Abstract

The paper describes a new cloud-oriented workflow system called as Flowbster. It was designed to create efficient data pipelines by which very large data sets can efficiently be processed in clouds. The Flowbster workflow can be deployed in the target cloud as a virtual infrastructure through which the data to be processed can flow and meanwhile it flows through the workflow it is transformed as the business logic of the workflow defines it. Instead of using the enactor based workflow concept Flowbster applies the service choreography concept where the workflow nodes directly communicate with each other. Workflow nodes are able to recognize if they can be activated with a certain data set without the interaction of central control service like the enactor in service orchestration workflows. As a result Flowbster workflows implement a much more efficient data path through the workflow than service orchestration workflows.

1 Introduction to Flowbster

Workflow systems are getting more and more popular as the data sets to be processed grow and more and more complex and sophisticated processing is needed. To organize the steps of a complex data processing activity without the automatization of a workflow system is very tiring and cumbersome. There are many different workflow systems proposed and used in the previous decades [1]. They can be categorized into two major classes:

1. Service orchestration (enactor) based workflows [2] [3] [4] [5] [6]
2. Service choreography based workflows [7] [8]

The service orchestration based workflow systems work with a centralized control mechanism called as the enactor. The enactor recognizes that certain activities (nodes)

of the workflow are completed and hence new set of activities (nodes) can be started. The enactor takes care of passing the required data to an executable node and initiate the execution of this node.

When web services and generally service technology became popular it was a natural idea that the nodes of the workflow could be realized as communicating services and the data to be processed just could flow through the set of services and transformed as defined by the services (nodes) of the workflow. A workflow belonging to this service choreography class works as a virtually hard-wired set of services through which the data should flow and meanwhile it flows it is transformed. The advantages of this concept are:

- No enactor is needed and hence workflow management is extremely simple.
- The data is directly passed among the nodes (services) of the workflow and hence the data path is very efficient.
- Data is passed through the workflow in the form of a data stream. This enables that the workflow can be used as a pipeline.

Recently processing very large data sets becomes more and more important and hence the service choreography workflow concept might get momentum since it provides more efficient data paths than the enactor based concept.

The research described in this paper aims at providing a convenient and easy to use workflow system based on the service choreography concept in order to support the processing of very large data sets. The new workflow system is called as Flowbster and will be described in detail in Section 2. Flowbster is based on the idea of dynamically set up the required data pipeline in a cloud until the whole data set is processed. Then the data pipeline can be removed. Such a dynamic creation of the data pipeline requires the usage of a dynamic cloud deployment tool which is in our case Occopus. Flowbster and Occopus together represent a tool family by which complex data processing applications can be built on demand in various cloud systems. They represent different layers of the required software stack as it is explained in Section 2. The Flowbster Workflow System Layer is described in Section 3 including the explanation of Flowbster's support for static node scalability. Section 4 contains details on the Flowbster Application Description Layer and Graphical Design Layer. Finally, in Section 5 we conclude and show the directions of further developments of Occopus.

2 Concept and Overview

The initial assumption for the use of Flowbster is that a scientist would like to process a large set of scientific data (for example, images of the Via Lactea star system in order to find star formulation patterns) stored in a large data center. The processing pipeline (workflow) is either developed or stored in a workflow repository like the SHIWA

Workflow Repository [9] from where the scientist can initiate the deployment and usage of the workflow or should be developed as a new workflow.

The toolset we offer should support the easy development of the workflow as well as its easy and efficient on-demand usage. With the new Flowbster/Occopus concept our objective is to simplify the data pipeline (workflow) creation and operation activities. For example, instead of maintaining a science gateway the scientist herself can initiate the execution of an already developed workflow (data processing service pipeline) in a target cloud by some simple clicks and given the URL of the data location where the data set to be processed is stored and where the processed data should be written back. As a result the workflow will be deployed in the target cloud and will enable the flow of data from the source data storage to the target data storage. The deployed workflow can be considered as a virtual infrastructure that processes and transforms the data meanwhile it flows through this virtual infrastructure. After the data is processed the virtual infrastructure can be removed from the cloud.

The data to be processed by a Flowbster workflow represents a data stream. The elements of the stream are the data elements and Flowbster works as a stream processing workflow system that can exploit both pipeline and workflow branch parallelism. There is a third type of parallelism that can also be exploited in Flowbster workflows. This is what we call node scalability parallelism. It means that if the processing of a node becomes too slow then several instances of the node can be applied in parallel in order to accelerate its work. A typical example of the usage of node scalability parallelism appears in the parameter sweep workflow pattern (see later).

The Flowbster workflow approach is based on a layered concept. Here we distinguish four layers from bottom to top:

1. Occopus Cloud Deployment and Orchestration Layer
2. Flowbster Workflow System Layer
3. Flowbster Application Description Layer
4. Flowbster Graphical Design Layer

The Occopus layer serves to automatically deploy and manage the data pipeline in the target cloud. It is described in detail in [12]. In this paper we describe the three Flowbster layers.

The Flowbster Workflow System Layer defines its uniform building block (workflow node) and execution framework by which complex data pipelines can be built. Flowbster is based on the service choreography concept where the services autonomously work meanwhile communicating with other services. These services realize the functional nodes of the data pipeline workflow. The beauty of Flowbster is that it uses a uniform building block concept for defining these services. This uniform building block can be customized according to the applied workflow execution pattern and the required node functionalities.

