Linked Compute Units and Linked Experiments: Using Topology and Orchestration Technology for Flexible Support of Scientific Applications

Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
leymann@iaas.uni-stuttgart.de

Linked Compute Units and Linked Experiments: Using Topology and Orchestration Technology for Flexible Support of Scientific Applications

Frank Leymann
Institute of Architecture of Application Systems (IAAS)
Universität Stuttgart
Leymann@iaas.uni-stuttgart.de

Abstract: Being able to run and manage applications in different environments (especially in clouds) is an urgent requirement in industry. Such portability requires a standard language to define both, the structure of an application as well as its management behavior. This paper sketches the main ingredients of such a language and explains its underlying concepts. Next, the concept of linked compute units is introduced providing verifiability of the results of data-intense work. Considering human beings in this concept results in linked social compute units. The benefits of describing scientific applications by this concept are worked out. The resulting vision of being able to run in silico experiments everywhere and its supporting high-level architecture is presented.

1 Introduction

The extensive focus on data is a major shift in science [HTT09]. This data must be shared and often connected with other data sources in order to allow reproducibility of research results [HB11]. In case data has been produced by software (like simulation software, for example) making this software (publicly) available and linking it to this data and also to the input data sources will significantly increase the reproducibility of research results. For this purpose, this software and all its prerequisites (i.e. the overall *application*) must be defined accordingly.

The definition of an application can be done base on a standard specification called TOSCA (Topology and Orchestration Specification for Cloud Applications) [TOSCA]. The specification not only allows defining the structure of an application (its so-called *topology*) but also its management behavior (by means of so-called *plans*). Plans are workflows (a.k.a. *orchestrations* in the SOA domain) that realize functionality like (i) provisioning an application, i.e. installing, deploying and configuring all of its associated software components and setting up its infrastructure, (ii) updating selective components, or (iii) moving data associated with the application or some of its components. The topology, its associated plans, and the artifacts required to actually run the overall software is included in a corresponding *package*. Thus, a package not only contains the software of the actual business logic of interest, but also all prerequisite software (or links to this software, respectively).

In cases in which data has been produced or has to be analyzed by software, linking the package of this software to the data extends the model of linked data with infrastructure elements required to produce or verify the data; we call this extended model *linked compute units*. Especially in data-intense science linked compute units provide a significant step forward towards reproducibility of scientific results. Linked compute units will also contribute to significantly increase trust in research results that are based on data and simulations. By also considering humans involved we present the concept of linked experiments that additionally support folding in skills etc required to solve a data- or compute-intense scientific problem.

2 TOSCA Conceptual Overview

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a language to define both, the overall structure of an application as well as its management behavior in a format called *service template* (see Figure 1).

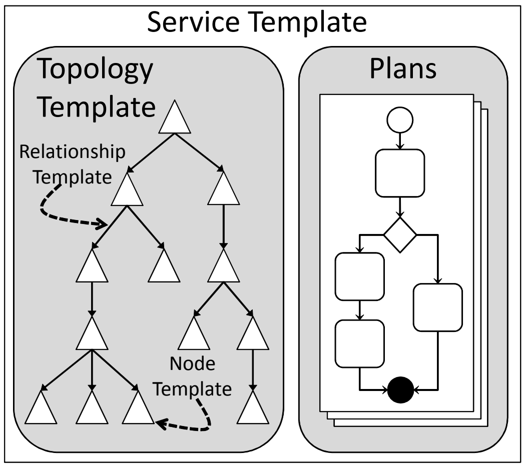

Figure 1 - Structure of a Service Definition

The overall structure of an application is defined as a *topology template*. A topology template describes all the kinds of ingredients of an application (so-called *node templates)* as well as the kinds of relations between these ingredients (so-called *relationship templates*). Node templates are considered as vertices and relationship templates as edges of an acyclic directed graph that make up the topology of an application. Examples of node templates are programs encoding the logic of a proper application, a JEE container for a program (that is offered as an EJB), a database system required by the program as persistent store, data definition scripts to set up the database structure of the program in the database system, and an operating system for the JEE container as well as for the database system. Examples of relationship templates are `deployed_in` specifying that a program is deployed in a JEE container, or `hosted_by` specifying that a JEE container is hosted by an operating system. Both, node templates as well as relationship templates are typed by referring to corresponding node types and relationship types; we ignore this indirection here for the sake of conciseness.

