9th IEEE/ACM International Symposium on Cluster Computing and the Grid

On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud

Tim Dörnemann, Ernst Juhnke, Bernd Freisleben
Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str. 3, D-35032 Marburg, Germany
Email: {doernemt, ejuhnke, freisleb}@informatik.uni-marburg.de

***Abstract*— BPEL is the de facto standard for business process modeling in today's enterprises and is a promising candidate for the integration of business and Grid applications. Current BPEL implementations do not provide mechanisms to schedule service calls with respect to the load of the target hosts. In this paper, a solution that automatically schedules workflow steps to underutilized hosts and provides new hosts using Cloud computing infrastructures in peak-load situations is presented. The proposed approach does not require any changes to the BPEL standard. An implementation based on the ActiveBPEL engine and Amazon's Elastic Compute Cloud is presented.**

I. Introduction

With the advent of service-oriented architectures (SOA) and web services as the most widely used implementation technology of SOA, a new way of writing distributed applications has emerged (often referred to as "Programming in the Large"). Applications may be composed of existing components (i.e. web services) which leads to higher reusability and thereby faster development and reduced costs. The Business Process Execution Language for Web Services (BPEL4WS or WS-BPEL [10]) is the de facto standard for service composition in business applications. Access to a process is exposed by the execution engine through a web service interface, allowing the process to be accessed by web service clients or to be used as a basic activity in other processes.

While BPEL works well for modeling processes with target hosts for the execution of process steps known at runtime (called static process in the following), it has some drawbacks when it comes to dynamically selecting target hosts at runtime. Static definition of target hosts might be no problem for typical business processes such as the often quoted loan approval example, but when it comes to modeling of computationally intensive processes like scientific workflows with BPEL [9], [15] it would be highly beneficial to have a BPEL implementation that automatically selects target hosts with low load offering the required service.

Furthermore, the process execution system should be able to react on peak loads. Consider the case that researchers share computational resources in a Grid's virtual organization (VO) or a company offers some web-based service that utilizes computationally intensive BPEL processes in the background. If many researchers or customers simultaneously access the infrastructure (by invoking the processes), a reduction of quality [14] is quite likely. Therefore, the system should dynamically provide additional computational resources. This is one of the main ideas of Cloud computing, and a well-established service in this area is the Elastic Compute Cloud (EC2) offered by Amazon [1].

In this paper, an approach that extends a well-known open-source BPEL implementation (ActiveBPEL, [8]) to dynamically schedule the service calls of a BPEL process based on the target hosts' load is presented. To handle peak loads, it integrates a provisioning component that dynamically launches virtual machines in Amazon's EC2 infrastructure and deploys the required middleware components (web/Grid service stack) on-the-fly.

The implementation is non-invasive in two respects: it does not require any changes to the BPEL standard, and changes to the BPEL engine's source code are not necessary. Our implementation makes use of ActiveBPEL's extensibility mechanisms and thus preserves portability towards new releases of the engine. Finally, to ease dynamic process modeling, an Eclipse-based BPEL designer [12] is extended to allow the graphical definition of required settings.

The paper is organized as follows. Section II presents the problem statement. Section III shows the proposed architecture as well as the design of our approach. Section IV describes implementation details. Section V presents experimental results using a sample workflow. Section VI discusses related work. Section VII concludes the paper and outlines areas for future research.

II. Background and Problem Statement

BPEL is the de facto standard for business process modeling in today's enterprises. In the area of scientific (Grid) computing, no such clear preference exists yet, but many researchers begin to use BPEL as their composition language. BPEL offers a rich vocabulary and control mechanisms to express sequences of activities like *receive*, *invoke* and *reply*, parallel execution, loops, error handling as well as compensation-mechanisms to perform roll-back actions. For this work, the *invoke* activity is of particular interest. It is used to model the invocation of external services. The target service to be invoked is described via a so-called *partnerLink* that – besides others – contains two important elements: (1) *partnerLinkType* and (2) *EndpointReference* (EPR). (1) is static information that must be known at design time. It refers to the WSDL description of a partner service's *portType*, while (2) refers to the concrete service to be invoked. An EPR contains a service's name, its port (and binding mechanism) and its address given by a URI. The EPR is evaluated at runtime and may even be set at runtime, as Listing 1 illustrates.

```
<assign>
<copy>
 <from>
  <literal>
   <wsa:EndpointReference xmlns:ns="NSPACE">
    <wsa:Address>
      http://FQDN:PORT/SERVICE-ADDRESS
    </wsa:Address>
    <wsa:ServiceName PortName="Port">
     ns:SERVICE-NAME
    </wsa:ServiceName>
    <wsa:ReferenceParameters>
     <wsa:To>...</wsa:To>
     <wsa:Action>...</wsa:Action>
    </wsa:ReferenceParameters>
   </wsa:EndpointReference>
  </literal>
 </from>
 <to variable="targetEPR"/>
</copy>
<copy>
 <from variable="targetEPR" />
 <to partnerLink="targetPL" />
</copy>
</assign>
```

Listing 1. Manual setting of a dynamic endpoint in BPEL

The listing demonstrates that it is quite complicated to set a service endpoint at runtime, and that it bloats the business logic with code related to infrastructural settings. Furthermore, prior to setting the EPR, the target address must be determined via calling a scheduling or discovery service.

