSystem, a Message-based System for the life cycle management of choreographies. In other words, the project builds upon published works on Provenance Holder, and provides an architectural design of Provenance Holder in higher detail in order to address all the important technicalities that enable the component to work within the system. The Provenance Holder is a component that is incorporated as an extension to the ChorSystem. The extension will contribute the required reproducibility of results for scientists when using the ChorSystem in collaborative scientific workflows.

2 Introduction and Motivation

Information technology is revolutionizing the way many sciences are conducted. The eruption of new techniques, results, and discoveries from quickly evolving, multidisciplinary fields such as bioinformatics, biomedical informatics, cheminformatics, ecoinformatics, geoinformatics, etc. only solidifies the fact that science is at a breakthrough. However, while many efforts focus on cloud computing [2], scientists are ultimately interested in tools that bring the power of network based databases and other computational resources to the desktop, and allow them to conveniently put together and run their own scientific workflows. These scientific workflows are process networks that are typically used as "data analysis pipelines" or ones that compare observed and predicted data through simulations and other types of scientific experiments. These processes can include a wide range of components, for example, for querying databases, for data transformation and data mining steps, for execution of simulation codes on high performance computers, etc. Ideally, the scientist should be able to plug-in almost any scientific data resource and computational service into a scientific workflow, inspect and visualize data on the fly as it is computed, make parameter changes when necessary and re-run only the affected "downstream" components. Furthermore, the scientist should be able to capture sufficient metadata in the final products such that the runs of a scientific workflow, when considered as (computational) experiments themselves, help explain the results and make them *reproducible* by the computational scientist and others. Thus, a scientific workflow management system becomes a scientific problem-solving environment, tuned to a cloud-based and service-oriented infrastructure.
However, before this grand vision can become reality, a number of significant challenges have to be addressed. For example, the software needs to be simple enough to be used for the average scientist, and fast changing versions and evolving standards require that these details be hidden from the user by the scientific workflow management system.
Another set of challenges arises from the inherent complexity of scientific data. The nature of science itself mandates that results are reproducible by scientists of any domain through proven, scientific procedures. This specific area, in which scientific workflow management systems lack, is the broader focal point of this bachelor thesis.

The current state of the art (to be explored in higher detail in the *next* section) does not enable the effective reproduction of results for scientists involved in collaborative workflows.
The resolution of this problem would tremendously improve the ability for scientists coming from different domains to effectively and efficiently collaborate on scientific projects. These projects would be comprised of a much larger number, and domain pool of scientists which opens up the ability to produce scientific discoveries to pressing issues at a faster rate, therefore potentially expanding the limits of scientific capability. Enabling access to a system with the potential for such an impact holds a measure of importance which is equal to that of a resolution of any and all pressing issues that seek scientific answers.

This leads to the following main research questions:

- Which architecture for the Provenance Holder best adheres to a scientist's requirements with regards to provenance in a scientific workflow management system?
- How can the chosen architecture be successfully integrated into the ChorSystem?

These broad questions can be sub-divided into several smaller, more specific research sub-questions that, when answered, will provide all necessary knowledge to ensure the successful completion of this project:

- What are the specific scientific requirements of a workflow management system with regards to provenance?
- Where do currently available SWfMSs lack with regards to the aforementioned requirements?
- How is the architecture of the ChorSystem designed?

This project will produce the design and prototype implementation of the Provenance Holder, a component that will be implemented as an extension to the ChorSystem. This system allows for modelling, execution, and adaptation of scientific workflows and choreographies while supporting the trial-and-error manner of exploration that scientists prefer [1]. The ChorSystem, when combined with the Provenance Holder, presents an effective scientific workflow management system that serves to the benefit of the efficient and trustworthy execution of scientific workflows. Furthermore, this project will also provide a sufficiently tested mock-up system interface to serve as the initiator for the complete implementation of this system.

In the preliminary sections, this paper expands upon the current state of art with regards to scientific workflow management systems. The reader is introduced to the high-level architecture of the Provenance Holder including its internal components and operations. Following this, the implementation of the prototype interface is expanded upon in detail through the stating of practical design choices together with the reasoning behind them. Throughout the last sections, namely *Conclusions* and *Future Work*, the project is analyzed from a broader perspective with regards to the requirements mentioned above. If a requirement could not be fulfilled, these sections contain a brief analysis of the problem together with suggestions on the different possibilities of tackling the problem in future implementations of the component. The paper concludes with the *Appendices*, where snapshots of relevant code snippets can be found, followed by the list of published works used to research the broader focal point of this project.

3 State of the Art

3.1 Provenance

Provenance information is information that describes the origins and the history of data throughout its life cycle. It includes the origins of the data and the means through which the information has been modified and adapted. Such information, also called lineage, is important to many data management tasks. Historically, databases and other electronic information sources were trusted because they were under centralized control. In other words, it was assumed that trustworthy and knowledgeable people were responsible for the integrity of data in databases or repositories, however, with the development of modern science and technology, this assumption is no longer valid for online data [3]. Today, data is often made available on the Internet with no centralized control over its integrity. Particularly, data is constantly being created, copied, moved, and combined indiscriminately by a tremendously large number of people, all with different backgrounds and levels of adequacy (or lack thereof) in their respective fields. Due to the fact that information sources (or different parts of a single large source) may vary widely in terms of quality, it is essential to provide provenance and other context information which can help end users judge whether query results are trustworthy [3].

3.2 Scientific Workflow Management Systems and Provenance

There are Scientific Workflow Management Systems (SWfMSs), which already support provenance to a certain extent [4]. For example, the SWfMS *Apache Taverna* supports provenance of workflow execution by recording input and output of service calls during execution. Provenance of data is done through annotating the workflow description at design time and generating origin-annotations at run time. However, changes to workflow definitions are not captured and have to be tracked by other means. As a consequence, reproducing changes is not at all in the focus of this system [1]. Other examples of state of the art in big data provenance include Kepler, Triana, VisTrails, Pegasus, Swift, Trident and VIEW which all show unique features [5].