Node templates are descriptions of the ingredients of an application, but they are not the actual ingredients themselves. Basically, a node template defines the potential properties of an actual ingredient as well as the operations that can be used to manage the ingredient. In order to materialize such an actual ingredient the corresponding node template must be instantiated. When instantiating a node template its properties get values assigned. For example, a node template representing a JEE container has an `IP_Address` property; when instantiating the node template the corresponding JEE container receives a concrete IP address. Also, the cardinality of a node template within a topology template is defined. This cardinality controls the minimum and maximum number of instances of that node template in an instance of the topology template. For example, a service template may define that its included JEE container has a cardinality of 2 to 5 instances. Instantiating a service template means to set the cardinalities of the node templates of its topology template, and to set the properties of the instances of these node templates.

Node templates have an interface that describes the operations made available by a node template for the purpose of managing its instances. For example, the JEE container node template has operations for starting and stopping its instances, for deploying a program on them and so on. These operations can be invoked by tasks of plans (see Figure 2) of a containing service template for managing the instances of its node templates individually, and, thus, for managing corresponding applications (i.e. instances of a topology template) as a whole.

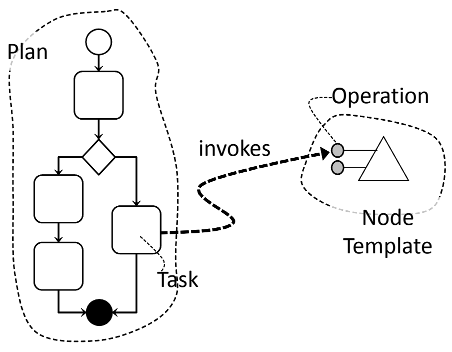

Figure 2 - Relation Between Tasks and Operations

A *plan* is a workflow that executes in the context of an instance of a service template, e.g. it has access to the actual property values of the instances of the node templates of the topology template. More precisely, the tasks that make up a plan use the operations of the interfaces of the node templates of the topology template. Such a task passes input to an operation, thus triggering actions on an instance of a node template. On completion of the operation it passes its output back to the task, such that this data is available for further tasks of the plan. For example, a task may use the `deploy()` operation of the

JEE container node template passing as input both, the identifier of the concrete JEE container on which an EJB should be deployed as well as the EJB itself; the `deploy()` operation returns the actual state of the deployed EJB.

In order to create an instance of an application described by a service template a special plan called *build plan* must be run. The build plan orchestrates the actions required to create the instances of the node templates by invoking proper operations, setting values of the properties of the node templates by passing the actual values as parameters of the operations, running tasks repetitively in order to reflect the cardinalities specified for certain node templates in a topology template etc. In general, the values to be passed to the operations may come from various sources: for example, some values may be contained in the input message starting the build plan, or the build plan itself may contain tasks that interact with human beings requesting corresponding values, respectively. Once the build plan completes successfully, the application is provisioned and can be used. While the application is provisioned several maintenance actions might be required: for example, the database underlying the application must be backed up, or a fix to the underlying operating system must be applied. These kinds of maintenance actions are performed by plans that are collectively referred to as *management plans*. Finally, when the application should be de-provisioned a *termination plan* is performed.

Typically, people (like system administrators) understanding the overall structure of an application and its needs in terms of management and administration specify both, its topology template as well as its plans. Application developers focus on the logic aspects of an application, often passing management requirements to administrators as well as structural information about the application. Sometimes, the roles of developers and administrators may coincide: for example scientific applications are often developed by people that also take care about all systems management aspects of it.