Figure 1: Generator-Worker-Collector parameter sweep processing pattern.

Here we mention as examples only two typical workflow execution patterns: the parameter sweep pattern and the if-then-else pattern.

The simplest parameter sweep processing pattern consists of three service (node) types:

1. Generator node
2. Worker node
3. Collector node

Figure 1 shows the basic architecture of a Generator-Worker-Collector parameter sweep processing pattern. The role of the generator is to split a large incoming data element into N smaller ones and output these generated small data elements as an output stream (0,1,2,N in Figure 1). This output stream is taken as an input stream and processed by the worker node. Once the input stream is processed the worker node produces an output stream and passes it to the collector node. The collector collects all the elements of this output stream processes them and merge them into a single data element. This single data element can be placed on its output and can be considered as the result for the incoming data element that originally arrived to the input of the generator node.

The worker node is typically represents a bottleneck in the processing of the parameter sweep pattern since meanwhile the generator and collector are executed only once the worker should be executed on a stream as many times as many data elements are in the stream. In order to accelerate the processing of this data stream the node scalability parallelism should be applied for the worker node. It means that several instances of the worker node are created and the input stream of the worker is evenly distributed among its instances. Figure 2 shows the case when three instances of the worker node is created (W, W‘ and W”) and it also shows an example how the data elements of the incoming data stream of W is distributed among the instances of W.

If the instances are created statically at deployment time based on the scalability parameters given by the user the so-called static node scalability is implemented. Otherwise, if the instances are generated at run time dynamically based on the actual load of the worker node dynamic scalability is realized. Flowbster is currently realized to support static node scalability as written in Section 3.5. Notice that even in this way Flowbster can support both pipeline, workflow branch and node scalability parallelism. This makes Flowbster an extremely efficient system to implement large and complex data processing pipelines.

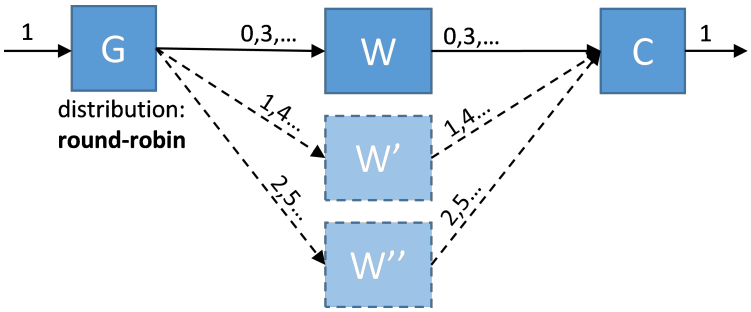

Figure 2: Scaling of worker nodes.

Flowbster Application Description Layer. As we have seen the main concept of Flowbster programming is that Flowbster provides the uniform node building blocks and users should customize them in order to create the concrete data processing pipeline. The Flowbster application description layer has got exactly this role. It enables for the user to

- define graph topology of the data processing workflow by defining the required number of input and out data arcs of the individual nodes and their connections
- define the functionality of the individual nodes

Flowbster Graphical Design Layer. Since the workflow will be deployed and managed in the target cloud by Occopus this definition should be done according to the Occopus node description language. This would require the knowledge of this language that could be too complicated for scientists. Therefore we have defined a new graphical layer on top of this layer. This is called as Flowbster graphical design layer. It provides an extremely easy and intuitive graphical user interface to design the workflow layout and to define the functions for the various nodes. Scientists can use this graphical layer to draw the graph view of the required workflow and then this layer will automatically create the Occopus node descriptors by which Occopus can deploy the workflow in the target cloud.

The paper describes the top three layers of Flowbster in detail in the next two sections and also explains their potential usage.

3 Flowbster Workflow System Layer

The main motivation behind Flowbster was to create a very simple, easy-to-use, basic building block for workflows to provide low-level service for receiving, executing and forwarding a pieces of data item belonging to a data stream. Flowbster was originally planned to be a simple universal building block based on which complex network of processing nodes, i.e., workflows can be built by interconnecting the nodes with various predefined topologies to implement flow of processing network.