Kepler models a workflow as a composition of components called actors, controlled by a director. Seeing as that it is a relatively well-known SWfMS, there are several third-party extensions that provide provenance features to the Kepler system. However, no official provenance component has been released or standardized. *Triana* provides a sophisticated graphical user interface for workflow composition and modification, including grouping, editing, and zooming functions [6],[5]. *VisTrails* focuses on workflow visualization supporting provenance tracking of workflow evolution and data product derivation. *Pegasus* provides a framework that leverages artificial intelligence planning techniques to map complex scientific workflows onto distributed Grid resources. *Swift* uses a scripting language called SwiftScript to support specification of large-scale computations over a Grid. *Trident* is a scientific workflow workbench built on top of a commercial workflow system, Windows Workflow Foundation (WF) included in the Windows operating system. *VIEW* system features efficient provenance management based on RDFProv that combines advantages of Semantic Web technologies with relational databases [5].

Each of the SWfMSs provides a platform to support individual scientists in composing workflows from various resources. Some systems show some collaboration features, in the sense that they allow a scientist to compose a workflow from shared resources and services. However, they provide limited support for multiple scientists to collaboratively compose a shared workflow. [5].

Since reproducing adaptations is not in the focus of currently available SWfMSs and changes on workflow are essential for collaborative scientific workflows, it is necessary to establish a way to record information about the execution of workflows and their activities, as well as their adaptations [1]. This would enable the scientists working on these collaborative projects to reproduce every step taken and every adaptation made, all while abstracting the trust factor amongst them. One prerequisite here is to ensure that all adaptation steps are coordinated among all participating partners, regardless of the level of trust they feel towards each other [5].

3.3 ChorSystem

The system is a realization of the Model-as-You-Go approach, which enables users to model and execute workflows in an iterative process that eventually results in a complete scientific workflow [7]. This approach is discussed in higher detail in previous research conducted by the designers of the ChorSystem [8]. The article motivates the need for the capability to, partially or completely, repeat the logic in a choreography instance with a clear focus on the eScience community. This need stems from the fact that scientific workflows are often not completely known prior to execution and hence designed in an explorative, iterative fashion. This originates from scientists knowing the goal of their experiments but usually lacking the means of achieving it. Consequently, they rather approach their goal in a trial-and-error manner. This can be seen in, for example, scientists trying different values for and combinations of experiment parameters, substituting different solvers for equations, or the need to repeat steps of an experiment. Additionally, incidents might make it necessary to change the approach towards a goal during workflow run time by, for example, not having access to available services or receiving unexpected scientific results. The article [8] presents a formal model which describes choreography models and instances while also considering loops and multiple instances of a particular participant. In using this approach, one has to distinguish between iteration, which executes logic again without undoing already completed work, and re-execution, which aims at the compensation of already completed work before executing it again. The importance of the aforementioned distinction becomes clear once a scientific workflow management system wants to enable storing of provenance information. The distinction needs to be made in order to guarantee the validity of the data. Otherwise, if a user executes already executed logic yet again with new data and, as a consequence, previous data is wiped, enabling provenance on such a system becomes impossible. A scientist would not be able to validate the data they are using which makes the existence of such a system futile.

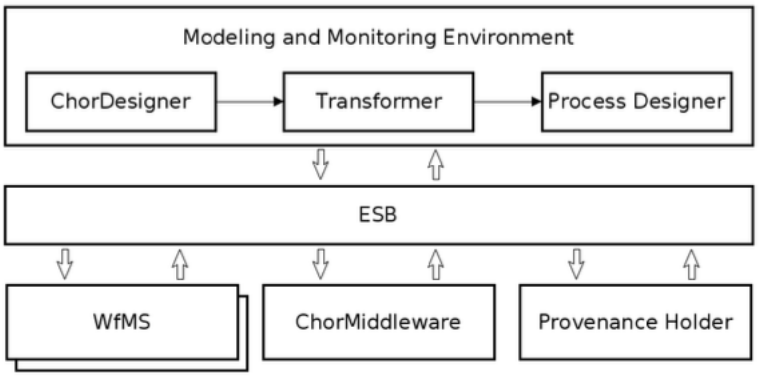

Figure 1: An architectural overview of the ChorSystem [1]

As depicted in Figure 1, the ChorSystem's architecture persists of a modeling and monitoring environment, WfMS, middleware which coordinates choreography adaptations (ChorMiddleware), and Provenance Holder component. The modeling and monitoring environment is composed of three components, namely the ChorDesigner, the Transformer, and the Process Designer. It serves as a monitoring tool while enabling modeling of collaborative experiments, generation of visible interfaces, and redefinition of workflow logic. The system contains a WfMS (workflow management system) which supports execution, adaptions and monitoring of the workflows and choreographies coupled with the Provenance Holder component that enables the storage and retrieval of provenance information. All components are provided as services and interact with each other through a service middleware, the ESB (Enterprise Service Bus). [1]. Up to this point, the primary focus of this section was to emphasize the lack of provenance information in currently available scientific workflow management systems. However, it is important to note that the ChorSystem is more than a sWfMS. With the current state of research conducted on the system, a choreography is comprised of multiple workflows in linear order. Seeing as the architecture of the system contains a workflow management system as a component coupled with a middleware that coordinates choreography adaptations, it becomes clear that the ChorSystem far exceeds the capabilities of a mere sWfMS. Consequently, even though throughout this document, I put the ChorSystem in context with sWfMSs, I do so only from the perspective of the Provenance Holder and am aware of the distinction.
With this project I contribute the low-level architecture of a system that supports the collaborative adaptations in experiments while, without third parties, allowing for the interchangeable use of different technologies to support trustworthiness through aforementioned Provenance Holder component.