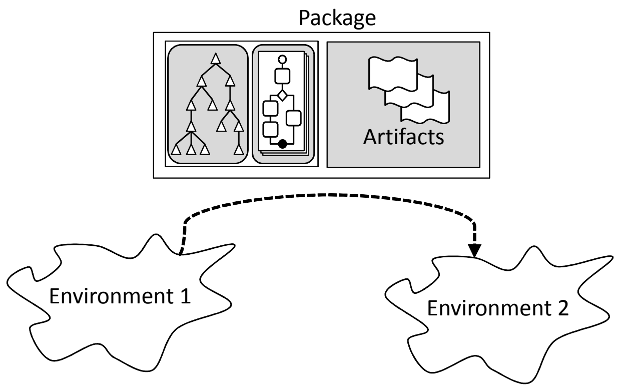

Figure 3 - Packages and Their Exchange

Once the service template of an application has been specified, the artifacts required to execute and manage the application have to be identified. For example, the code of the application logic might be an EJB, the implementation of the operations of the JEE container might be a JAR file, or the implementation of the operating system might be in a virtual image, respectively. All these artifacts are combined with the topology template and the plans into a *package* that can be exchanged between environments (see Figure 3). In essence, a receiving environment will parse the package, identify the contained artifacts, store them and make them available for the plans (especially the build plan), and run the build plan of the service template. After that, the application is ready for use.

The name of the specification indicates that TOSCA is applicable to cloud applications only. But this is not the case: Any kind of application can be described by TOSCA, and a corresponding package can be created. The package may then be exchanged and processed as sketched before. Neither the sending nor the receiving environment might be a cloud environment: what is required at the receiving side is a new middleware component (called *topology container*) that can receive packages, parse packages, and process them. A tooling environment to create such packages (similar to the ones described here) as well as the use of packages to support the movement of applications across environments is described in [LFM11].

Note that "exchange of packages" may have different realizations. For example, the package might be sent to the receiving environment and stored there in some sort of catalogue, for example, instead of immediately processing it. Users may browse the catalogue, find interesting packages, and ask for running the corresponding build plan (so-called "self-service"). We will discuss this kind of usage later in section 4.

3 Compute Units

The concept of a social compute unit (SCU) has been introduced in [DB11]: it is a collection of socially networked humans that provide "compute power" to solve a given problem based on the individual skills of the participating humans as well as software tools that support them. The ingredients of an SCU can be dynamically discovered, composed, and installed.

No details are given in [DB11] about how the software tools are provided, discovered, or installed. We propose to use TOSCA in order to specify the software tools (like any other application) of an SCU. Software tools can be made available as a corresponding package for being composed into an SCU. TOSCA supports the inclusion of documentation elements in each element of a service template. Such documentation elements may be used to specify metadata about the elements of service templates supporting their proper discoverability for composing SCUs. Furthermore, TOSCA is extensible, thus allowing defining new typed metadata for enhanced discoverability of software tools for an SCU. Finally, the problem of how to install software tools that are composed into an SCU now has a simple solution: the build plan of the service template of the corresponding package is executed to install (and configure…) the software tool.

In Figure 4, the social compute unit U_1 is shown. U_1 consists of three human beings one of which is linked to an application depicted as a package. A social compute unit may explicitly include all other resources (like data sets or documents) that are critical for solving the given problem. In Figure 4, we show a data set as such a resource. U_1

includes one particular data set required for solving the problem U_1 has been composed for.