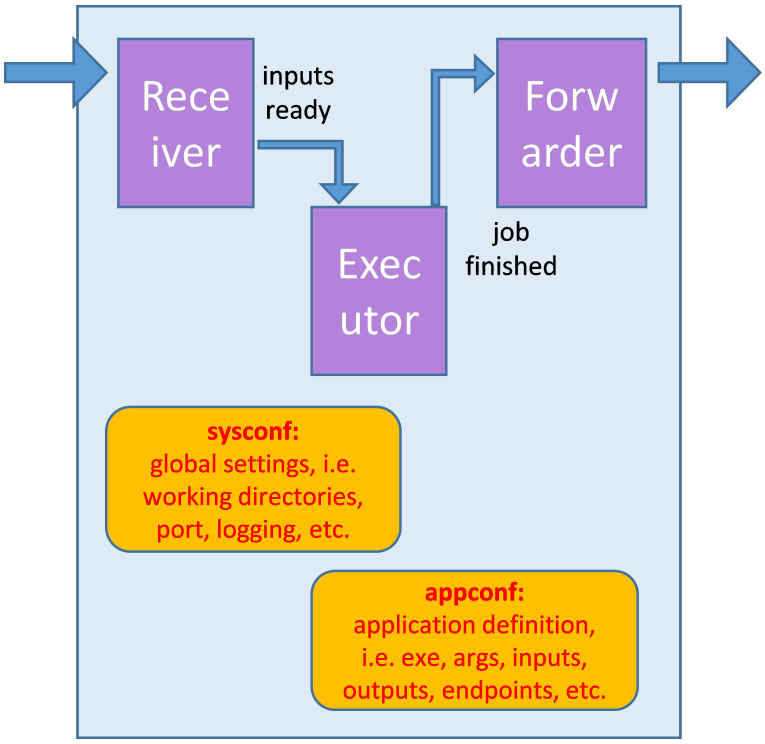

Figure 3: Basic components of a Flowbster node.

3.1 Structure

In order to implement the basic building block of Flowbster the focus was to design a very simple node structure. The Flowbster node is constructed by three basic components:

- Receiver
- Executor
- Forwarder

The *Receiver component* is a service continuously waiting for input data items to stage down on the node for execution. Whenever a new input data item arrives, the receiver decides whether a data set can be passed for execution. The Receiver continuously registers the pieces of data items and monitors if all the required inputs, i.e., all inputs of the preconfigured executable is available. The pieces of input data items for a particular execution can arrive in any order. Once all the inputs are available for an execution, the inputs are marked as ready-to-run and passed for the Executor component.

The *Executor component* is the most simplified part of the Flowbster node. It continuously monitors the ready-to-process input data items. When one is found,

Flowbster app config

```
executable:
  filename: myexe.sh
  tgzURL: http://somewhere.com/myexe.tgz
inputs:
  -
    name: in
outputs:
  -
    name: out
    targetname: in
    targeturl: http://flowbster.node.com:5000/flowbster
arguments: -i in -o out
```

Figure 4: Example application config file of a Flowbster node.

the execution prepared i.e. separate working directory with properly named inputs and all configuration files required by the application are staged. The execution of the predefined application is done as a next step and all the outputs are finally created. Once the execution finished and the outputs are there the job is marked as ready-to-forward and passed for the Forwarder component.

The *Forwarder component* is the link between a successful run and a Receiver node. It means it has the task to properly index and forward the output data items to one or more receiver node(s). During indexing of the outgoing data item the indexes are calculated based on the indexes of data items of the run and on the sequential number of the execution (see details in Section 3.4). For each particular output the Forwarder has the details of the target endpoint of the next Receiver specified.

3.2 Configuration

The configuration consists of two main parts: system and application. The system wide configuration contains settings independent of the target application or executable. It contains for example port definitions, working directories, settings for logging, etc. The parameters defined here are not affecting the execution of the application only the behavior of the three Flowbster components.

Application configuration describes the details of the application to be executed and interconnection details. The application-related details are for example executable path, arguments, list of input and output files necessary for the application. The interconnection related details are URL and name of input for the target Flowbster node to which the output data must be forwarded.

Once the system and application config files are deployed and the components are launched on the Flowbster node, the node is ready to process incoming data sets. The example app config file (see Figure 4) on the Flowbster node will execute ‘myexe.sh‘

command (after downloaded and unpacked the file from tgzURL with ‘-i in -o out‘ command line parameters, and input data stored in file called ‘in‘ when a new data arrives marked as ‘in‘ input. When the execution finished, the file ‘out‘ is forwarded to the next Flowbster node located at targetURL as ‘in‘ input file. This very simple example shows the simplicity of the basic building block of the Flowbster node.

3.3 Feeding, gathering data items

Flowbster nodes can have either internal input/output ports or external input/output ports. Internal input/output ports serve to transfer date between the nodes of the Flowbster workflow. External input ports serve to get the data to be processed from an external file system or database. In order to keep the Flowbster workflow nodes uniform we have to create a new component that can access the external file systems or databases and transfer the fetched data according to the input port protocol of Flowbster nodes to the target external input ports of the Flowbster workflow. This new component is called Feeder. This is not part of Flowbster since there are many different kind of file systems and databases. Usually the Feeder is a very simple service that should be written by the user. However as an example how to write such Feeder services we have create two types of Feeders. The first on can read data from a local file system and the other from an s3 type cloud storage. The source code of these Feeders just like the code of the whole Flowbster/Occopus system is open source and downloadable from github. Based on this downloadable Feeder code user can write access to other types of storages and databases. In the case of storages the usage of the Data Avenue service makes even simpler to write Feeder code for a large set of different storage types [10]. Similarly to the Feeder component we need a Gather component that writes the result data of the Flowbster workflow to the target file systems or databases. The Gather component takes the workflow results from the external output ports of the workflow using the Flowbster data communication protocol. In order to give example how to write Gather components with access to different file systems and storages we have created two types of Gathers. The first on can write data to a local file system and the other to an s3 type cloud storage. The source code of these Gather variants can be downloadable from github, too. Using the Feeder and Gather components the whole system looks like as shown in Figure 5.