4 Architecture and Design

The collection of all indispensable information to guarantee the origin, reproducibility and trustworthiness in collaborative adaptations, is the responsibility of the Provenance Holder component. The component's initial implementation comes within the context of the ChorSystem, however, a crucial design approach has been taken towards this system which is important to note. The design approach to this solution is generic, reusable across different scenarios and is less intrusive due to the separation of concerns principle that was followed during its high-level architectural design in [1].

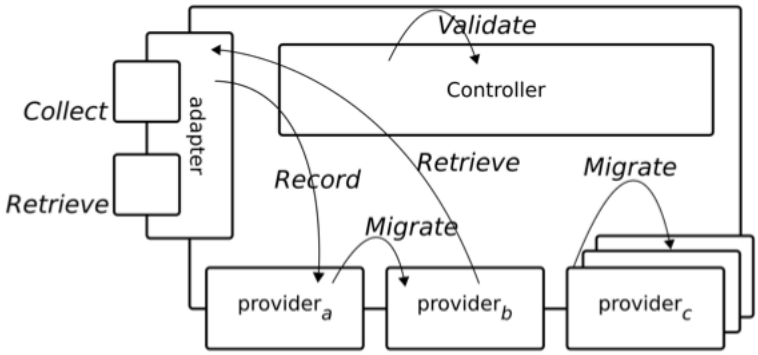

Figure 2: The high-level architecture of the Provenance Holder including its methods [1]

As depicted in *Figure 2*, the Provenance Holder component is a separate service responsible to ensure the aforementioned necessities with regards to trust and provenance in adaptive collaborations and is not part of any other functional component of the architecture it is linked to. In the previously referred to article, two main operations were identified for the component's interface, which include *collecting* and *retrieving* provenance data. These can be realized by four interaction scenarios, also called *methods*, carried out by its components. The components of Provenance Holder are the *adapter*, the *controller* and one or more provenance *providers*. The implementation of the components and their methods are explained in thoroughly in the *Implementation* section.

5 Implementation

The implementation of the mock-up Provenance Holder interface was done in Python combined with several libraries and frameworks. Apart from specifying the high-level architectural design, previous work motivated the need of digital signature verification [1]. For this, we chose Ed25519 due to several benefits that it offers compared to its counterparts. Today, the RSA is the most widely used public-key algorithm for SSH key. However, even though some RSA-type systems provide faster verification, this advantage decreases with the increase of the level of security. Moreover, for many applications, the advantage is outweighed by much slower signatures and significantly larger keys. Another benefit in using Ed25519 is the compactness of its public-key. It only contains 64 characters, compared to RSA 3072 and its 544 characters. Furthermore, generating the key is almost as fast as the signing process. Fast batch signature verification and collision resilience are also among the list of benefits of using Ed25519 [9]. The aforementioned advantages in combination with the simplicity of the Ed25519 Python library were the key factors influencing the decision for Ed25519 over other digital signature verification schemes.

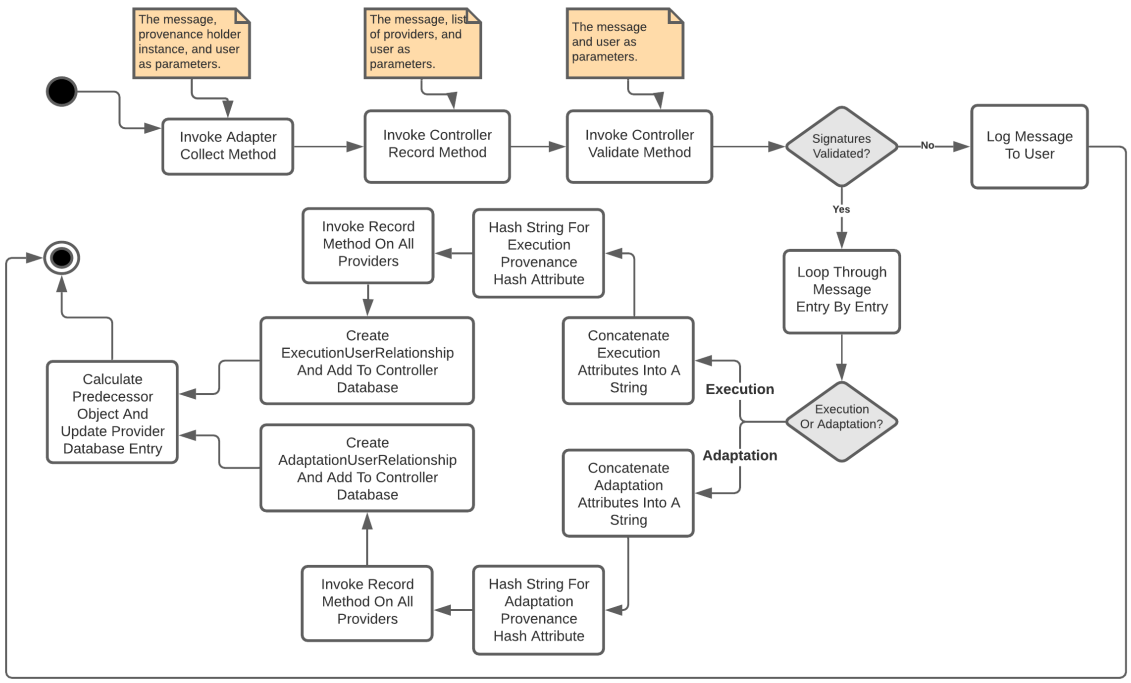

Figure 3: UML Activity Diagram of the collect operation

For the management of the database, we decided upon combining SQLALchemy with Python. The reasoning behind this is expanded upon in the *Database* section. Additionally, the Provenance Holder is meant to be connected, through the adapter component, to ActiveMQ, an open source message broker written in Java together with a full Java Message Service client. ActiveMQ will be incorporated into the system through the use of Docker. In the future, the message server will connect to the Provenance Holder via the adapter component, which will await messages from the bus (see *Section 5.2*). Since the adapter can only receive two types of messages from the scientific workflow management system it is integrated into (see *Adapter* section), there are two possible operations that are invoked, namely `collect` and `retrieve`. *Figure 3* shows the the collect operation in a UML activity diagram while *Figure 4* shows the retrieve operation.