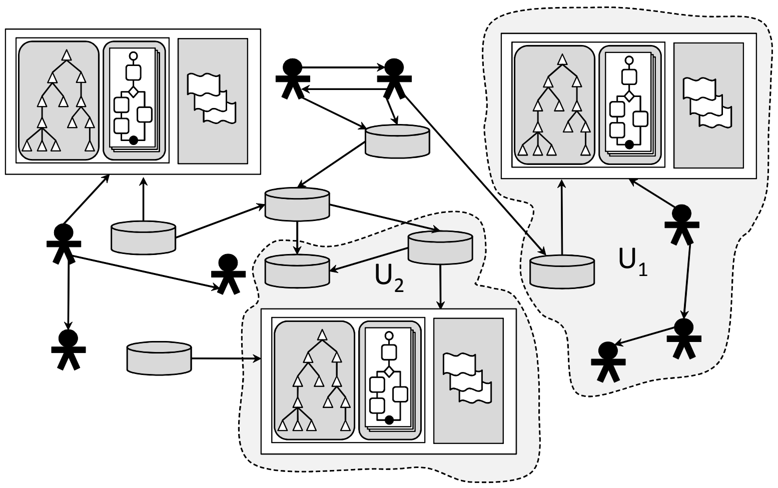

Figure 4 - Extended Social Compute Units

U_2 is another collection distinguished in Figure 4. It consists of two data sets and an application. Because this collection does not include a human being, we refer to it just as a *compute unit*. Compute units may be turned into social compute units by adding human beings to them; vice versa, social compute units can be turned into compute units by having all human beings leaving them. Thus, compute units may be perceived as granules of reuse for composing social compute units. And compute units may be considered as residuals of the work performed by social compute units. The following sections will motivate compute units and their relations to social compute units.

4 Linked Compute Units

Since centuries, the two main approaches of gaining scientific insight are performing experiments and developing theories. A few decades ago, simulations (i.e. computations) became a third established approach to cope with situations where the equations making up a theoretical model are too complicated to be solved analytically: simulations approximate solutions that are hard to determine purely analytical and, thus, gain insight based on computations – this is referred to as the 3rd paradigm of science [HTT09]. When using simulations, potentially huge data sets are often generated that must be analyzed to gain results. Similarly, using large instruments (like large telescopes or particle colliders) also results in huge data sets that are to be analyzed. Often, this analysis takes place at some later point in time, i.e. the data sets have to be maintained. Compared to simulations, which emphasize the importance of computations, large instruments emphasize data in itself as a fundamental aspect of science in addition to the software proper required for the analysis of the data. This new focus on data and the corresponding data processing pipeline is a separate approach for gaining insight, referred to as the 4th paradigm of science [HTT09].

To allow for the verification of research results that are based on data or computations, both, this data as well as all required software (like the simulation software producing the data or software for analyzing the data) must be made available: without the corresponding software, data-intense research results cannot be verified. In case data is made available, it is often associated with other data sets that also must be made available to allow for verification: for example, the data that served as input for a simulation producing the data at hand must be accessible. Furthermore, explicit relations between these data sets must be established easing the comprehension of the dependencies between the data: for example, it must be specified that one data set has been the input for a computation producing another data set. For this purpose RDF links [RDF] may be used, especially to specify the semantics of these relations. The resulting concept of data sets related to each other via links is referred to as *linked data* [HB11].

In [B06], guidelines for linking data have been provided: (i) each data set is to be identified by a URI, (ii) this URI can be dereferenced, (iii) dereferencing will return the data (ideally encoded in a standard format), and (iv) a data set provides links to its related data sets. This way, a "global data space" will evolve [HB11]. Thus, a scientist has to identify all relevant data via a URI. This URI must in fact be a URL that allows using the protocol of its schema (e.g. HTTP, FTP) to retrieve the data, and the retrieval must return this data, ideally in a widely used format (typically dependent on the domain). Finally, the scientist establishes links between the data sets provided, also pointing to data sets that are perhaps relevant but provided by someone else.