3.4 Parameter sweep support

One of the most frequently used workflow pattern is the parameter sweep. Therefore we selected it as the example pattern through which we show the power of the Flowbster/Occopus concept. Flowbster supports the parameter sweep execution pattern as described in Section 2 and illustrated in Figure 1 and Figure 2. A generator always adds 2 new components to the metadata of the incoming data elements in order to produce the metadata of the data elements of the output data stream it generates for the incoming data element. These two new components of the metadata are:

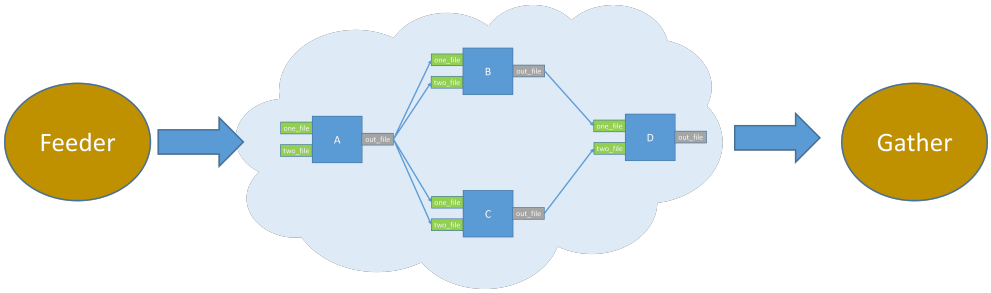

Figure 5: Handling external inputs and outputs of Flowbster nodes.

- Index of the newly created data element inside the output data stream
- The number of data elements belonging to the output data stream

When a generator node (G in Figure 1) emits a data stream with multiple data elements, two new metadata components are automatically assigned to each of them. Flowbster forwards each data elements with its index and the number of generated elements as metadata, i.e., 0,N, 1,N, N-1,N index pairs will be generated as metadata. Worker node (W in Figure 1) keeps the indices untouched, i.e., node W will copy the metadata of an input data element to its corresponding output element as metadata. In Flowbster, a Collector node (C in Figure 1) will knows the overall number of data items in a stream based on the second metadata component in the message. Based on this information collector waits until all the data elements belonging to a certain data stream arrives, i.e., it collects all the data elements of the incoming data stream. At this point the collector function is activated and it processes the data elements of the stream. The result will be placed on its output as a single data element. This is achieved by removing the two metadata components that were defining the input data stream.

As mentioned in Section 2 generators can be created by customizing a generic Flowbster node as generator. Here customization simply means that at least one output port of the node should be defined as generator output port. The *mask* property serves to define which result files should be selected to be considered as the output of the generator. Similarly, collectors can be created by customizing a generic Flowbster node as collector by defining at least one input port as a collector port. The presence of the *collector* and *format* properties will result in making the given input port work as a collector port. More details will be given in Section 4.1 where an AutoDock example will be used to explain the actual description of a simple parameter sweep workflow.

3.5 Static scalability

Node scalability most often used when a workflow node must be multiplied in order to provide enough processing capacity to meet response time requirements as it was

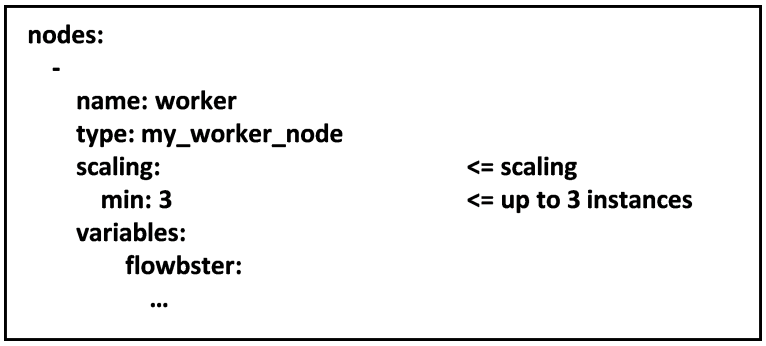

Figure 6: Scaling up section in Occopus .

explained in Section 2. For example, in a workflow where the structure follows the Generator-Worker-Collector pattern node scalability can be used for the worker node as shown in Figure 2. In this case multiple data elements emitted by the Generator (G) must be distributed (e.g. with round-robin) among the Workers (W,W‘,W”) in a way that the Collector (C) receives the union of the outputs of the workers. Scalability can have two versions as described in Section 2:

- **Static scalability** when the user specifies the required number of node multiplications at workflow deployment time

- **Dynamic scalability** when the workflow execution is monitored and if a certain node is overloaded new multiplied copy of the overloaded workflow node is automatically deployed and connected to the workflow at run time without user interaction.

Notice that in both cases the workflow graph contains only a single Worker node as shown in Figure 2. It is only Occopus that can create (and ”see”) the multiple instances of the Worker node either based on the user definition or the monitoring information. Currently Flowbster supports static scalability. In order to scale up a node at startup in Occopus extra scaling section must be provided in the infrastructure description for the node as shown in Figure 6. In this example the worker node will have exactly 3 instances after building up the infrastructure due to the minimum specified. Optionally, maximum can also be specified for Occopus to limit the number of instances in case dynamic scaling is utilized. In this example the maximum will be equal to minimum by default due to unspecified maximum value.

When Occopus builds the nodes of the workflow in Figure 2, the dependency in the infrastructure description will force Occopus to keep the following order in node deployment: first the node Collector is deployed, then all Workers, and finally the node Generator. The Generator will therefore receive the endpoints of all the worker nodes and hence the Generator can perform the distribution of data elements among the multiple worker nodes.

In Flowbster, output data elements can be transferred to the target input ports according to two options:

- **Copy (default):** Without any special setting on the output port with multiple endpoints, Flowbster will copy each outgoing data element to all the attached endpoints (i.e. to multiple instances) of the target node.

- **Distribution:** When the output port is marked with the keyword "distribution" in Flowbster, the outgoing data elements will be exclusively assigned to one of the attached target nodes. With this option the output data is distributed to multiple instances of the target nodes supporting the scalability concept (for example, the outputs of the Generator are distributed for the three worker node instances W, W', W" in Figure 2). Currently two ways of output data distribution is supported: random and round-robin.

4 Flowbster application layers

In this section we present the Flowbster application layers: the description and the graphical design ones. These two layers, building on top of Occopus and the Flowbster framework, enables users to develop workflow applications on top of managed cloud infrastructures.

The two layers of the application layer are the application description layer and the application graphical design layer. The first layer is responsible for representing the workflow application in a textual form, building on the features provided by the Flowbster/Occopus framework. The second layer offers a convenient graphical tool for designing workflows without the need to know anything about the description layer.

In this section we present these two layers in detail through the AutoDock Vina [11] application and its workflow. The AutoDock Vina tool can be used to perform molecular docking simulations. For this, it needs a configuration file, a receptor molecule and a set of ligands to be docked against the receptor molecule. The different dockings are completely independent from each other, thus the different dockings can be run in parallel.

4.1 Flowbster Application Description Layer

As discussed earlier in this paper, the Flowbster framework offers convenient building blocks to create complex workflow applications. The description layer relies on this framework for composing a workflow application. The task of the user exploiting this layer is to create an Occopus infrastructure description by customizing the Flowbster nodes properly, i.e., by defining all the required properties.

```
infra_id: copy_workflow
user_id: foo@bar.com
name: copy_workflow

variables:
    flowbster_global:
            gather_ip: &gatherip 192.168.1.1
            gather_port: &gatherport 5001
            receiver_port: &receiverport 5000

nodes:
    - &Copy
        name: Copy
        type: flowbster_node
        scaling:
                min: 1
                max: 1
        variables:
            flowbster:
                app:
                    exe:
                        filename: copy.sh
                        tgzurl: http://foo.bar/copy.tgz
                    args: -i in_file -o out_file
                    in:
                        -
                            name: in_file
                    out:
                        -
                            name: out_file
                            targetname: copy_result
                            targetip: *gatherip
                            targetport: *gatherport
```

Figure 7: Descriptor of a single-node workflow.

4.1.1 Single-node workflow

The very basic Flowbster workflow is a single-node workflow. Such workflow has only one single node running one executable, accepting a set of input files, and producing a set of output files. The example in Figure 7 shows the descriptor of such a single-node workflow where the single node runs a "copy" service.

In the above example, we have only one single node in the infrastructure called *Copy*. The type of the node is *flowbster_node*, which refers to the uniform node provided by Flowbster. The task of this descriptor is to instruct Flowbster how to customize the uniform node to work as a "copy" service. In order to enable node scaling the user can specify the minimum and maximum number of instances that can be created from this node. In the current example the enabled number of instances is 1 for

simplicity.

The variables inside the node descriptor define properties of the executable to be run inside the given node. First, the name (*filename*) and the download location (*tgzurl*) of the executable are defined. The Flowbster framework will fetch the executable TGZ archive from the given URL, expand it into a temporary directory, and invoke the file set inside the *filename* property when executing the application. Next, the command line arguments (*args*) are defined for the application. Afterwards, the input and output files' properties are set. The common property for both input and output files is the *name* property, specifying the name as the application will open the files. In the output file section, three additional properties specifying the target of the produced output files is set: the *targetname* specifying how the target file will be called, the *targetip* and *targetport*, specifying the location of the target data service where the output file will be written.

Besides the local (node-level) variables there are also global ones. The global variables define those properties each node inside the workflow will receive. These are the IP address and listening port of the Gather component where the result data should be transferred, and the listening port of the receiver components inside each Flowbster node.

4.1.2 Multi-node workflow

Multi-node workflows are capable of exploiting the true power of the Flowbster framework. Beside the basic executable and input/output file properties, one can define data dependency among the workflow nodes, and special collector and generator properties as well.

As an example how to define a multi-node workflow we show a simple parameter sweep workflow (Generator $\rightarrow$ Worker $\rightarrow$ Collector) as described in Section 2. The example parameter sweep workflow implements an AutoDock Vina application and its descriptor is shown in Figure 8.

This workflow contains three nodes implementing a parameter sweep (PS) version of the AutoDock Vina molecular docking tool: a *GENERATOR* node, a *Vina* node (with 16 node instances) and a *COLLECTOR* node. The Vina node as the worker of the PS workflow pattern can be multipled to support node scalability parallelism. In order to achieve it the *scaling* parameters *min* and *max* are set to 16. As a result Occopus will deploy 16 Vina nodes in 16 virtual machines that are connected to the rest of the workflow nodes as explained in Section 2.

The task of the *GENERATOR* node, after receiving the Vina configuration file, a receptor molecule and a set of ligands to be docked against the receptor molecule, is to split the input ligand set into as many parts as many has been specified in the command line arguments (1024 in our example). The split ligand set is distributed in a random manner among the 16 instances of the *Vina* node. The receptor molecule and the Vina configuration file are sent to all the instances of the *Vina* node to perform the docking simulations. Finally, the results of the Vina docking simulations from the

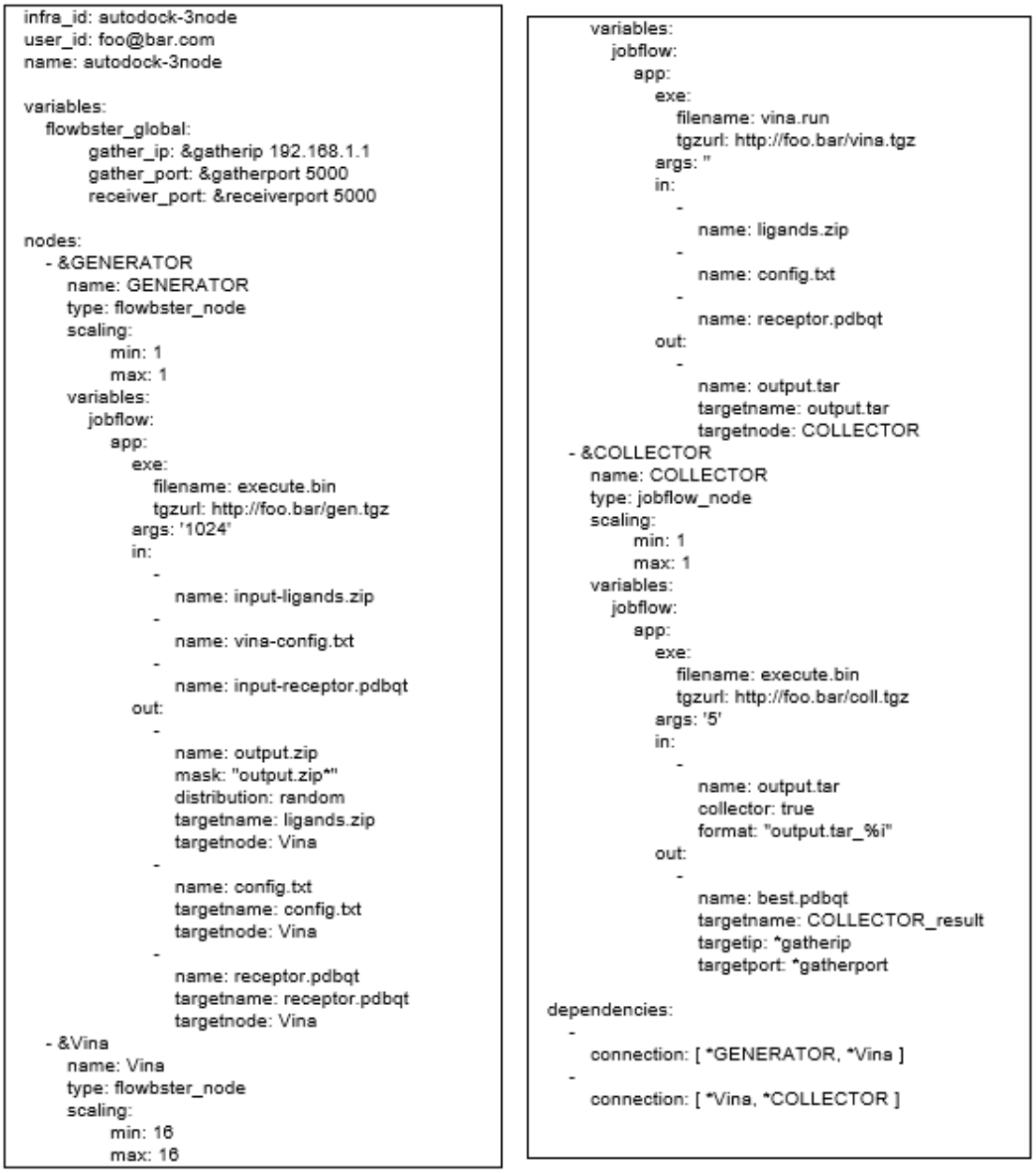

Figure 8: AutoDock workflow descriptor.

Vina node instances are sent to the *COLLECTOR* node, which selects the five best docking results and passes to the Gather component as the results of the AutoDock workflow.

The following differences can be identified regarding input and output files when comparing with the single node workflow:

1. Connecting ordinary output ports to ordinary input ports: The *targetnode* property in the ”out” part of a node descriptor specifies the name of the target node to which the produced output file (or files) should be sent to. In this way node outputs can be connected to inputs of target nodes.

2. Defining generator output port: The *mask* property serves to select files to be considered as the generator outputs. It instructs the Flowbster framework to create output file names according to a regular expression. For example, output.zip* is defined in the case of the first output of the GENERATOR. Any file matching the regular expression will be considered as a generated instance of the given output file. The optional *distribution* property defines the way of distributing the generated output file instances. Two distribution methods are supported: *random* (choosing randomly from the set of target node instances) and *round-robin* (distributing file 1 to node instance 1, file 2 to node instance 2, etc.). If the *distribution* property is omitted, the generated output file set is sent to all instances of the target node.

3. Defining collector input port: the presence of the *collector* and *format* properties will result in making the given input port work as a collector port. It means that the COLLECTOR node can only be executed when all the files generated by the Vina instances have arrived to the COLLECTOR node as explained in Section 2.

To sum up, the above workflow will be deployed in a target cloud as a set of 18 virtual machines that are connected to each other according to the Flowbster workflow topology.

4.2 Flowbster Workflow Design Layer

Although the workflow description layer is relatively simple, it is still not a convenient interface for users. A straightforward step towards providing a user-friendly interface for creating Flowbster-based workflows is creating a graphical user interface which enables creating the workflow layout structure, and setting executable and input/output file properties for the different nodes in the workflow.

In the graphical design layer connections between nodes of the workflow can be defined by dragging an output file of the job, and connecting it to the input port of another job. The graphical representation of the Vina workflow described earlier is shown in Figure 9.

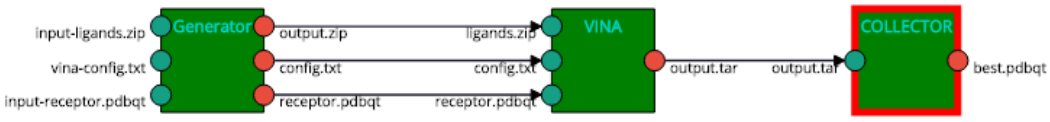

Figure 9: AutoDock Vina workflow constructed.

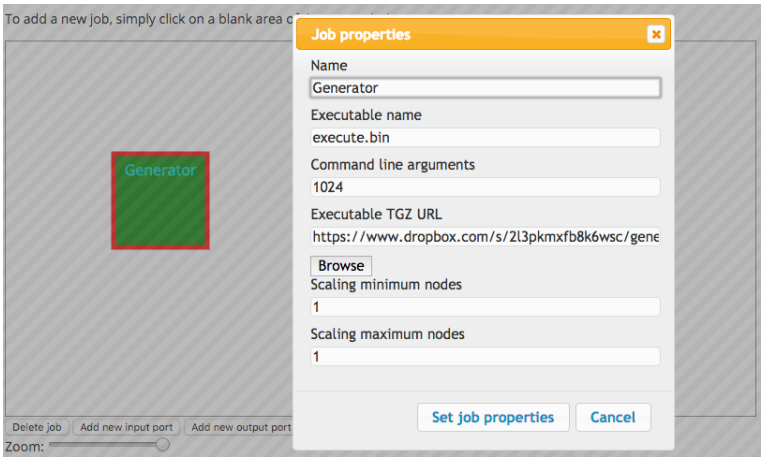

Figure 10: Job property dialog and graph workflow interface.

New jobs can be created by simply clicking onto a blank area of the main view, which will result in bringing up the Job properties dialog, where the user can enter properties of the job as shown in Figure 10.

Once a job has been selected (by clicking on it), the user can add new input and output files by clicking the *Add new input port* and *Add new output port* buttons (see in the bottom of Figure 10). The files' properties can be set by double-clicking on the circles attached to the job. For example, Figure 11 shows the output port definition dialog.

Instead of creating a user interface from scratch, we have decided to evaluate existing user interface frameworks for defining graphs. A number of graph and workflow definition frameworks exists, and we have selected to create the Flowbster graph editor based on JointJS [13] , with the additional discrete event system specification (DEVS) extension [14]. The advantage of using JointJS is that it can be extended in many different ways, is based on pure Javascript, HTML and CSS, and finally, is able to serialize and deserialize the graph into and from JSON. These advantages did allow us to produce a usable Flowbster editor very quickly, which can be deployed onto a big variety of website hosting environments.

The final task for supporting Flowbster workflows using the graphical user interface is the conversion between the internal representation of the graphical workflow editor and Occopus. JointJS can dump and parse the graph along with its properties into a

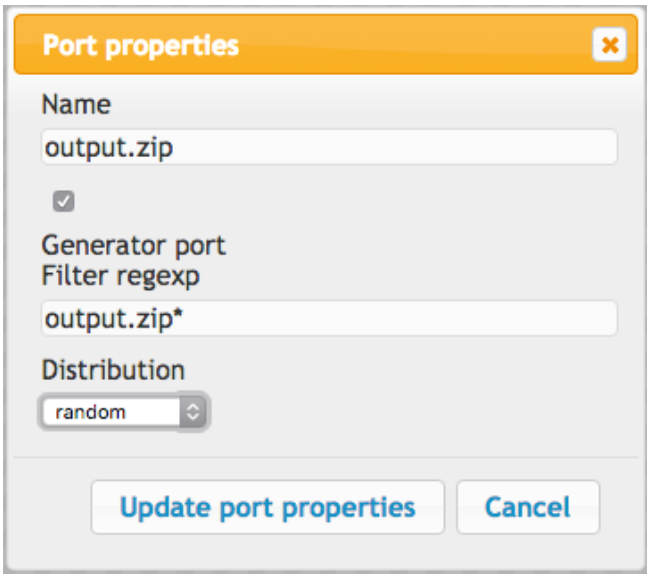

Figure 11: Output port property dialog.