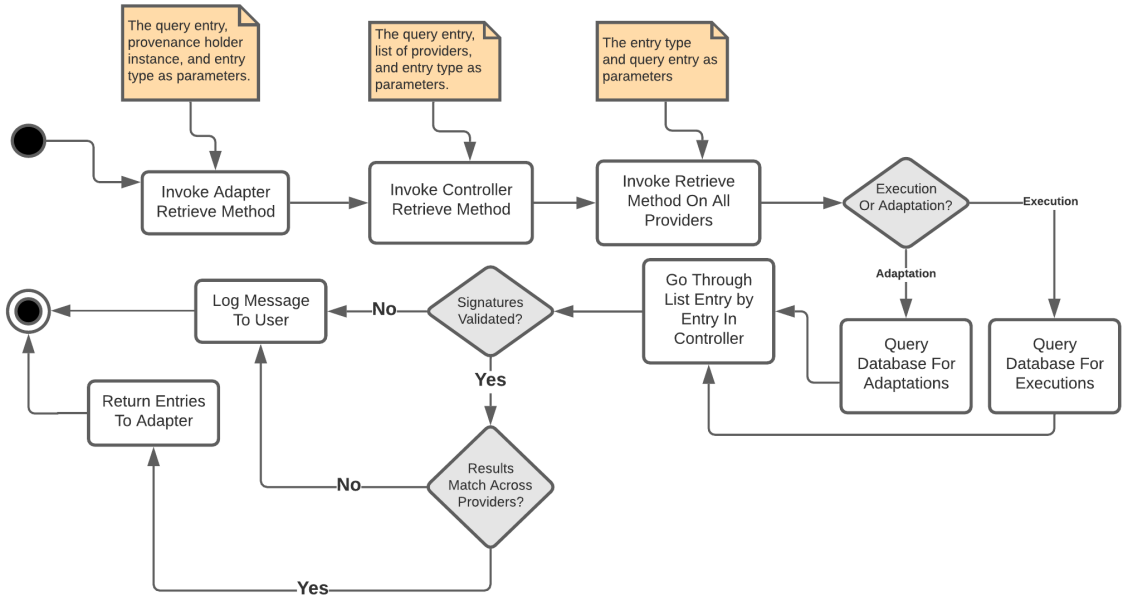

Figure 4: UML Activity Diagram of the retrieve operation

5.1 Database

The Provenance Holder's architectural design was created in a way that requires the component to run with several providers (possibly simultaneously). Given the fact that the development approach taken is of a generic nature, the implementation of the component adheres to this requirement. Furthermore, during the development of the Provenance Holder interface, in order to fulfill the aforementioned requirement, a generic and object oriented way of programming was taken which enables the component to not only be linked with the ChorSystem but any arbitrary scientific workflow management system in the future.
For the mock-up interface, the Provenance Holder is only composed of one provider, however, it is designed in such a way that the component is fully functional with an arbitrary number of providers. Even though the prototype was tested and is fully functional with multiple providers, it was only tested with providers of the same type (SQL database providers). Since storing the same data in the same format multiple times provides no additional functionality or benefit to the component, a decision was made to implement the provenance holder containing only one provider. The chosen single provider contains an SQL database that was developed with PostgreSQL and SQLAlchemy as the object relational mapper.

Using Python in combination with SQLAlchemy has the benefit of being able to write Python code in the project to map from the provider's database schema to the application's Python objects. Since this mapping allowed SQLAlchemy to handle the underlying database, no pure SQL was required to create, query, and maintain the database. This enabled the implementation to have accented the object oriented aspect of the development rather than having to write bridge code to get in and out of relational tables.

The SQL database on the provider consists of two types of database entries, namely *Execution*, and *Adaptation* entries. Both entry types along with their attributes can be seen in *Figure 5*, and are explained in further detail in the following sections.

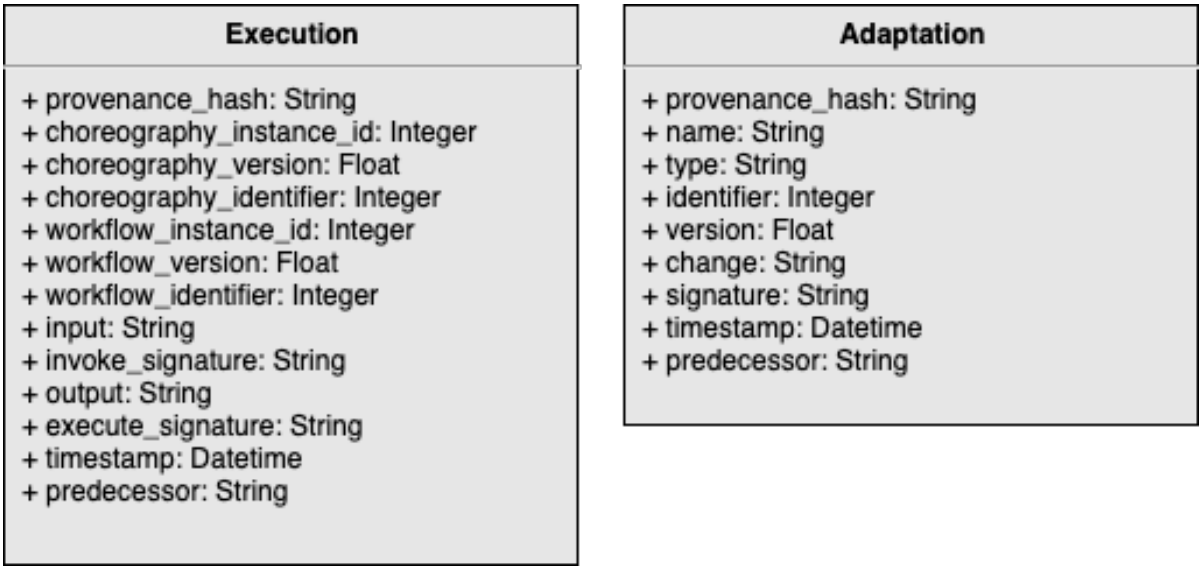

Figure 5: A UML Class Diagram of the SQL database stored on the provider

5.1.1 Execution

As shown in *Figure 5*, the execution database entry contains the most provenance information out of the database entry types. Its primary key, the `provenance_hash` attribute, is a SHA256 hash of the concatenation of the rest of the attributes apart from the `predecessor`. Next, it contains `instance_id`, `version`, and `identifier` attributes for both the choreography and the workflow of the entry respectively. Furthermore, it contains the input and output strings of the execution, `invoke` and `execute` signatures, a timestamp, and a `predecessor` object. The `invoke` and `execute` signatures are attributes that allow the provenance holder to maintain the integrity of the data. The `invoke_signature` is a digitally signed string that is comprised of the concatenation of the following attributes:

- `choreography_instance_id`
- `choreography_version`
- `choreography_identifier`
- `workflow_instance_id`
- `workflow_version`
- `workflow_identifier`
- `input`

Meanwhile, the `execute_signature` is comprised of the concatenation of the following attributes:

- `invoke_signature`
- `output`

Further, the execution entry contains `output`, `timestamp`, and `predecessor` attributes. The predecessor attribute of an execution entry is the `provenance_hash` of its predecessor object. For an execution, the predecessor object is defined as the latest entry to the database with a matching `choreography_instance_id`.

5.1.2 Adaptation

With an adaptation entry changes to the workflow or choreography model are captured. As a result, an adaptation contains no instance information in its attributes. Changes/adaptations to the models are separate from the execution data and need to be recorded as such. Similar to the execution entry, an adaptation entry's private key is a SHA256 hash of all of its attributes excluding the `predecessor` object. The other attributes include:

- `name`
- `type`
- `identifier`
- `version`
- `change`
- `signature`
- `timestamp`
- `predecessor`

The three attributes that are worth noting are `change`, `signature`, and `predecessor`. The signature attribute is a digitally signed string comprised of the concatenation of name, version, and change. Its predecessor is, again, a SHA256 hash of its predecessor object's `provenance_hash`. Even though the change attribute is a topic of future research, considering how the system is planned to be built in the future, a decision was made to set the change attribute to a string. The change attribute will most likely, similarly to git, be a SHA256 hash of a diff operation.

5.2 Adapter

The adapter is the component that enables communication between the Provenance Holder and any generic sWfMS. In the context of the ChorSystem, the adapter is linked to the Enterprise Service Bus (ESB) and awaits messages. In this implementation, the adapter component incorporates the STOMP protocol in combination with ActiveMQ in order to send and receive messages. It is subscribed to a message queue that receives two types of messages, namely ones that invoke calls to the collect and retrieve operations. The adapter component contains a `read_message` and `parse_message` methods. The former is invoked upon the receival of a message from the bus. It checks the type of message and invokes the `parse_message` method which parses the message into data that can be used accordingly by the controller. Due to the current state of the research on the ChorSystem, the `parse_message` method is not implemented in the prototype. This is due to the fact that it has yet to be decided what format the messages from the bus will arrive in (see section Future Work). Upon completion, the method returns a parsed string that is then used to invoke the controller `retrieve` or `record` methods. In the case of the former, the adapter awaits messages that contain data whose provenance information can be stored, whereas in the latter case it is subscribed to receive types of messages that require a retrieval of provenance information.

As a consequence, the adapter component only contains two of the four operations of the Provenance Holder, namely the retrieve and collect operations. Upon the invocation of either of these methods, the adapter sends data to the controller. Depending upon whether the retrieve or the collect method was invoked, the adapter either waits for the controller to return data or simply continues awaiting for messages from the ESB respectively.

5.3 Controller

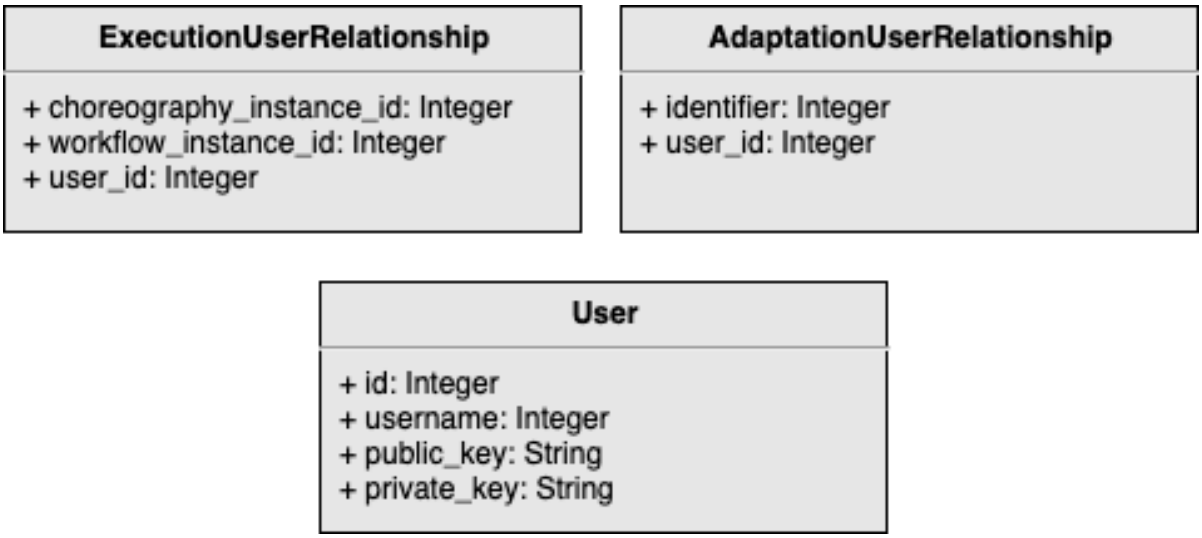

Figure 6: A UML Diagram of the controller database data types

The controller component is the most active component amongst them. It is connected to the adapter, and all providers that a Provenance Holder instance may contain. The component implements all four methods of the Provenance Holder, and is the only component that controls the digital signature validation. Therefore, the controller component is has it's own database that is separate to those of the providers. In this database (see *Figure 6*), the controller stores information about the relationships between the users and the provider database entries. In other words, the controller stores the `user_id` of the user that signed the data that is being processed, together with the respective identifiers of the provenance data entries. They are stored into the `ExecutionUserRelationship` and `AdaptationUserRelationship` classes. The purpose of this database is to enable the identification of the signee during the process of validation.

As depicted in *Figure 6*, the controller database also contains a *User* entity. It is important to note that this entity is only incorporated into the implementation for testing purposes and is only necessary to enable the digital signing of dummy data. In a finalized implementation, only public key entities would be stored by the component.

The controller class contains two sub-classes, namely `Execution` and `Adaptation`. The classes are only used in the controller and their purpose is to enable the controller to convert data passed to it into data that can be received by the providers. The `Execution` and `Adaptation` sub-classes contain the same attributes as their respective provider database entries, and are only used when data is being recorded.
The following subsections explain how the controller handles all its methods apart from one, the *migrate* method. Since the intention of the designers of the Provenance Holder is to incorporate blockchain technology into the providers, the migrate operation/method is beyond the scope of the implementation of this project. The operation is expanded upon in the *Migrate* section.

5.3.1 **Retrieve**

In order to query the provider for database entries, the controller needs to know several things. In theory, all providers are meant to essentially store the same data, their difference lies in the form that they store it in. The controller needs to be aware of this list of providers, and is therefore handed that list as a parameter upon invocation. Further, the controller is sent a database entry where all the attributes of the entry are set to `None` except for the attributes that the controller is querying on. For clarification purposes, in the rest of this document we shall call this entry the *query entry*. The query entry can have any number of not `None` attributes that will be used to query the database. In other words, the query entry attributes very closely resemble a filtering system of search results one would use on, for example, apartment renting websites. The choice is there to use the attributes to query if the user wishes so, however, it is not necessary to fill in more than one attribute. Along with the query entry, the retrieve method is also given an `entry_type` parameter that it uses to distinguish between the database entry types. The controller goes through the providers one by one and invokes the provider's `retrieve` method to return a message that is being queried for, and begins the process of validation and comparison before retrieving the data, if any, to the adapter. Once the data is validated, the controller invokes the `compare_results` method in order to check the data across all providers for consistency. This extra check is done in order to prevent tampering while the data is stored in the providers. If the comparison check fails, the user is made aware that the data has been tampered with, and that they shall only receive an empty list as a query result.
For now, if the data has been tampered with, the provenance holder isn't capable of recognizing exactly which of the providers of the component have been tampered with. However, for future work on this component, identifying the provider with the broken data from the list of providers is a functionality that can prove to be useful. It would enable the component to, in the case of data tampering, clean out its data without the need for maintenance.
Moving further, for clarification, it is important to note that a message is a list of multiple entries. Therefore, an invocation of the retrieve method can have two types of returns. Like the aforementioned case, one of the types of returns is an empty list. An empty list is a result of either a query in which the data could not be validated and compared successfully (in which case the user is made aware of the issue through a log message), or the database simply does not contain entries with the specified attributes. The other type of return is the trivial one that contains the list of entries that were queried.

For the validation process, the data from the provider is converted into a format that is readable by the validate method (see next section). Further, the controller searches its database using the attribute provided in the query entry in order to identify the user that originally signed the data. The controller then goes through the message, and entry by entry invokes its own `validate` method which returns a list of validated entries in the message. The controller then converts the message entries back into data that is usable by the adapter and returns the message to the adapter.

5.3.2 **Validate**

The validate method only takes two parameters. It receives the list of entries (the message composed of entries) that need validation, and the user that signed the data when the data was being recorded. It goes through the message, entry by entry, and recreates the invoke and execute strings using the data in the message. This is due to the fact that the `invoke_signature` and `execute_signature` attributes in executions `signature` in adaptations) is a signed string composed of several concatenated attributes (see *Execution* section). Once the strings have been recreated, the entries are validated using ed25519's `verify` method (see Appendix A).

It is important to note that, in our mock up implementation, prior to validation, the method checks for the type of entry it is validating. This is the case merely due to the process of recreating the signatures (they differ across different database entry types). If an entry is validated, it is added to a list of validated entries that, once the method has iterated over all the entries in the message, is returned to where the method was invoked from.

5.3.3 **Record**

In order to record, the controller must be aware of the list of providers, the user that carried out the execution or adaptation, and the message that needs to be recorded. The controller begins by validating the message. After this, similarly to the other methods, the record method goes through the message entry by entry. It checks for the type of entry through the data stored in the message, and proceeds to store the data accordingly. There are several important design choices to note here. Firstly, in our implementation, we have alleviated the burden of calculating the provenance hash and the predecessor object for every provider separately. Initially, the controller sends the data for the providers to store with the `provenance_hash` and `predecessor` attributes set to `None`. Following this, entry by entry, the controller calculates the provenance hash of the entry and determines the predecessor object, if any, by calling the SQL provider's retrieve method with adequate parameters. Once the attributes are determined, the already stored entries are updated in the database. The code snippets for these calculations can be found in Appendix B. It is also important to note that, before sending the data to the providers to record, an entry-user relationship (see Figure 4) is created and added to the controller's database.