Using the linked data concept is a key step forward towards reproducibility of data-intense research results. But what is still missing in this concept is making the software available that is required for reproducibility (e.g. software for the generation, analysis, visualization etc of data). We propose to make this software available as packages. Similar to data, each package is identified by a URI, that can be dereferenced, which will return the package, and packages and data will be linked. What results is a compute unit introduced in the section before extended by links relating its included artifacts – we call this a *linked compute unit*.

This way, data and corresponding software is jointly published. By adding corresponding links the relations between data and software is specified. To verify (or audit…) research results a single URI is required that provides "the entry" into this collection. Based on this entry URI all the other entities required for the verification can be discovered by following links: a link originating at an entity points to another entity by specifying the URI of its target. This URI is dereferencable, i.e. the entity identified by the URI can be retrieved; this link may be source of further links that can be followed again, etc. If a URI identifies data this data can be retrieved; if a URI identifies a package (of an application, software, computation…) this package can be retrieved and "installed". "Installation" basically means to drop the package into a topology container (cf. section 2) that will process it: i.e. the topology container will identify the build plan of the package and run the build plan resulting in an installed (i.e. ready to execute) instance of the corresponding application (or software, computation…). Figure 5 illustrates this. The application may now be used with data sets D_1 and D_2 as input, as specified by the linked compute unit U. This situation would correspond to $D'_1=D_1$ and $D'_2=D_2$; for successful verification the expectation is $D'_3=D_3$. But of course the application can be used with other input data (i.e. $D'_1 \neq D_1$ or $D'_2 \neq D_2$) producing new results establishing trust in the application itself, or producing new insight, respectively.

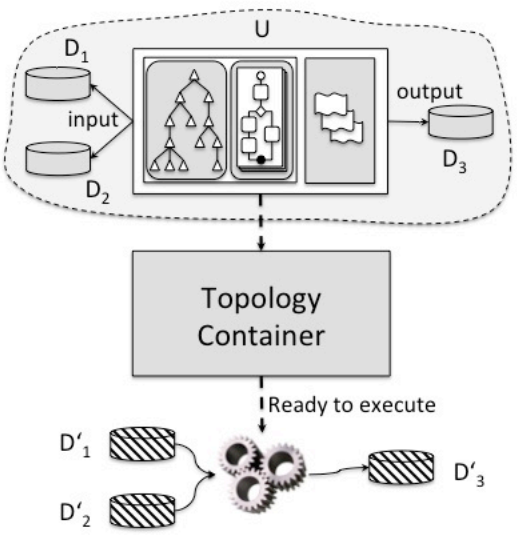

Figure 5 - Processing Linked Compute Units

5 Linked Experiments

Often, when data is produced in experiments human beings must collaborate. For example, astronomers configure a large telescope, database engineers define strategies to store and maintain the captured data, software engineers change or develop software to analyze the captured data at a later point in time, physicists discuss the analysis results of the data, mathematicians determine proper parameterization of simulation runs, and so on. Typically, such people are specialized having dedicated skills and a corresponding team must be discovered and composed: thus, these people match the concept of a social compute unit [DB11].

We propose to identify people of a social compute unit by means of URIs too. Also, people are linked expressing relationships between them or the other ingredients of the social compute unit. For example, a person might be linked to a package indicating that this person has created the package; a person might be linked to a data set specifying that this person owns and maintains the data set; two people might be linked expressing that the first person requires input by the second person; and so on. Note, that instead of a concrete person being a member of a social compute unit an abstract role may be defined. Such a role may again be identified by a URI, which is dereferencable. When dereferenced, a description of the kind of person required (e.g. in terms of skills need) is returned. By adding people or roles to a linked compute unit and linking these people or roles to the other ingredients of the linked compute unit a *linked social compute unit* results.