Furthermore, current BPEL implementations do not provide features for scheduling service calls based on the load of possible target hosts. When computationally intensive processes are run simultaneously on a dedicated infrastructure, either the response times will increase or the invoked service might not react at all, resulting in an abandonment of the entire process. This may lead to loss of stability in the workflow system; a high cost due to wasted CPU hours due to lost intermediary results of preceding process steps is another consequence. The system can be stabilized and the response times can be decreased by adding additional hosts on demand (see Figure 1). Since Cloud computing infrastructures allow to dynamically provide hosts with custom software installed, such an infrastructure could be used to react to peak loads or even to replace dedicated infrastructures. In this scenario, resources would only have to be paid when they are actually used. Administration complexity would be reduced to a minimum.

Cloud computing is a style of distributed computing in which resources are provided as services. Users may access resources on-demand (mainly computational power and storage) through simple interfaces like web services, without knowledge or control of the technology and the infrastructure used by the provider. For the implementation, many Cloud computing providers like Amazon [1] make use of virtualization. This allows to run more than one virtual machine on a dedicated host, offers user isolation and enables the provider to give the customer full (root) access to the (virtual) machines. Root access is often necessary to install and configure software and to open the Cloud infrastructure for a broad variety of applications. In the Amazon Cloud infrastructure, users can configure virtual machine images with customized operating systems and installed user applications that are stored in the Cloud infrastructure. On request, such a virtual machine is booted on a physical host in the infrastructure and may be used like a dedicated physical host shortly after the request. Depending on the requested configuration of the virtual machine (number of CPUs, amount of RAM, instance storage), the price per time slot varies.

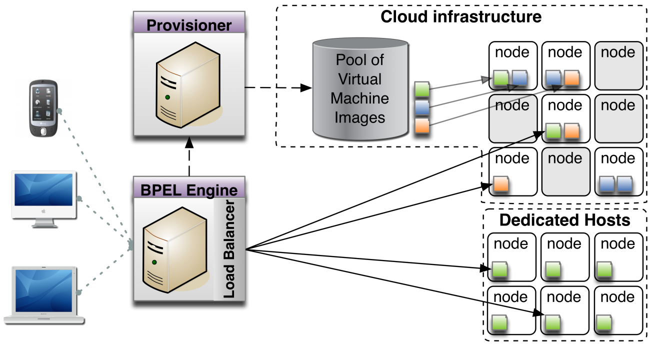


Fig. 1. Topology of a Cloud-enabled workflow system

III. Cloud-enabled Load Balancing Architecture

In this section, the design of the solution to satisfy the challenges mentioned in the previous section is presented. Implementation details will be presented in Section IV.

The proposed solution combines the versatile and powerful composition and control mechanisms of BPEL with an adaptive runtime environment without breaking the compatibility with existing standards. The system seamlessly integrates dedicated resources and on-demand resources provided by infrastructures like Amazon EC2. When used in conjunction with our previous extensions to BPEL [12], the proposed system even allows to invoke stateful web services (i.e., services implemented according to the Web Services Resource Framework [5]) that are widely used in a Grid middleware like the Globus Toolkit 4 (GT4) [17], [3].

The architecture has three components (see Figure 2):

1) The Dynamic Resolver (DR) extends the BPEL engine's invocation mechanism
2) The Load Balancer (LB) manages existing hosts, schedules service calls and provides new hosts
3) The Load Analyzer (LA) collects information about the system load of hosts

The interplay of the components is as follows: whenever, during the execution of a workflow, a service has to be called, the workflow system checks whether the target host

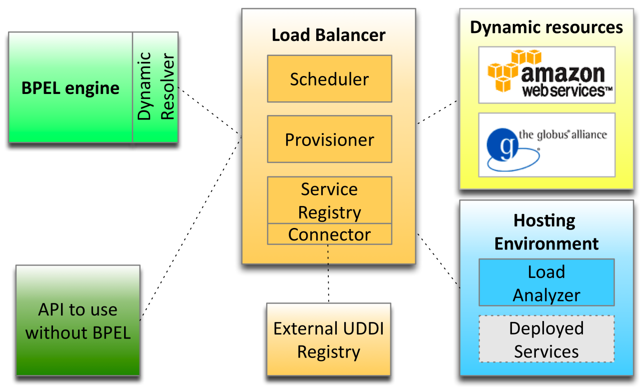


Fig. 2. Three-tier architecture of our solution

has already been selected or not. In the latter case, the DR tries to determine the best-matching target host. It contacts the LB and passes the service description to the LB, which then queries an internal registry for hosts having the requested service installed. The resulting list is enriched with information concerning the performance and current load of the hosts. The list is then passed to a scheduling component that determines the best-matching host or provides a new host in case all hosts have high load. Finally, the selected target host's address is passed back to the workflow system to execute the call.

In the following, the design of the components and their interplay is discussed in detail.

A. Extensions to the BPEL engine

The DR is designed to avoid any changes to the BPEL engine's code, which eases portability to new versions. It is invoked whenever, during the execution of a process, a *partnerLink* is set to *dynamic* with the attribute *invokeHandler* pointing to the DR's implementing class. It then checks whether the corresponding BPEL *partnerLink* already has an endpoint address set or not. In the latter case, it invokes the LB component to determine a concrete endpoint. In order to do so, the *QName* of the *portType* and other parameters are passed to the LB component. The LB (see Section III-B) in turn returns the endpoint address of the service that matches best. In this way, the DR constitutes the interface between the BPEL engine and the LB component.

Since different scheduling strategies and infrastructural backends may be implemented, it is necessary to have the ability to define arbitrary parameters for dynamic endpoints. For instance, the workflow designer might want to define a minimum load threshold to start new virtual machines; authorization information for infrastructures like Amazon EC2 may be required.

B. Load Balancer

The LB is the main component of our architecture. It has three components: scheduler, provisioner, and registry. It manages dedicated target hosts, continuously monitors their system load and makes scheduling decisions to avoid that hosts get overloaded. To make the infrastructure as cost-efficient as possible, it releases Cloud resources if the overall load is normal.

The DR contacts the LB whenever an endpoint for a target service needs to be resolved. Then, the registry is queried for hosts having the requested service installed. Using the result list, the registry then collects the current system load on all qualified hosts. This information is then passed to the scheduler to make a scheduling decision according to the configured scheduling algorithm. This scheduling decision might involve starting new virtual machines by calling the provisioner. Eventually, the scheduler returns the target endpoint reference to the LB that in turn replies to the DR.

Details about all involved components are presented below.

1) Registry: The registry manages all information about hosts and services offered by them. In particular, the endpoint of services must be accessible via their *portType*'s *QName*. Since the start and shutdown of virtual machines are essential parts of the system, the registry must also be able to cope with the appearance and disappearance of hosts and services. Because a query may happen while a virtual machine is shutting down, it must be assured that the virtual machine must not appear in the result set of the query. To ensure that virtual machines are not shut down while they are in use, a construct that locks the shutdown procedure for virtual machines has been implemented.

When virtual machines are started, information about them has to be saved persistently. This makes the registry resistant to failures and ensures that running virtual machines may still be managed after a restart of the LB.

Furthermore, the registry can query other (UDDI) registries, such as Grimoires [4], to extend its scope. Also, using a configuration file, a set of "static" hosts can be included in the registry – for instance, a company's dedicated hosts.

To collect load values, the LA is queried by the registry every time the registry is queried. To reduce communication costs, a cache holds the load information in memory; the cache is invalidated after a configurable interval. Thus, frequent queries do not affect the registry's performance.

2) Provisioner: The provisioner encapsulates interfaces to manage virtual machines in on-demand infrastructures like Amazon EC2 and Globus Virtual Workspaces [18]. It provides general abstractions, such as starting and stopping virtual machines, and methods for preparatory steps, such as starting middleware components.

With this abstraction, it is possible to develop drivers that interface with providers that use different implementation technologies and communication patterns for their management interfaces. These drivers may be plugged in to the provisioner using a simple configuration file. Thus, the approach allows to integrate new infrastructures without a re-design of the component itself. For example, Amazon EC2 provides web service based access to their management interface and does not deliver notifications when machines have finished booting – a polling-mechanism needs to be implemented in this case. On the other hand, Globus Virtual Workspaces make use of WSRF based services and implements WS-Notification, meaning that a caller gets notified when certain events happen. The provisioner therefore needs to be generic and must hide implementation intrinsics.

The provisioner registers and de-registers virtual machines in the registry after booting and shutdown. The registry then adds or removes all services installed on the virtual machine.

3) Scheduler: The scheduler gets information about the infrastructure, makes scheduling decisions and invokes the provisioner to manage dynamic resources. Since for different application scenarios different scheduling strategies are required, the scheduler allows to plug-in scheduling algorithms. A sample scheduling strategy and its implementation is described in Section IV-B.3.

The scheduler memorizes the starting time of virtual machines, since services like Amazon EC2 are billed per use, meaning that shutting down a virtual machine before expiration of an accounting period saves money. Therefore, it must tell the provisioner to shut down virtual machines in due time if the overall load is low. The system has to take into account that there must not be any running service calls on the machine to be shut down. While this may sound trivial, it is not straightforward to determine whether a service call (or the induced computation) has finished or not. In Section IV, this problem and a solution is discussed in depth.

C. Load Analyzer

Many scheduling algorithms make scheduling decisions based on optimizing the throughput, meaning that it is favorable to schedule jobs or calls to hosts with low load, since assumingly these hosts finish the task faster than a host with higher utilization. Therefore, the system needs to collect load information for all target hosts to provide the scheduling algorithms with this information. The proposed solution employs a web service for analyzing the load that is installed on every host. It is queried by the registry in order to update system load values. In our currently implemented prototype, "system load values" represents the following tuple: the number of CPU cores, the load of the host and an estimated measure of the system performance.

On systems where one (head) node represents a number of hosts (e.g. a cluster site), the number of cores and the load are calculated by the following simple formula. Let n be the number of nodes in the system, $c(\cdot)$ a function that returns the number of CPU cores for a given node and $l(\cdot)$ a function that returns the load for a given CPU core. Then:

$$cores := \sum_{i=0}^{n} c(i) \qquad load := \frac{\sum_{i=0}^{n} \sum_{j=0}^{c(i)} l(j)}{\text{cores}}$$

If a cluster has heterogeneous nodes, the values above have to be calculated for each subgroup of homogeneous nodes.

Furthermore, the LA provides a method to generate a list of all services being exposed by the application container to enable the LB to automatically determine the services' WSDL documents.

IV. Implementation

Our implementation is based on ActiveBPEL [8], the workflow enactment engine developed by ActiveEndpoints, a stable and well-documented software that is published under GNU Public License (GPL).

A. Extensions to the BPEL engine

To realize the main goal of minimizing changes to the workflow engine, the DR makes use of interfaces and observers provided by the workflow engine. The DR implements the `IAeInvokeHandler` interface provided by the workflow engine. The DR is used when a call is executed on a *partnerLink* marked with our *invokeHandler* (see Listing 2). Due to this *invokeHandler*, the binding stays dynamic. Since the *invokeHandler* is specified via an URI, necessary arguments may be passed to the DR via an URL encoding (lines 4 and 5 in Listing 2).

```
<partnerLink name="decoderPL">                          1
 <partnerRole endpointReference="dynamic"               2
  invokeHandler="java:LoadBalancer                      3
  ?threshold=1.0;accessID=***;secretKey=***;            4
  imageID=ami-95cc28fc;availZone=us-east-1c"/>          5
```

Listing 2. Definition of a custom invoke handler

When the DR gets executed, the *partnerLink* associated with the invoke activity is checked to determine whether its endpoint reference has already been set or not. The LB returns an endpoint reference (bean); its information is set as the partner reference of the used *partnerLink* before the call is executed.

B. Load Balancer

The LB is the only connection between the BPEL engine (i.e., the DR) and the load balancing solution. No direct access to the scheduler, provisioner, or other components is necessary. This provides exchangeability and extensibility, since the components do not largely depend on each other. Furthermore, all described components implement interfaces (`IPerformanceMonitor`, `IProvisioner`, `IScheduler`) to ease exchangeability.

The LB is implemented as a singleton per BPEL engine; it is instantiated via a factory, and so are the other components like the registry and the provisioner. A configuration file is used to set the actual implementations, which are then loaded using Java reflection.

When the LB is invoked (step 1 in Figure 3), it first queries the registry (2) for services matching the given port type, represented as a `QName`. A list of endpoints, encapsulated in `EndpointReferenceBeans` (ERBs), is returned (3) and passed to the registry's performance monitor (4). It hands back a list of `double` values representing the target hosts' load (5). The LB then passes the two lists and parameters, such as the mentioned threshold, to the scheduler (6); the configured scheduling algorithm is employed, and the endpoint of the computed target host is returned to the DR (7, 8). If

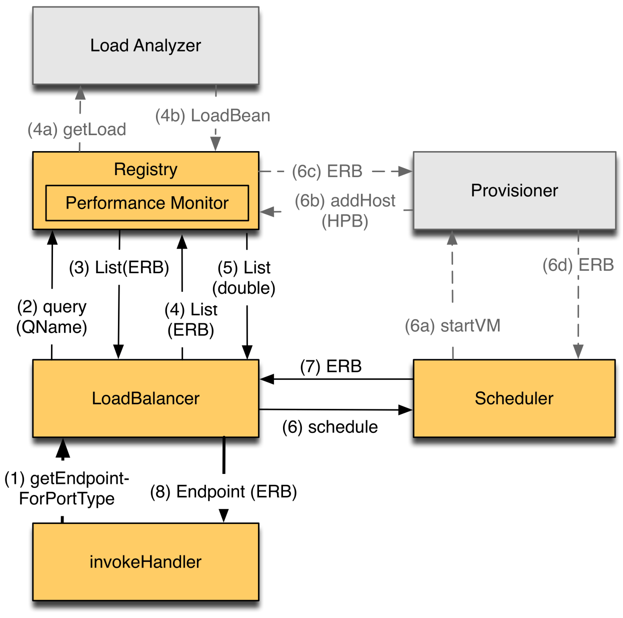


Fig. 3. Sequence of calls to determine a dynamic endpoint

the scheduler decides to start additional virtual machines, the provisioner is invoked (6a). After starting a new virtual machine, it is registered (6b, 6c) and the corresponding endpoint is returned (6d).