5.3.4 **Migrate**

Even though the migrate method is not implemented in the mock up interface and is beyond the scope of this project, this section describes the behavior of such a method for future work.

In case of the need to transfer data to a new type or instance of storage, the migrate method is used when an addition or change of instances is required. This method enables the simultaneous retrieval of the stored provenance information from a provider component. On the one hand, when speaking of addition of storage instances of the migrate method, an expansion of the storage is conducted through a new instance of an either already existing or not yet used technology. This can be seen in, for example, data within an SQL database that is being saved in another instance, either as a copy of the previous or now as additional instance in the form of e.g. a flat file. On the other hand, a change of storage describes the replacement of one storage instance by a different one, either within the already used storage technology or through replacing the existing technology with another. Within this process, the used technology impacts the complexity of such a migration due to the given differences in characteristics and features.

6 Conclusions

The nature of this document is one of a proof of concept project. The paper focuses on the implementation and design choices of the Provenance Holder component in order to demonstrate its feasibility. The implementation of the prototype focuses on fulfilling the requirements of scientists in the field of eScience addressed in previous papers introducing the high-level architecture of the component. Previous work on the Provenance Holder provided a design of a separate software component responsible for and capable of storing provenance data in a trusted manner while being able to support the reproducibility of experiments based on the provenance data. In this document, I have presented a detailed implementation of a prototype interface for the component therefore demonstrating the fulfillment of most requirements in the scope of this project. In order to help visualise the more verbose architectural design of the Provenance Holder's components and their interactions, *Figure 7* shows a UML component diagram of the prototype implementation.

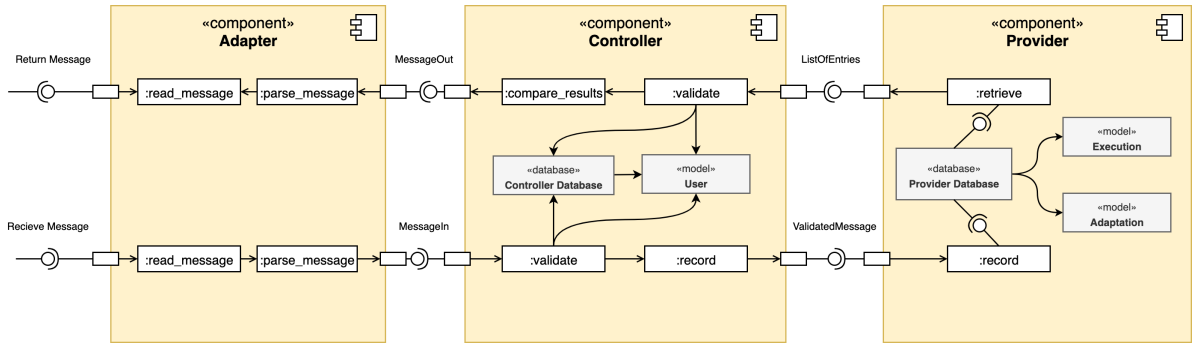

Figure 7: A UML Component Diagram of the prototype

The implementation of the component is capable of successfully storing and retrieving provenance information. Furthermore, through the use of digital signature verification and data comparison amongst providers, the storing and retrieval of data is implemented in a manner that enables the necessary trustability of results. There are requirements addressed in previous work that are beyond the scope of this project (see the next section) that are a topic for future improvements of this prototype. One of them being the implementation (or lack thereof) of a migrate operation. However, despite the lack of such an operation, the proposed implementation still adheres to the reproducibility and trust requirements of such a component.

Looking back at the research questions mentioned in the beginning of this document it becomes clear that future research is necessary to answer some of them. The first main research question is the focal point of this project and is therefore directly tackled by the implementation. The architecture chosen is a generic one, in theory adaptable to any scientific workflow management system. This allows the scientists to choose whatever system they are most comfortable with. With regards to the latter main research question however, future research and progress needs to be made in the context of the implementation of the ChorSystem (information that I do not have access to) in order to integrate the component into the system. The prototype presented in this paper is a standalone component with a single SQL provider and a skeleton of an adapter that is connected to an ActiveMQ message bus.

The rest of the research questions were all necessary to answer in order to grasp the bigger picture of the ChorSystem. I addressed those questions in the *State of the Art* section of this paper and used the information from their answers in order to correctly place the Provenance Holder within the context of the plans for future improvements and research.

7 Future Work

My implementation of the Provenance Holder, to the extent possible given time restrictions and complexity level of a bachelor project, adheres to the reproducibility and trust requirements such a component. However, there are several safety concerns that should be addressed in future implementations. Firstly, the component needs an encryption of the data connections in order to send information safely. Seeing as that this work does not have encryptions of the sort, it is less safe than the optimal solution. Unfortunately, acquiring ssl certificates was not possible for the prototype implementation of the Provenance Holder and it is therefore a suggested topic of future research.
Another security issue arises once the possibility of data tampering is considered. Once the retrieve operation is invoked, data is validated and compared in order to ensure integrity (see *Retrieve* section). If the comparison test fails, that means that at least one (possibly multiple) provider was compromised. The issue that arises here is that, in the current implementation, the provenance holder is not aware of how many and which of the providers is compromised. Since the component does not return any data upon the failure of the comparison of the results, this case puts the component in a denial of service (DOS) state. The component would then require maintenance in order to pinpoint what data was tampered with and remove it from the system. For future works on the component, enabling the detection of which providers were tampered with would resolve all the problems mentioned above.

As briefly touched upon in the *Adapter* section, the adapter component lacks the ability to parse the data from a message into data that is usable by the controller. This is due to the current state of research conducted w.r.t. the implementation of the ChorSystem. The format of the messages the adapter will receive is still to be decided upon, e.g. flat text file. The parsing of the messages in the adapter into usable data is therefore a topic of future research.

Further, as explained in the *Database* section, my implementation only consists of one provider. Even though the prototype was tested and works with multiple providers, when testing, the providers were all of the same type (SQL database). For future works on the topic, the interface should be designed with multiple providers that store data in different formats. This would immediately open up the opportunity for the developer to start working on the migrate operation.

Another topic that is to be addressed in the future is the idea of implementing the providers using blockchain technology. Due to the nature of blockchain, that implementation of the component would tremendously improve the Provenance Holder's security. Furthermore, migration of one storage technology to another poses complexity implications and in particular what concerns the block-chain technology and its characteristics that need to be paid special attention when it is employed in a provenance provider component. It is important to note that the cost of the migrate method must also be considered, especially when it comes to blockchain technology where the needed information can be spread virtually over the whole ledger. The migration step might also involve purging data from the source after a successful data transfer. Here, block-chain technology might also pose additional challenges. There are implications on the cost and complexity of such a migration [1].

8 Acknowledgements

I would like to sincerely thank my supervisors prof. dr. D. (Dimka) Karastoyanova and
Dipl.-Inf. Ludwig Stage for the constant feedback and support throughout the duration of this project. They made this project a pleasurable experience and I definitely couldn't have done it without their support.

Appendices

A Reconstructing Strings and keys for Validation

```
# The piece of data at entry[7] is the invoke_signature
# The piece of data at entry[9] is the execute_signature
invoke = str(entry[0]) + \
         str(entry[1]) + \
         str(entry[2]) + \
         str(entry[3]) + \
         str(entry[4]) + \
         str(entry[5]) + \
         str(entry[6])
invoke = bytes(invoke, 'utf-8')
execute = str(entry[7]) + entry[8]
execute = bytes(execute, 'utf-8')

# Reconstruct the public key using the ed25519 constructor and validate
public_key = ed25519.VerifyingKey(ed25519.from_ascii(user.public_key, encoding='hex'))
try:
    public_key.verify(entry[7], invoke, encoding='hex')
    public_key.verify(entry[9], execute, encoding='hex')
    new_message.append(entry)
    print("Validated signature of execution with workflow_id: " + str(entry[3]))
except:
    print("Could not validate signature of user " + user.username)
```

```
# The piece of data at entry[5] is the signature
# Concatenate the data in order to recreate the original data that was signed
sig_msg = entry[0] + str(entry[3]) + entry[4]
sig_msg = bytes(sig_msg, 'utf-8')

# Reconstruct the public key using the ed25519 constructor and validate
public_key = ed25519.VerifyingKey(ed25519.from_ascii(user.public_key, encoding='hex'))
try:
    public_key.verify(entry[5], sig_msg, encoding='hex')
    new_message.append(entry)
    print("Validated signature of adaptation with identifier: " + str(entry[2]))
except:
    print("Could not validate signature of user " + user.username)
```

B Calculating the predecessor object for Executions and Adaptations

```
# Use the overloaded retrieve method to retrieve all executions
# The method returns a list of executions in decreasing order by timestamp
# The first entry in the list with the same choreography id is the predecessor
executions = providers[0].retrieve('execution')

# Retrieve the entry from the database in order to set its predecessor correctly
temp = Execution()
temp.workflow_instance_id = new_entry.workflow_instance_id
db_entries = providers[0].retrieve('execution', temp)

# Iterate over the list of executions in order to find an entry that has the same
# choreography instance id as the entry being recorded
for execution in executions:
    for db_entry in db_entries:
        # Make sure that the provenance hashes don't match
        # so we don't set an object as its own predecessor
        if execution.choreography_instance_id == db_entry.choreography_instance_id \
                and execution.provenance_hash != new_entry.provenance_hash:
            db_entry.predecessor = execution.provenance_hash
provenance_session.commit()
```

```
# Use the overloaded retrieve method to retrieve all adaptations
# The method returns a list of adaptations in decreasing order by timestamp
# The first entry in the list is the predecessor
adaptations = providers[0].retrieve('adaptation')

# Retrieve the entry from the database in order to set its predecessor correctly
temp = Adaptation()
temp.identifier = new_entry.identifier
db_entries = providers[0].retrieve('adaptation', temp)
for db_entry in db_entries:
    if adaptations.count() > 1:
        db_entry.predecessor = adaptations[1].provenance_hash
    provenance_session.commit()
```

References

[1] Dimka Karastoyanova and Ludwig Stage. Provenance holder: Bringing provenance, reproducibility and trust to flexible scientificworkflows and choreographies. In *Proceedings of Second Workshop on Security and Privacy-enhanced Business Process Management at BPM 2019*. IEEE, September 2019.

[2] Kefeng Deng, Kaijun Ren, Junqiang Song, Dong Yuan, Yang Xiang, and Jinjun Chen. A clustering based coscheduling strategy for efficient scientific workflow execution in cloud computing. *Concurrency and Computation: Practice and Experience*, 25(18):2523–2539, 2013.

[3] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2007.

[4] M. Herschel, R. Diestelkmper, and H Ben Lahmar. A survey on provenance - what for? what form? what from? *International Journal on Very Large Data Bases (VLDBJournal)*, 2017.

[5] Jia Zhang, Daniel Kuc, and Shiyong Lu. Confucius: A tool supporting collaborative scientific workflow composition. 2014.

[6] Ludäscher Bertram, Altintas Ilkay, Berkley Chad, Higgins Dan, Jaeger Efrat, Jones Matthew, Lee Edward A., Tao Jing, and Zhao Yang. Scientific workflow management and the kepler system. 2005.

[7] Mirko Sonntag and Dimka Karastoyanova. Model-as-you-go: An approach for an advanced infrastructure for scientific workflows. *Journal of Grid Computing*, 11(3):553–583, 2013.

[8] A. Weiss, V. Andrikopoulos, M. Hahn, and D. Karastoyanova. Model-as-you-go for choreographies: Rewinding and repeating scientific choreographies. *IEEE Transactions on Services Computing*, pages 1–1, 2017.

[9] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. September 2012.