JSON representation. This JSON representation contains two main properties: the list of cells and the list of links. The cells property enumerates the nodes of the workflow along with their properties, and links property enumerates the connection between the different nodes' output and input files. As a result transforming the JSON representation into the Flowbster/Occopus descriptor like the one in Figure 8 is straightforward. As a result the users do not have to learn the Flowbster/Occopus descriptor language, only the very intuitive graphical user interface.

5 Conclusions and further development

The appearance of cloud systems and the need for workflows by which very large data sets can be efficiently processed requires to find radically new directions in organizing scientific workflows. Such a radically new concept is the Flowbster/Occopus framework by which users can virtually hardwire their workflows as virtual infrastructures into the target clouds. Occopus guarantees that the workflow can be deployed in any major types of IaaS clouds (OpenStack, OpenNebula, Amazon, EC2, CloudSigma). It also enables if needed to deploy the workflow in a multi-cloud environment through several different kinds of clouds. It takes care of not only deploying the nodes of the workflow but also to maintain their health by using various health-checking options. Occopus also enables the implementation of the node scalability parallelism an important feature to accelerate data processing inside the workflow. Performance measurements show that the exploitation of node scalability parallelism indeed significantly accelerates the work of the data pipeline in case of parameter sweep applications. Currently Flowbster/Occopus supports the static version of node scalability

parallelism but soon the dynamic version will be investigated.

Flowbster defines and implements the service assembly layer of the workflows by defining and implementing the required micro-services and their communication protocols. Based on these micro-services it creates the uniform building blocks of Flowbster workflows that can be customized by the workflow developers to create specialized workflow nodes like generator, worker, collector, join, fork and others from which complex, high-level workflow patterns like parameter sweep, if-then-else and others can be easily created. Due to space restrictions in the paper we focused on the description of how the very popular parameter sweep pattern can be realized in Flowbster. This pattern also well demonstrates how pipeline parallelism and node scalability parallelism can be exploited in Flowbster besides the usual workflow branch parallelism that is naturally coming from the typical parallel branch topologies of the workflows.

It was also shown how the Occopus descriptors should be created for Flowbster workflows to make them deployable by Occopus. An important feature of the Flowbster/Occopus framework that it provides an intuitive graphical user interface that completely hides and automatically generates the required Occopus descriptors. As a result developing Flowbster workflows and deploying them in target clouds is extremely easy and does not require any cloud knowledge. The user can concentrate the business logic of her workflow while the Flowbster/Occopus framework guarantees and efficient cloud based execution. In order to further improve the cloud based execution we plan to create the docker version of Flowbster. In this way the Flowbster workflow nodes will be packed into docker containers instead of virtual machines that will significantly further improve performance and portability.

The Flowbster/Occopus framework is an open source product licensed under the Apache License, Version 2.0 (the ”License”). The complete source code can be downloaded from github at https://github.com/occopus.

Acknowledgement

The research leading to these results has received funding from the European Union Horizon 2020 research and innovation programme under Grant Agreement No. 644179 (ENTICE) and from the COLA - Cloud Orchestration at the Level of Application project, Grant Agreement number 731574 (H2020-ICT-2016-1).

This work was also partially supported by the National Research, Development and Innovation Fund of Hungary under grant No. VKSZ_12-1-2013-0024 (Agrodathu), and by the International Science & Technology Cooperation Program of China under grant No. 2015DFE12860.

On behalf of Project Occopus we thank for the usage of MTA Cloud (https://cloud.mta.hu/) that significantly helped us achieving the results published in this paper.

References

[1] Ji Liu, Esther Pacitti, Patrick Valduriez, Marta Mattoso: A Survey of Data-Intensive Scientific Workflow Management in Journal of Grid Computing, vol. 13., No. 4., pp 457-494 (2015)

[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. d. Silva, M. Livny, and K. Wenger. Pegasus: a workflow management system for science automation. Future Generation Computer Systems, 2014

[3] A. Balasko. Workflow concept of ws-pgrade/guse. In P. Kacsuk, editor, Science Gateways for Distributed Computing Infrastructures, pages 33-50. Springer International Publishing, 2014

[4] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In 16th Int. Conf. on Scientific and Statistical Database Management (SSDBM), pages 423-424, 2004

[5] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology, 11(8):1-13, 2010

[6] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics,* 20(17), 2004.

[7] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, "A Language for Service Behavior Modeling," in CoopIS, Montpellier, France, Nov 2006

[8] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon, "Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation," Tech. Rep., November 2005

[9] G. Terstyanszky, T. Kukla, T. Kiss, P. Kacsuk, A. Balasko, and Z. Farkas. Enabling scientific workflow sharing through coarse-grained interoperability. Future Generation Computer Systems, 37:46-59, 2014

[10] Hajnal Á, Márton I, Farkas Z, Kacsuk P: Remote storage management in science gateways via data bridging CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE 27:(16) pp. 4398-4411. (2015)

[11] O. Trott, A. J. Olson, AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading, Journal of Computational Chemistry 31 (2010) 455-461

[12] J. Kovács, et al., Orchestrating federated clouds by Occopus, PARENG'2017, Pécs, Hungary, 2017

[13] http://www.jointjs.com/

[14] http://www.jointjs.com/demos/devs