To assure cost-effectiveness, the scheduler monitors virtual machines and, if the overall load is low, tells the provisioner to shut them down before expiration of an accounting period. The system has to take into account that there must not be any running service calls on the machine to be shut down, which is not trivial to determine. For example, consider an asynchronous call to a service that results in an asynchronous callback when the service has finished the computation. To solve this problem, the DR increments a counter for each host in the registry that represents the number of active calls to services on that host. When a process finishes, the process graph is examined for invoke operations. For every invoke operation, the target host is determined and the counter value is decremented. The scheduling algorithms simply have to check whether the counter for the virtual machine to be shut down is zero, before telling the provisioner to shut it down. Therefore, the implementation needs to be notified about the termination of processes to decrement the per-host counter that prevents the virtual machines from being shut down. For this purpose, the LB registers itself as an event-listener in the BPEL engine (`IAeEngineListener`). On process termination, `handleEngineEvent` is invoked; the host names of the finished process' invocation targets are extracted and the registry is invoked to decrement the lock counter.

1) Registry: The registry manages available information on hosts and services and keeps them persistent. It is configurable via an XML document that is used to set up a set of permanently available hosts. Hosts are managed by so-called `HostPropertyBeans` containing information about the startup time (relevant for virtual machines), the virtual machine type, the type of middleware (*serverType*) and so on.

When a host is added – either by loading the static hosts or by the provisioner – the LA is queried to acquire a list of all services exposed by this host. Depending on the type of middleware (*serverType*) on the target host, different clients are used to query the service list and system load. Currently, two clients for plain web services and WSRF-based services exist.

When the registry is queried for a specific *portType*, the list of available services is scanned. Furthermore, external registries are queried, if configured. The access to them is realized by an interface `IRegistryConnector`. A list of all services matching the given *QName* is returned.

2) Provisioner: Since the provisioner has been designed to support several backends, it is organized into two parts: (1) an abstract class provides two methods `start` and `stop` that handle backend-independent tasks like registering and deregistering hosts. Before that, they call `_startVM` and `_stopVM` in the implementing sub-class (2) whereby the start operation returns an ERB with information needed to register the machine. In addition, the abstract class is responsible for the persistence of running virtual machines to avoid information loss due to a restart of the LB component.

The `_startVM` method must not return before the virtual machine and the middleware have booted. Otherwise, the DR or the workflow engine might try to invoke a service on the new virtual machine prior to its availability. It returns as soon as the service can be successfully invoked or aborts after several retries.

As an implementation example, the prototype implements a backend for Amazon EC2 [1]. It makes use of "Typica" [7], a Java-based open source library developed at Xerox. The implementation is able to start and stop virtual machines, configure the virtual machines' firewalls (some ports need to be open for SOAP communication and all other ports should be closed) and execute user scripts after booting a virtual machine. Since Amazon Machine Images (AMI) cannot be altered after creation, we have developed a script that downloads the middleware from an Amazon Simple Storage Service Bucket (S3, [2]). This assures that the middleware can easily be upgraded (or new services may be installed) without the need to create a new AMI. The middleware is downloaded from S3, since external traffic (like a *wget* from an external server) is billed and Amazon-internal traffic is not billed.

3) Scheduler: In the following, the implementation of a simple scheduling strategy is discussed as an example.

The input values for the scheduler are the names of matching hosts (M), load values (load) and a threshold value t. The scheduler first computes all qualified hosts M_q, where $M_q := \{m \in M | \text{load}_m < t\}$. If M_q is non-empty, for all hosts in M_q, the free capacity

$$c_m = \frac{WIPS_m}{max(\text{load}_m, 0.01)}$$

is computed. *WIPS* stands for Whetstone Instructions Per Second and is computed using the Whetstone benchmark that performs integer, floating point and array operations to determine the performance of a host. The value of 0.01 is required for the case that a host has a load of 0. Then, the host with the highest value for c_m is chosen, meaning that the host with the lowest utilization is selected. Otherwise, if M_q is empty, the scheduler requests a new host from the provisioner. After the startup of the new virtual machine, the waiting call is immediately scheduled to the host.

C. Load Analyzer

The LA in our prototype has been realized for Linux. It inspects `/proc/loadavg` and `/proc/cpuinfo` to determine the *Load Average* and the core count. In virtualized environments, the computed percentage of CPU usage cannot be used as an indicator for CPU load, in contrast to *Load Average*. The machine may only have been assigned a part of the physical computing power – in this case, the percentage of CPU usage would always be below 100%. The *WIPS* benchmark is executed in order to approximate the system's performance. For cluster environments, only a prototypical implementation is currently available. It queries the Monitoring and Discovery System of the GT4 to acquire information about available and used cores.

V. Experimental Results

In this section, experimental results with respect to the performance of our load balancing approach are presented. Our sample application is a real-life video analysis workflow that performs face detection and text recognition in MPEG videos. More information can be found in Seiler et al. [23]. This workflow consists of four main services. These are:

- Decoder: splits an MPEG video as the input of the workflow into single frames.
- Face Detector: analyzes a single frame for human faces and returns the result as an MPEG-7 element.
- Text Recognizer: scans a frame for text and returns the result as an MPEG-7 element.
- MPEG-7 Converter: merges all information given by the Face Detector and returns an MPEG-7 document.

The services are realized as WSRF services based on the GT4 [3]. The data exchanged between the services contains video frames. For efficient data transfer between these services, Flex-SwA [19] is used. When using SOAP messages, the binary data, i.e., the video frame, has to be encoded in order to be transmitted via a SOAP message. Binary data in SOAP messages is encoded as BASE64 – with the consequence that the message is bigger than the data itself. When Flex-SwA is used, only a reference is transmitted via the SOAP message, and the binary data is transferred from host to host by the Flex-SwA middleware. Thus, the engine does not represent a bottleneck for transferring data.

To orchestrate the Grid services, WSRF extensions for BPEL [12] are used. To integrate the WSRF extension as well as the LB, the *invokeHandler* of the factory link is set to the LB; the resource link is kept dynamic and therefore determined by the *createResource* operation. Figure 4 shows the design of the workflow using the Visual Grid Orchestrator (ViGO) [13], a BPEL editor for developing Grid workflows. It supports the user in several ways, e.g. by providing wizards to create an invoke operation for WSRF services.

The workflow engine and all virtual machines are hosted by Amazons EC2. This allows to perform a computationally intensive video analysis workflow without having an in-house infrastructure and, more importantly, eliminates network latency between Europe and the USA in the measurements.

Some of the video analysis services make use of native C/C++ code. The decoder utilizes the OpenCV library [6]. It is quite complicated to deploy the service on different platforms, because the native code has to be recompiled for each target platform. Virtual machines make this task much easier. A virtual machine image for the video analysis workflow has been created that is booted by all hosts. In this way, it can be assured that all hosts are capable of executing the native code.

The workflow has been run more than 150 times; in more than 40 runs new virtual machines were booted due to high load of existing machines. For the first measurements, the least powerful host type available at Amazon EC2 has been chosen. Such a "small instance" has 1.7 GB of RAM, 160 GB hard disk, runs a 32 Bit Linux and costs \$0.10 per hour. The computing power is specified in "EC2 Compute Units" (ECU). Small instances have 1 ECU, which is comparable to a 1.2 GHz Xeon 2007 processor [1].

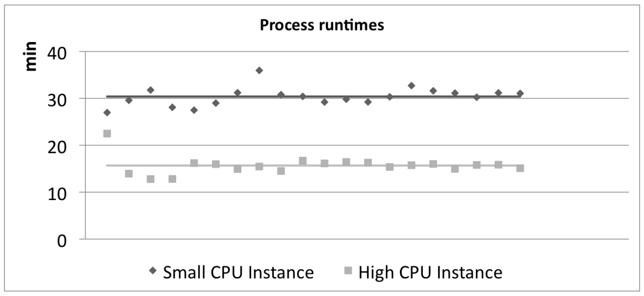
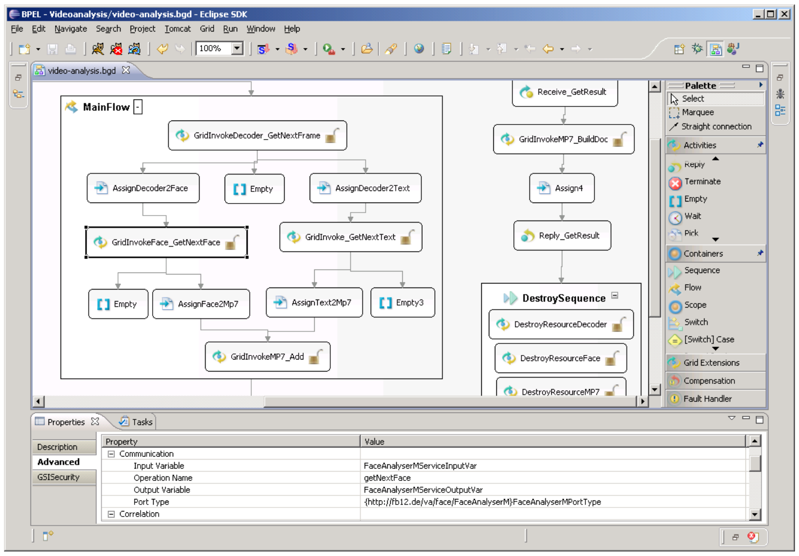


Fig. 5. Process runtimes for different host types

The average runtime including the startup of new virtual machines was 1821.49 sec with a standard deviation of 95.9 sec (see Figure 5). Contrary to "normal" BPEL workflows, the runtime not only consists of the computation time, but also of the boot time of virtual machines and the time needed for the deployment of the middleware. The boot time of a virtual machine running on small instances is 37.57 sec. The provisioning and the start of the middleware take another 51.17 sec. Compared to the runtime of the video analysis itself (30.17 min with a standard deviation of 1.69 min), the total average startup time of 88.74 sec is negligible. Once a virtual machine has been started, it can be used by several workflows, which further relativizes the overhead.

To compare the different host types offered by Amazon, the computation was also performed on "High-CPU Medium



Property	Value
Communication	
Input Variable	FaceAnalyserMServiceInputVar
Operation Name	getNextFace
Output Variable	FaceAnalyserMServiceOutputVar
Port Type	{http://fb12.de/va/face/FaceAnalyserM}FaceAnalyserMPortType
Correlation	

Fig. 4. Designing the video analysis with the Visual Grid Orchestrator

Instance” (2 × 2.5 ECU, more storage, \$0.20 per hour). The average process runtime with the high CPU instances is 15.69 min (standard deviation: 1.9 min). The average boot time of such a virtual machine is 33.36 sec and the container needs 31.37 sec to start (see Figure 6). Although a speedup of 4 to 5 could be expected, a speedup of about 2 has been measured. This is due to the fact that the used libraries are single-threaded and therefore do not make use of the additional CPU core. The speedup is less than linear due to network latency.

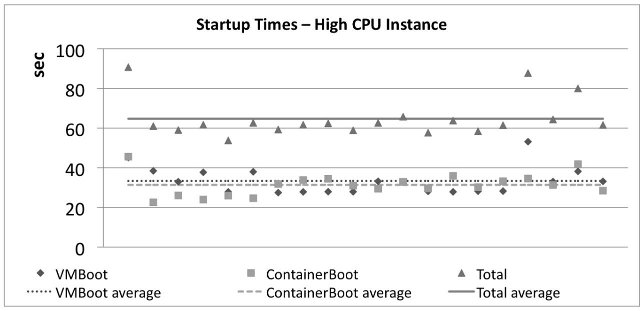


Fig. 6. Virtual machine and middleware boot times

In addition, the runtimes of the different components have been analyzed separately. A registry query takes around 2.5 msec; determining the load takes around 100 msec per host. If the load is cached, the time is reduced to about 1 msec, which is negligible. Making the scheduling decision takes about 1 msec, except when a new virtual machine has to be booted.

To sum up, our approach provides efficient load balancing capabilities. If, without our solution, multiple workflow instances are run in parallel, the runtime increases linearly with the number of instances. Using our solution, the runtime remains almost constant (the provisioning times vary slightly, as shown in Figure 6).

VI. Related work

Di Penta et al. [11] have presented a dynamic binding framework called WS Binder. It allows the (re-) binding of *partnerLink*s to services during the runtime of a process. For this aim, the authors use a proxy architecture. Based on a given policy, the framework is able to schedule an invoke operation to a specific service. This binding can either be done before or during the execution of the workflow. In addition, runtime recovery can be supported. However, the framework does not provide support to extend the pool of usable services, i.e. by booting a virtual machine.

TRAP/BPEL is another framework for dynamic adaptation of service compositions presented by Ezenwoye and Sajadi [16]. It makes use of a generic proxy pattern. The proxy is able to query a UDDI registry to find a suitable service. The binding is performed in an autonomic fashion, e.g., if a service call fails, a substitute is determined and the call is retried. Furthermore, the selection of a suitable service can depend on a policy. To make a BPEL process interact with the TRAP/BPEL framework, it has to be adapted. Although policies can be used for service selection, there is no possibility to specify post-invocation policies, e.g. for fault handling. Furthermore, dynamic scaling of the infrastructure is not supported.

By introducing a new element (*find_bind*) into the BPEL language, Karastoyanova et al. have presented their approach for runtime adaptability [20]. The mechanism is able to find services, e.g. by querying a UDDI registry. Based on policies, it selects suitable services and binds them to process instances. In case a service call fails, a process instance repair is guaranteed by rebinding to another port. Selection criteria can be modified at runtime. There is no support for dynamically providing additional target hosts.

Ma et al. [21] have presented a Grid-enabled workflow management system that is based on BPEL. The system allows interaction with stateful resources, dynamic service binding and scalability of workflow execution. In the scenario presented in the paper, the BPEL engine is a bottleneck. Scalability is achieved by placing a load balancer in front of a cluster of BPEL engines. Calls to the workflow engine are then scheduled with respect to the engines' workload. Dynamic service binding at runtime is achieved similar to our approach. At runtime, a provisioning service is contacted that looks up a host where the requested service is installed. However, the approach does not take the workload of target hosts into account. Furthermore, it does not provide any feature to dynamically provide additional target hosts.

Mietzner and Leymann [22] have discussed the problem that no standard for a generic provisioning infrastructure exists. It is argued that an architecture is needed that allows to deploy applications in different environments independently of the actual provisioning engine. The authors introduce a set of services related to provisioning and BPEL-based workflows that make use of these services. Once widely accepted and implemented, the proposed architecture could substitute our provisioning component. This would significantly reduce the implementation efforts.

VII. Conclusions

In this paper, the design and implementation of a workflow system based on BPEL to support on-demand resource provisioning was presented. Our approach automatically schedules workflow steps to underutilized hosts and provides new hosts using Cloud computing infrastructures in peak load situations. An implementation based on the ActiveBPEL engine and Amazon's Elastic Compute Cloud was presented. Experimental results for a computationally intensive video analysis application have demonstrated the feasibility of our solution.

Future work includes the integration of further backend implementations, such as Globus Virtual Workspaces. Furthermore, sophisticated scheduling algorithms need to be developed. The scheduler should, for instance, schedule tasks with regard to data dependencies between workflow steps, such that unnecessary data transfers could be circumvented. Instead of providing machines when a call is to be executed, the system could perform advanced provisioning with respect to the overall system load. Having a spare machine would be especially useful for workflows with many tasks that require a short time to execute. The provisioning time would not affect the execution time of a service call. A failure handling and recovery approach that takes advantage of the presented solution to dynamically provide resources is another interesting area of further research.

VIII. Acknowledgements

This work is partly supported by the German Ministry of Education and Research (BMBF) (D-Grid Initiative).

References

[1] Amazon Web Services LLC, Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[2] Amazon Web Services LLC, Amazon Simple Storage Service (S3). `http://aws.amazon.com/s3/`.

[3] Globus Toolkit, Project Homepage. `http://www.globus.org/toolkit/`.

[4] Grimoires, Project Homepage. `http://www.grimoires.org`.

[5] OASIS: Web Services Resource Framework 1.2 (WSRF). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf`.

[6] Open Compuer Vision Library, Project Homepage. `http://opencv.willowgarage.com/wiki/`.

[7] Typica, Project Homepage. `http://code.google.com/p/typica/`.

[8] ActiveEndpoints. ActiveBPEL Business Process Execution Engine. `http://www.activebpel.org`.

[9] A. Akram, D. Meredith, and R. Allan. Evaluation of BPEL to Scientific Workflows. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 269–274. IEEE Computer Society, 2006.

[10] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1*. Microsoft, IBM, Siebel, BEA und SAP, 1.1 edition, May 2003.

[11] M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. D. Nitto. WS Binder: a Framework to Enable Dynamic Binding of Composite Web Services. In *Proceedings of the 2006 International Workshop on Service-oriented Software Engineering*, pages 74–80. ACM, 2006.

[12] T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, and B. Freisleben. Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In *Proceedings of German e-Science Conference (GES)*, pages 1–8, 2007.

[13] T. Dörnemann, M. Smith, E. Juhnke, and B. Freisleben. Secure Grid Micro-Workflows Using Virtual Workspaces. In *Proceedings of 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 119–126. IEEE Press, 2008.

[14] J. Elson and J. Howell. Handling flash crowds from your garage. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[15] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price. Grid Service Orchestration using the Business Process Execution Language. In *Journal of Grid Computing*, volume 3, pages 283–304, September 2005.

[16] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In *In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, 2007.

[17] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2006.

[18] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang. Virtual Clusters for Grid Communities. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 513–520. IEEE Computer Society, 2006.

[19] S. Heinzl, M. Mathes, T. Friese, M. Smith, and B. Freisleben. Flex-SwA: Flexible Exchange of Binary Data Based on SOAP Messages with Attachments. In *Proc. of the IEEE International Conference on Web Services, Chicago, USA*, pages 3–10. IEEE Press, 2006.

[20] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending BPEL for Run Time Adaptability. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, pages 15–26. IEEE Computer Society, 2005.

[21] R.-Y. Ma, Y.-W. Wu, X.-X. Meng, S.-J. Liu, and L. Pan. Grid-Enabled Workflow Management System Based On BPEL. *International Journal of High Performance Computing Applications*, 22(3):238–249, 2008.

[22] R. Mietzner and F. Leymann. Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In *Proceedings of IEEE Congress on Services - Part I*, pages 3–10, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[23] D. Seiler, S. Heinzl, E. Juhnke, R. Ewerth, M. Grauer, and B. Freisleben. Efficient Data Transmission in Service Workflows for Distributed Video Content Analysis. In *Proceedings of the 6th International Conference on Advances in Mobile Computing & Multimedia (MoMM2008)*, pages 7–14. ACM and OCG Book Series, 2008.