A linked social compute unit may be seen as a template for solving problems that require the combined power of socially networked humans, software tools, and data resources. Allocating the required people, installing the software tools, and making the data resources available can solve "an instance" of the problem. "Allocating people" means to either request the identified person directly or to first discover team candidates by means of the criteria specified in a role definition and requesting one of them to join the team in a second step. "Installing a software tool" means to install the derefenced package as described before. "Making data available" means to dereference the identified data source.

When work is performed within an instance of the problem new data may be produced and added to the linked social compute unit as well as linked to other elements of the linked social compute unit. Also, some team members may leave the linked social compute unit, others might join. This way, a linked social compute unit is dynamic. The fact that URIs uniquely identify all elements of a linked social compute unit makes it easily traceable, i.e. the elements leaving or joining it are canonically identified (and are dereferencable). Once the problem has been solved, the team members might leave the linked social compute unit and a linked compute unit remains. This residual can be used later on to verify, analyze, or audit the results of the work performed.

We propose to call the application of the concept of a linked social compute unit to science a *linked experiment*. The people (either directly identified or indirectly identified via a role definition) are scientists or other people required to perform an experiment. The software tools are simulation packages, data analysis tools etc. The data sets are data available at the start of the experiment. When performing the experiment new data may be produced. Once the experiment is finished the remaining linked compute unit establishes trust in the scientific insight gained (as sketched before).

Both, linked experiments and linked compute units are granules of reuse. Linked compute units provide data and tools to base new experiments (or simulations etc) on. Linked experiments in addition even provide team structures that have been proven in the past to successfully perform a certain kind of experiment (or simulation etc).

In some areas of science initial steps towards realizing the concept of linked compute units are already taken. For example, the myExperiment website [myExp] shares hundreds of workflows relevant in the area of life science. A scientist may download such a workflow from the website and install it in the corresponding workflow system manually. Perhaps this requires that this workflow system first must be downloaded, installed, configured manually, which in turn might imply that the environment required by the workflow system (i.e. an application server) has to be set up before, and so on. By providing a package that includes the workflow system and its prerequisites as well as the appropriate workflows, setting up the workflow environment as well as the workflows will be a simple automated task.

Of course, this requires the ubiquitous availability of a topology container in the corresponding environment. Since this is a single component only that can bootstrap and set up all other environments provided as packages, requiring such a component is a tolerable prerequisite that supports problem solving and especially experiments.

6 Summary

The concept of linked compute units and linked social compute units introduced in this paper support both, the verifiability of results of data-driven actions as well as the reusability of proven settings in solving data-driven problems. The application of these concepts in science has been presented as linked experiments. These new concepts are based on social compute units, linked data, and topology and orchestration technology from cloud computing, which are combined to extend their area of applicability.

Bibliography (all links followed on February 11, 2012)

[B06] T. Berners-Lee: Design Issues – Linked Data, W3C 2006 http://www.w3.org/DesignIssues/LinkedData.html

[BPMN] Business Process Model and Notation (BPMN) Version 2.0, OMG 2012, http://www.omg.org/spec/BPMN/2.0/

[DB11] S. Dustdar, K. Bhattacharya: The Social Compute Unit, IEEE Internet Computing, Volume 15 (2011), Issue 3.

[HB11] T. Heath, Ch. Bizer: Linked Data – Evolving the Web into a Global Data Space (Morgan & Claypool, 2011).

[HTT09] T. Hey, S. Tansley, K. Tolle: The Fourth Paradigm – Data-intensive Scientific Discovery, Microsoft Research, Redmond, WA, 2009.

[JAR] JAR File Specification, http://docs.oracle.com/javase/1.4.2/docs/guide/jar/jar.html

[LFM11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, S. Dustdar: Moving Applications to the Cloud: An Approach based on Application Model Enrichment, International Journal of Cooperative Information Systems, Volume 20 (2011), No. 3.

[myExp] MyExperiment Website, http://www.myexperiment.org/workflows

[RDF] Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[TOSCA] TOSCA (Topology and Orchestration Specification for Cloud Applications, OASIS 2011, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca