RESTful Business Process Management in the Cloud

Alessio Gambi*†
a.gambi@infosys.tuwien.ac.at
† Vienna University of Technology, Austria

Cesare Pautasso*
cesare.pautasso@usi.ch
* University of Lugano, Switzerland

Abstract—**As more and more business processes are migrated into cloud-based runtimes, there is a need to manage their state to provide support for quality attributes such as elasticity, scalability and dependability. In this paper we discuss how the REST architectural style provides a sensible choice to manage and publish service compositions under the Platform as a Service paradigm. We define the design principles of RESTful business process management in the cloud and compare several architectural alternatives to support elastic processes which can be monitored and dynamically adapted to workload changes.**

Index Terms—**Cloud PaaS, RESTful BPM, elasticity**

I. Introduction

Composition as a Service (CaaS) [26], WorkFlow as a Service (WFaaS) [19] and Business Process as a Service (BPaaS) [30] are all emerging technologies that enable clients to outsource the modeling and execution of processes into the Cloud. Such dedicated runtime platforms for service composition have a need to manage the state of a very large number of process instances and provide support for quality attributes such as elastic scalability and dependability that have come to be expected by existing Platform as a Service (PaaS) offerings. Existing centralized and distributed business process execution architectures only provide limited support for exploiting the potential offered by large Cloud infrastructures [2], [7].

The main focus of this paper is the investigation on how composite services can be run on Clouds while exploiting the full potential of Clouds, that is, what design principles and architectural choices can be followed to implement "native" Cloud composite services [28]. In this paper we discuss how the notion of RESTful business process management (RESTful BPM), derived from the application of the REST architectural style to the composition of multiple services into business processes, can provide a fresh look at how to design the architecture of business process execution engines so that these can better fit with the requirements and constraints of modern Cloud runtimes. In particular, the goal is to distill the design principles behind a novel execution platform for Business Processes to be delivered as a Service (BPaaS) that is compliant with the constraints of the REST architectural style and to discuss the consequences on the elasticity, dependability and performance of the possible architectural alternatives that can be derived from these principles.

This paper makes the following contributions. It makes the case for using REST to publish business processes in the Cloud (so that clients may have full access to the state and full control over the execution of their processes) but also within the Cloud (so that the BPaaS infrastructure may efficiently replicate, migrate and consolidate their state) to deliver the expected elastic scalability and dependability. It presents the design space for a BPaaS platform with different concrete architectures trading off performance against dependability and illustrating the variety of options that become available to effectively manage the state and the execution of RESTful business processes.

The rest of this paper is structured as follows. After introducing background concepts of REST services, RESTful BPM, and Cloud, we identify main design principles of RESTful business process management in the Cloud. We compare several architectural alternatives to support elastic processes which can be monitored and dynamically adapted to workload changes, and discuss their impact on the quality attributes. Before drawing our conclusions we survey existing related work.

II. REST, Processes and Clouds

A. *On REST and Web Service composition*

The REpresentational State Transfer is an architectural style that recently gained a lot of attention in the field of Web service design [12]. By defining architectural constraints such as stateful resources, stateless interactions, global identification of unique resources, uniform interfaces and multiple resource representations, the REST style promotes the design of resource-oriented architectures [31]. These architectures are characterized by their intrinsic interoperability, loose coupling, high scalability and flexibility.

The REST style can also be suitably employed in the design of service-oriented architectures (SOA) (e.g., [9]). When REST and SOA meet, a novel, light-weight form of services is obtained: RESTful Web services [25]. RESTful Web services differ in several aspects from the abstractions provided by other styles. As opposed to promoting stateless services that interact by means of synchronous or asynchronous message exchanges, REST emphasize a uniform approach to access stateful services by means of well defined methods [33]. Interactions with RESTful services are compliant with the REST style: all the interactions between client and services are stateless, while the state of the system is either persisted at the service (associated with a resource identifier) or kept

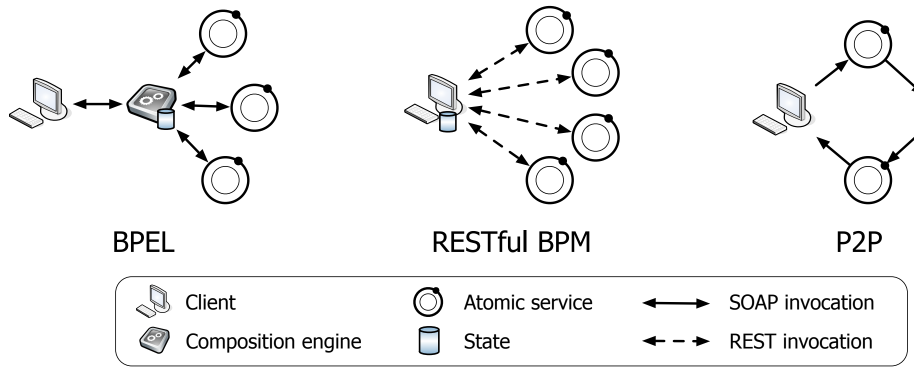

Fig. 1. Different styles of Web service compositions (BPEL-like, centralized; RESTful BPM, client-side; P2P, decentralized)

on the client-side. Services publish resources, whose state is externalized to clients that can retrieve it (and in general, modify it) according to the uniform interface semantics and their representation preferences. Clients can navigate across related resources by following hyperlinks (resource identifiers found within representations of the service state). For example, the result of a client request can be asynchronously published as a resource so that multiple clients can refer to it by sharing its resource identifier (URI).

Concerning the composition of multiple services into processes, Figure 1 exemplifies some of these differences. BPEL-like services are characterized by a centralized architecture, usually implemented as an orchestration engine, that manages the state of the composition and invokes the atomic services that are bound to it. Choreography-style services are fully decentralized. The state of the composition is shared among the participants and directly forwarded along the execution path of the composition. The client initiates the composition by sending a message to the first component and will eventually receive the result from the last component at the end of the process. Unlike the centralized approach, with P2P compositions there is no single place where the composition state can be found at one time [27]. Since services communicate directly with each other, the communication and scalability bottleneck of the centralized engine is removed [24].

RESTFul service compositions introduce a different approach, whereby the state of the composition is managed by the client. As the client follows the hyperlinks provided by each participant, it dynamically discovers the process composing them without the need for a dedicated engine [1]. The execution of the process is driven by the client that directly contacts all the atomic services involved [32].

B. RESTful Business Process Management

RESTful services enable a novel form of business process management, namely RESTful BPM, that exploits the peculiarities of the REST style not only to invoke and orchestrate a set of RESTful Web services, but also to publish processes on the Web as resources [16], [21]. More in detail, the process followed by a client that interacts with multiple resources can be published itself as a resource, thus shifting once again its execution from the client to the server. However, as opposed to a traditional process execution engines, which provide a stateless, message-oriented interface to invoke the process published as a service, the state of the execution of a business process can be published with a RESTful Web service interface [20].

To do so, unique resource identifiers are minted and associated with the process, its instances, its tasks and subprocesses and clients are offered a rich and expressive interface abstraction to interact with and control the execution of each of their process instances, whose state is dependably offloaded onto the server (running in the Cloud). For example, basic HTTP verbs, such as GET, POST, PUT, DELETE are mapped to the lifecycle of processes and their tasks. POST requests to a process template URI result in the instantiation of new process instances; DELETE may be used to cancel the execution of a process instance; GET may retrieve its current state, which – depending on the chosen representation – can include hyperlinks to its active tasks and to retrieve its execution results. These links can be shared among multiple clients and can be protected with various forms of authentication and access control. More in detail, the state of individual tasks can also be manipulated by means of PUT requests that can drive forward the execution of the process published as a RESTful service.

In this paper, we argue that RESTful BPM can be used to provide a suitable design for runtime environments to run composite services on the Cloud. Qualities such scalability, flexibility, and explicit management of state resource, which RESTful BPM inherits from the REST style, fit the requirements on elasticity, monitorability and scalability that are characteristics of Cloud applications.

C. On Cloud computing

Cloud computing is the game changing technology that is supposed to deliver significant advantages for final users and service providers, mainly through enabling strong costs savings and increased agility in the dynamic provisioning of IT infrastructures. The basic principles of the Cloud computing paradigm are: i) outsource and automate all the management activities that are not related to the primary application business logic, ii) access elastic infrastructures where a potentially unlimited set of remote resources can be allocated on-demand and on-the-fly to scale applications, and iii) pay for what is actually used.

Depending on what it is outsourced to the Cloud, three main delivery paradigms are identified [3]. Under the *Infrastructure as a Service* (IaaS) paradigm, Clouds deliver computing, storage and networking resources and take care of their runtime management. With *Platform as a Service* (PaaS), Clouds offer complete runtime environments where application providers can run their applications if they are developed according the cloud-specific architectural constraints and framework APIs (e.g. no direct calls to OS, no threads, and so on). Clouds, in turn, take care of deploying these applications to computing nodes, dynamically allocate the "right" amount of resources to them depending on their actual workload, and manage a series of background activities to support them. For example, Cloud providers offer secure communications, data privacy in multi-tenant environments, periodic backups of data, and so on. Under the *Software as a Service* (SaaS) paradigm, Clouds offer fully fledged applications to clients that access them thought thin clients, usually their browsers. In this case, Clouds take care of maintaining the whole software/hardware stack as well as providing storage for user data.

Usually, Clouds are organized in layers, where upper layers exploit the functionalities of lower layers. Accordingly, SaaS is implemented using PaaS, and PaaS "underneath" exploits IaaS. However, to fully benefit from Clouds, applications running at the different layers must be designed according to specific principles and patterns [10], [11]. Otherwise, Clouds may be a double-edged weapon as applications consume more resources than expected, scale less than expected, and inelastically retain resources after peaks in the load passed away [15]. In other words, running elastic systems based on the Cloud is not a simple matter of installing software components inside virtual machines, and deploy them on the Cloud. The same holds for frameworks and runtime systems for Web service composition.

III. Service Composition on the Cloud

The main requirements and features that Cloud users expect from a Cloud provider of a runtime platform for the deployment and execution of composite services are dependability and elastic scalability within a given quality of service envelope. In the next sections, we will show how RESTful BPM can address these requirements and discuss several alternative architectures for the implementation of the required features.

We argue that the PaaS layer is where business processes and Clouds can show the most suitable combination: Service providers and service aggregators reason in terms of high level composition languages, such as BPEL [17] or JOpera [22] or ROsWeL [5], and would rather focus on the declarative definition of the business processes, workflows and service compositions as opposed to ensuring they can be implemented with the appropriate level of dependability and scalability. The necessary runtime management capabilities are delivered by the Cloud infrastructure, providing for the elastic scalability and dependability of the composite services. Merging the two, we obtain platforms where service providers send to Clouds high level specifications of their composite services, and the Cloud takes care of publishing, running, and managing them. These activities are complementary to the ones provided in the context of Composition as a Service (CaaS) tools [26], [4]. CaaS targets the SaaS abstraction level with applications where users provide partial service compositions and specifications which are then used to generate recommendations about composed services, and public service *fragments* that are available to be reused.

An ideal Cloud-based platform for the management of composite services under the PaaS paradigm should provide a runtime environment that is able to correctly run user provided service compositions by dynamically optimizing its internal infrastructure for minimal resource consumption and maximum performance. Service providers should be able to specify high level conditions, in terms of Quality of Service (QoS) guarantees and cost models that will drive the runtime management of the Cloud, i.e., the *elasticity* of their services, without explicitly describe how these can be achieved [8]. For example, a service provider can specify a given response time for a service and the Cloud may dynamically allocate the right amount of resources to provide it in face of workload fluctuations. Likewise, the Cloud may time-shift the execution of low priority processes as well as run dependent services in co-located computing nodes, or move services compositions close to the external Web services that they depend on [13]. Alternatively to specify high-level properties, service providers may assume a best effort management where the Cloud is entirely responsible to balance the trade off between costs, resources and QoS. This is similar to what the Google PaaS platform, Google App Engine[1], actually does.

Composite services provided by users must be able to invoke other services. These can be both atomic or again composite services. In any case, these may be running in the same Cloud, in a separate Cloud or outside any Cloud. If both atomic services and the compositions composing them are run by the same PaaS, Cloud providers may perform optimizations that exploit their co-location, i.e., atomic services can be dynamically provisioned by means of migration and replication. If some of the services involved in a composition run in different infrastructures, there may be more limited opportunities for optimizations and these may depend on the actual characteristics of the atomic services, e.g. composed services can be moved towards external services to reduce latency, or specific caching solutions may be introduced at the boundary of the Cloud.

Cloud providers may also act on the runtime structure of composed service as long as the final semantic is preserved. For example, if a service contains a variable number of independent subprocesses (or tasks) that can be run in parallel, the Cloud provider may autonomously decide on the number of concurrently active subprocesses on the base of cost and

[1]https://cloud.google.com/products/index

QoS constraints, and find a balance between them. This is similar to what is done currently to dynamically optimize Map-Reduce or Hadoop[2] based systems [14]. In our context, sub-processes and tasks that enable such a runtime behavior can be defined as *elastic sub-processes* and *elastic tasks*. We discuss in the next section, how we can design them within the context of RESTful business process management.

IV. Design principles for RESTful BPM inside Clouds

In native Cloud applications, qualities like scalability, elasticity and dependability are provided by resorting to tailored software architectural decisions, design principles, and patterns [28], [11]. Native cloud applications employ modular architectures that allow application components to scale independently; they leverage loose coupling of components, asynchronous communications, and stateless interactions to enable scalability and to increase the overall system dependability. They are usually implemented over backbone queue-based messaging systems that maintain the decoupling of components and absorb communication spikes with distributed buffers. Components externalize their states, are organized according in a shared-nothing fashion, and make heavily use of multi-threading and parallelism [29].

We acknowledge that similar architectural choices and design principles should be applied in the design of Cloud runtime environments for running composite services. In particular, we stress the fact that the REST architectural style promotes stateless interactions and enhances the loose coupling of distributed components. Therefore it enables, through the abstraction of RESTful Web services, a design where services can be flexibly provisioned and deployed over a set of elastic computing nodes almost for free. However compared to other kinds of Cloud native applications, business processes have some peculiarities that must be considered with particular care.

In our context, *long-livedness* of composed services is one of the most important peculiarities to consider as it strongly impacts the system elasticity. With long-lived processes delivered as a service, it may happen that the Cloud cannot quickly free up computing nodes because active processes, tasks and services are distributed across them. For example, a task instance that is waiting for user input, or is blocked while retrieving the result of a remote service invocation, may prevent a server to be de-allocated even if its resource usage is below the minimum usage threshold. Compared to ordinary applications where sessions have shorter duration and where transactions can be easily rolled back and repeated, removing computing nodes that run long-lived services may result in aborting entire process instances. The resulting revenue losses may offset the cost reductions obtained with the resource consolidation in the Cloud.

Long-livedness therefore demands for proper primitives to manage the state of composite services, including their sub-processes and tasks. In particular, there is a need to *replicate*, *migrate*, and *consolidate* the state of composed services across computing nodes, in a way that is similar to the live-migration of virtual machines at the infrastructure level, but which also takes into account the state of the external interactions. In this work, we argue that by explicitly representing services, tasks, and sub-processes as resources under the RESTful Web service abstraction, we can leverage RESTful BPM for their runtime management, i.e., for suitably dealing with their long-lived nature. In particular, we propose to implement the migration of the state of services, tasks and sub-processes by means of *elastic URIs* and REST distributed transactions [18].

Elastic URIs provide a concept similar to Amazon's Elastic IP[3] applied to REST resources. Resources are exposed to users outside the Cloud using public URIs that are mapped to private resources by means of a private URI. The mapping is managed by the Cloud and dynamically updated every time a resource, e.g. the state of a composite service, is migrated inside the Cloud. The fact that states representing running service instances must be dynamically managed by the Cloud motivates the coupling of RESTful BPM and PaaS.

Depending on the actual state of the resource and its type, the Cloud may decide to keep it on a computing node, or may delegate its management to data services provided by the Cloud itself. For example, a resource referring to an active service composition can be placed on a running virtual machine first, and moved to the persistent layer of the Cloud once the process ends, thus freeing computing resources. The resource representing the service instance is available from the data store until a valid DELETE request is received. The private URIs that located the resources inside the Cloud are different, but the public URI remains the same. The public URI is elastic because it can be adapted dynamically. It must be noted however that not all the public URIs must be elastic.

RESTful BPM makes it possible to adopt the same approach to manage service compositions that are in an idle state: as long as the process is waiting for some external event, its state may be managed by the persistence service to free up computing resources; once a request is issued on its public URI, the waiting resource can be migrated back to a compute node before letting it process the request and evolve to its new state. RESTful BPM enables also the migration of resource state across computing nodes while processes are active, as long as the migration is scheduled while there are no active or pending interactions with external services. In fact, interactions are stateless, and in between them the resource state does not change. At this point, a resource that refers to an active composition instance can be moved to a different computing node, and the process can continue from there. Elastic URIs maintain a consistent view from outside the BPaaS. In any case, distributed REST transactions are used to atomically migrate resources across locations while preserving state consistency.

[2]http://hadoop.apache.org/

[3]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

Similar primitives can be used to perform the *live cloning* of compositions and tasks, further improving system scalability and effectively implementing elasticity at service level. Whenever the semantics of tasks and sub-processes allows it, these resources can be duplicated and distributed around the infrastructure to parallelize computations on-demand, and according to the high-level directives contained in the business process model. This can be implemented inside the Cloud by allowing the Cloud to issue POST and DELETE requests for tasks and subprocess, and by delegating shared dependencies, e.g., big inputs data sets, to the Cloud data services. The actual number of active task or sub-process instances is entirely managed by the Cloud and it remains transparent from the client of the process.

Dependability can be achieved by ensuring that the Cloud persistent storage service keeps a consistent and possibly redundant copy of the state of the process at each step. In case of failures and crashes of the computing nodes responsible for executing a given process or task instance, its state can be rolled back to the one managed by the underlying Cloud storage and the execution can be restarted from there without any intervention of the client of the process. Parallel and concurrent access to the state of the process instance is managed accordingly to the semantics of the REST uniform interface, in which read-only safe operations are clearly marked as such.

Effective elasticity requires prompt reactions to changes in the environment, specifically in the workload, that in turn require efficient monitoring of the service, task and sub-process state. RESTful BPM enables this by means of push notifications [23]. Furthermore, RESTful BPM also enables an easy recording of the evolution of compositions, for debugging or mining activities, by allowing state versioning. To improve storage and management of service history, compact or differential representations of multiple states can be chosen.

V. Architectural alternatives

We identify and discuss some basic design alternatives, and as a second step combine them into possible architectures that emphasize different trade-offs among specific quality attributes.The architectural alternatives concern two dimensions: the management of the *state* and the actual *execution* of instances of composite services as well as their sub-processes and tasks.

1) State: The state of a RESTful business process is usually stored and managed by the clients executing the process itself. Alternatively, the execution of the process can be delegated to the Cloud. Furthermore, as an independent, orthogonal decision, the state can be co-located with the processing units that evolve it, or it can be externalized through a data storage abstraction provided by a Cloud service provider.

Figure 2 shows three possible state locations when clients manage the process execution: Client C1 keeps the state locally, while clients C2 and C3 externalize the state management to Clouds. C2 is connected to the state by means of an elastic URI (depicted with a coil spring icon) that is transparently implemented by the PaaS. C3 uses a Cloud data service provider which does not support this feature. Still, it may enable features such as distributed caching, state replication and redundant persistent storage transparently with respect to the client. In the case of C2, the Cloud provider for the state storage and the service invoked by the composition is the same. This may also be a source of optimizations concerning the data flow of the process and help to reduce runtime operational costs due to network transfers.

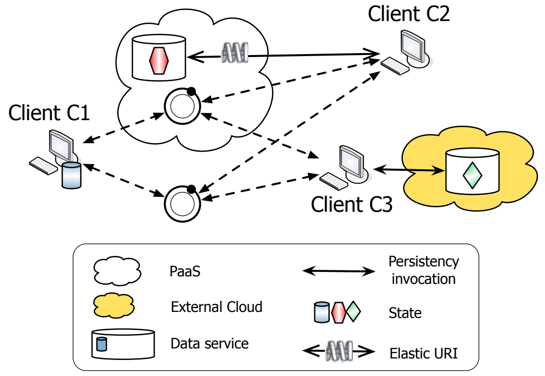

Fig. 2. Client-based state management: C1 co-located, C2 elastic cloud storage, C3 cloud storage

Figure 3 shows the available alternatives when clients outsource to PaaS the execution of the processes. For client C1 the state is managed locally by the same process engine (identified by a gears icon) that interacts with the atomic services inside and outside the Cloud. For client C2 the state is externalized to a local data service provided by the PaaS, and for client C3 the state is externalized to another Cloud provider. As composed services are entirely managed by the PaaS, clients can access service state and other service-generated resources by means of elastic URIs. The Figure for example shows clients C2 and C3 accessing the service states using elastic URIs. For the case of client C3 in particular, the PaaS can hide from the client the fact that an external Cloud provider is being used to store the state of its process. Alternatively, we can envision PaaS which can let the client control the choice and location of the storage service provider used to persist the process execution state of the client. Assuming that the external Cloud is more dependable than the PaaS, the elasticity of the URI in this case lets the PaaS provide a dynamically varying level of dependability to its clients.

2) Execution: Instances of composed services, their tasks and subprocess, as well as the RESTful services participating in the composition may run in different places by different actors. Composite services may run on client premises or inside the PaaS. If they run in the PaaS, they can be run elastically with computing resources dynamically adjusted to their demands. RESTful services that are bound by the compositions can be either atomic or complex, and can run inside or outside the PaaS Clouds. If they run in the PaaS, they can be made elastic and their runtime management can be aligned to the one of calling compositions. Tasks and sub-processes running in the PaaS and can be made elastic as well.

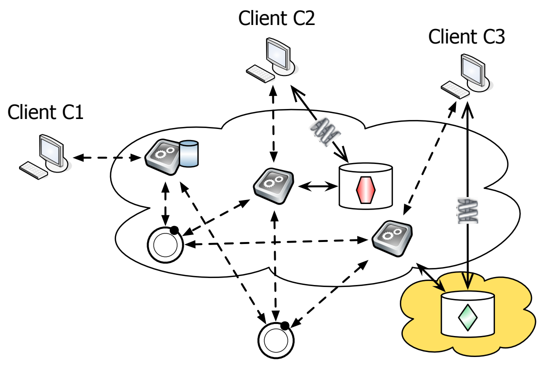

Fig. 3. PaaS-based state management: C1 co-locate, C2 elastic cloud storage (in the PaaS), C3 elastic cloud storage (through the PaaS)

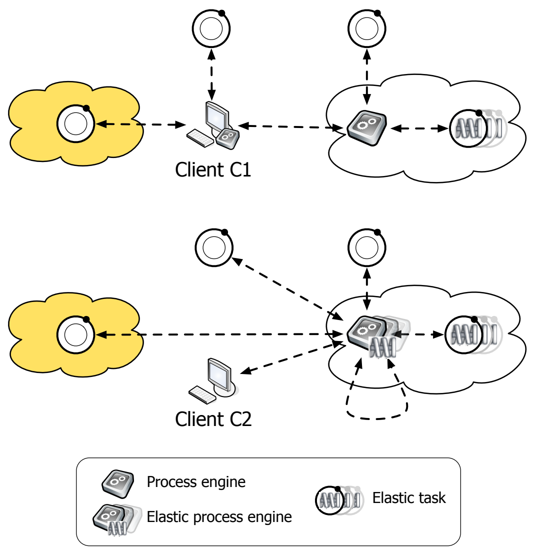

Fig. 4. Distribution of running instances of the composite service: C1 co-located, C2 elastic process engine

Composite services, tasks and sub-processes that run in the PaaS are implemented according to the same REST principles, therefore can be uniformly managed with RESTful BPM by the Cloud provider. The basic pattern adopted in the PaaS is to have parallel processes that run RESTful processes (and process fragments), and to dynamically allocate processes to computing nodes. As a matter of fact, the PaaS provider combines RESTful BPM with the management of the infrastructure layer, which in turns is implemented using REST. Moreover, thanks the use of standard technologies to implement REST architectures, all of these elements can easily interoperate provided that they can reach each other across the network.

Figure 4 shows two alternative architectures with respect to running service instances. In the first case, client C1 runs locally the composite service, and invokes other composite and atomic services deployed under different settings. In particular, the PaaS runs a composite service that invoke an elastic task and an atomic, but external, service. In the second case, client C2 delegates to the PaaS the execution of the composite service and the Cloud can exploit further optimizations: As the PaaS runs both the process and the one of the services bound to it, the Cloud can align their runtime management policies and thus facilitate achieving the elasticity of the process engine [13].

3) Architectures for Cloud-based RESTful BPM: The basic alternatives presented in the previous section can be combined to create different system architectures. However, not all their possible combinations may result in valid PaaS design, even if they can be implemented and run by leveraging Cloud technologies and RESTful BPM. A valid design for a PaaS system must have both states of composite services and their running instances managed by the Cloud. If the execution of the composite service is managed by the Cloud but the state is managed by the client, the design represents a weak form of PaaS. If the execution of the composite services is managed outside the Cloud then the design does not represent a valid form of PaaS for composite service.

Figure 5 shows representative architectures that belong to the different classes. The composite service depicted in the Figure is meant to call an atomic service and a composed service (i.e., a sub-process) inside the PaaS, and one service outside the PaaS. The bound composed service has a dependency outside the Cloud. The PaaS can execute a task elastically, and this task publishes its results as a resource that the client wants to retrieve. The architectures on top of Figure 5 are valid designs for PaaS because in both cases clients delegate the execution of their composite services and the state management to the PaaS. The bottom part of the Figure reports on the left a weak PaaS architecture where clients delegate the execution of the composite to the Cloud but maintain the state co-located; and on the right, an architecture that is not a valid PaaS design because the clients manage locally both the state and the execution of composite services.

In the rest of the section, we describe and compare the qualities only of the three valid PaaS designs shown in Figure 5.

The first alternative emphasizes the elasticity and dependability of the design. The PaaS runs the composite service on behalf of client C1 using an elastic engine that externalizes the state of the service composition to the data service offered by the same PaaS platform. The same pattern is adopted by the elastic task that externalize to the data service both its state and the resources its outputs. The bound composed service instead keeps the state locally to its own processing node. Clients take advantage of elastic URIs to access the state of the composite service, the state of its elastic tasks and their output. Their requests are transparently routed to the data service.

The second alternative emphasizes the performance of the processes by keeping their execution state as close as possible to the processing node that runs them, thus reducing the time to access the state and to update it. In particular, the PaaS runs the composite service using an elastic engine that 1) maintains the state of the composition locally and 2) also runs and keeps the process and its sub-process locally. The elastic task instead adopts a slightly different solution: it keeps the state of the execution locally, but externalizes its output to a

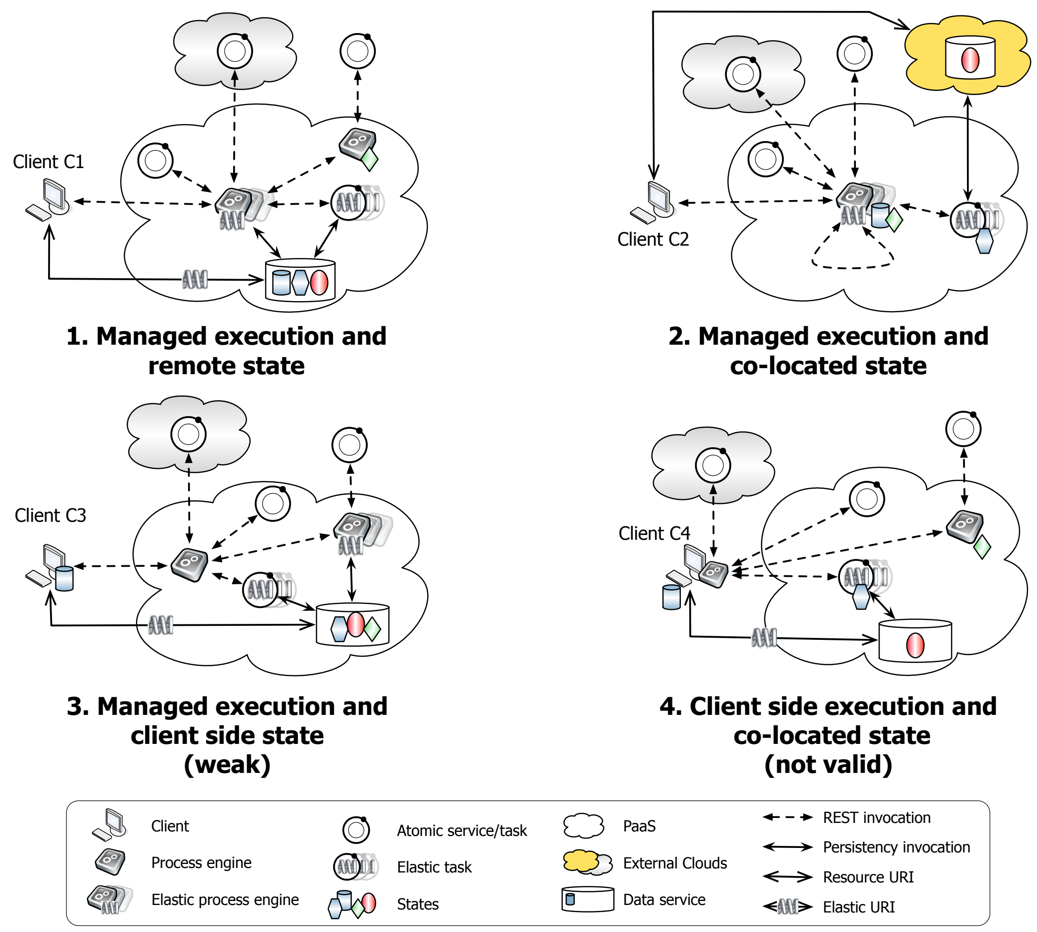

Fig. 5. Architectural alternatives for RESTful BPM in the Cloud: C1 managed process execution and remote state management (full), C2 managed process execution and co-located state management (full), C3 managed process execution and client-side state (weak), C4 local process execution and co-located state management (not a valid PaaS design).

data service provided by a different Cloud. In this case the client C1 interacts with the PaaS cloud to access the state of the composite service, its sub-process and its elastic tasks. However, it will be redirected to the external cloud provider when it attempts to read their output. In this way, the PaaS is offloaded from the management of service results, which is advantageous in cases where the composite services generate big amount of data, and, in turn, can use additional resources into the execution of process instances.

The third alternative emphasizes composite service adaptability by keeping the state of the composed service in control of clients. In this case, client C3 uses the PaaS to execute the process instance and can decide, between different interactions with the PaaS, to customize the service composition by partially changing its state. States of tasks, sub-processes, and the results are managed by the Cloud, and the client can access them via elastic URIs as in the previous architectures. The client however takes the full responsibility over the state of its process instance, which – e.g., for privacy concerns – cannot be deferred to Cloud data services as in the previous architectures.

In the following, the three architectures are compared in terms of their non functional quality attributes. The first architecture provides elastic scalability for the composite service deployed by the client and for its elastic task. Since their states are managed explicitly by the PaaS platform storage abstraction, the processing nodes remain completely stateless, and this makes easy to add and remove processing nodes, i.e., virtual machines on which the process engine is running. In particular, no state migration between processing nodes is needed. Furthermore, the data service of the PaaS is designed to handle big amounts of parallel requests, as long as they access independent objects and do simple put and get. And this is in line with what happens in the system: Process executions of different service instances work on independent set of resources and access their state in parallel. This architecture provides also dependability because by delegating the persistent storage of the execution state, a failure of a processing node does not have a big impact on the stored states. If the processing node fails the Cloud can allocate a new processing node and recover all the processes that were running on it from their currently stored states. However, there is a cost in terms of access time and latency: In general both are longer than accessing the state information locally. For this reason, the composite service that is invoked as a sub-process by the composition may provide better performance because its state is managed locally on the processing node. In general the state co-location is less flexible and dependable because the service cannot exploit a dynamically variable set of resources without migrating its state between different processing nodes, and because in case of failure of the processing node all the states of the sub-processes running on it may be lost. The first architecture also provides elastic access to the processing outputs. Depending on the user requirements, output data can be moved towards the client for faster access, and can be replicated for fault tolerance. Persistence of data is also guaranteed, therefore all these resources can be found in the Cloud also after the service instances terminate, but before a DELETE request is issued on them.

Compared to the first architecture, the second one may provide increased performance because all the states are managed locally with the processing nodes that access and update them. Moreover both process and sub-process run within the same elastic processors that improve performance and scalability of the composed services, as this may further reduce communication latency between running instances. This architecture however is less elastic than the former one because the state of the services must be explicitly migrated between computing nodes to consolidate the load and free up the resources of some of the processing nodes. Depending on the semantic of the composition and the size of the service state, live-migration of services may be time and resource consuming. This architecture is less dependable as failures to processing nodes, for services and tasks, may result in the corruption of their states and failure of the composition. However, the operational cost of consuming a dedicated data service of the PaaS platform is removed. By using an external provider for persisting the outputs of the process and having the client directly accessing it, the PaaS is offloaded as the burden of delivering the data to the client is shifted entirely on the external provider. In this case, the PaaS cannot provide strong guarantees about the delivery of the data, and cannot provide elastic URIs, making the infrastructure more rigid. Nevertheless, the resources originally devoted to deliver results to clients can now be allocated to run processing nodes if the system needs additional process execution capacity.

Compared to the others, the third architecture has increased flexibility because the client can dynamically change the composed service by manipulating its state. Its performance and dependability are similar to the first architecture when considering the execution of the elastic tasks and sub-processes because their state is fully managed by the Cloud. However, this architecture may show degraded performance and dependability at the level of main process. Performance degradation happens because making progress with the service composition involves the interaction between the Cloud and the client, since part of the state of the process needs to be shipped back and forth at every step of the execution. Since the client is driving the execution of the process, it is also responsible for making branching and – in general – adaptation and binding decisions that make it challenging for the Cloud to perform optimizations based on assumptions about the structure of the process. Finally, the dependability of the composed service depends on the client's ability to manage the state reliably and consistently during the process life-span. The process may fail, or even worse, its state may simply be lost if the client crashes and lacks the ability to recover the process state. Thus, for a dependable execution the client also needs to implement all state management activities, such as persistence, backup, recovery, checkpointing as well as state history logging and versioning that are usually provided within the PaaS.

These architectures differ also in how they deal with the long-livedness of service compositions. The first architecture can easily deal with long running processes provided by the user, as processing nodes are completely stateless. Since the state is maintained by the Cloud data service, processing nodes need only to be kept alive while processing requests and can be deallocated when idle, which for long running processes is likely to be most of the time.

The second architecture is the least flexible, and long running processes may make it impossible to deallocate any processing node if state migration is not available. This happens because the whole state of processes is co-located with the processing nodes executing them, therefore, processing nodes can only be stopped after the instances running on them have completed running or they have been pre-empted and migrated on a different processing node.

In the third architecture, the state of the composition is managed entirely by the client, hence the problem of long-lived service composition is moved outside the Cloud; this means that whenever the process completes its ongoing interactions, the processing node that are running the composite service can be deallocated, which is similar to what happens in the first architecture. Subsequent requests from the client to continue with the process execution will be routed to available processing nodes or will result in the allocation of additional processing nodes depending on the current workload.

It must be noted that the architectures presented here are some of the possible combinations to combine RESTful BPM and Clouds, and should be considered only as examples. In fact, by leveraging state migration, elastic URIs and elastic process engine nodes, one can dynamically reconfigure the infrastructure based on runtime monitoring: For example, one can keep the state of a service in the processing elements when several changes of the state must be committed, or while the service is active, but then can move the state to the data service when state updates become less frequent to trade performance against dependability. Likewise, the PaaS may create checkpoints of the locally managed state, either at regular time intervals or upon significant state changes, and then use these to recover from failures of the processing nodes.

VI. Related Work

This work investigate the combination of Cloud and composite services under the perspective of REST architectural style. In particular, we argue that RESTful BPM is a sensible choice to fully exploit Clouds as the backbone infrastructure to run and manage composite services that are truly elastic.

Related work in this area clusters around the concepts of Business Process as a Service (BPaaS), Workflow as a Service (WFaaS), and Composition as a Service (CaaS).

BPaaS and WFaaS mainly target traditional BPEL-based service compositions and tackle the problems to port them to the Cloud. This is usually achieved by extending some state-of-the-art engine to run over a cluster of virtualized serves. The preliminary work by Anstett et al. [2] studies the feasibility of running BPEL compositions under the different delivery paradigms that are enabled by Clouds. The focus lies on the security implications and in defining suitable trust requirements for multi-tenant BPEL engines. Pathirage et al. [19] present the design of a multi-tenant runtime system that runs BPEL service compositions in isolation, as clients use their own private instance of the composition execution engine. Damasceno et al. [6] propose the SSC4Cloud framework that provides a shared environment for designing service compositions, and executing them on the Cloud. Composite services modeled with a Cloud-based tool are then automatically translated to WS-BPEL and deployed to virtual computing nodes for the execution. Isolation is achieved by allocating all the processes of a client to one or more replicas of a private virtual machine. In our approach, multi-tenancy is achieved by means of public URIs and isolated service states: each composition is tied to unique and distinct resources and its state is evolved by one processing node at the time. All the executions are independent and the execution of a service acts on a separated set of resources, e.g. tasks, subprocesses, that have again unique URIs inside the PaaS. The hierarchical nature of URIs also helps to provide scoping for identifiers so that users can only instantiate and interact with processes, tasks and services within their scope. Isolation is provided at process level: each processing node is able to maintain separate executions, i.e., processes, for different services. As pointed out by Anstett et al., this may be detrimental to *performance isolation* because several processes share the same processing node. This can be mitigated by means of elastic processing nodes that align their capacity with the current load.

A different approach is proposed by Dörnemann et al. [7] with the *cloud-bursting* of BPEL tasks on the Amazon Cloud depending on the current load of the infrastructure. Instead of replicating the business engine across several virtual machines, the authors extend the BPEL engine architecture with a provisioning module to allocate VMs on-the-fly, a flexible request dispatching module, a load-balancer and a load analyzer, and define policies that dynamically scale the system and redistributed the load. In this work, the authors discuss the difficulties to enable elasticity for BPEL based service compositions that suggests – as we propose in this paper – the use of more flexible and light-weight architectures for composite services and business process management such as the ones based on REST.

In the same domain, Görlach and Leymann [13] discuss the potential of dynamic service provisioning for the Cloud. They propose models to support a flexible runtime management of the composite services at PaaS level for the replication and placement of composite services, and for the distribution of service tasks based on the structure of the service compositions. Similarly to our approach, scalability and elasticity are considered as the primary quality attributed to be delivered. However, their approach focuses more on the design of runtime management activities within the Cloud infrastructure, while we have directed our attention to analyze the architectural alternatives that can enable elastic and scalable execution of business processes in the Cloud.

Amazon recently launched the Amazon Simple Workflow service (SWF)[4], a PaaS offering that fits within the conceptual framework we have outlined. Even if details on its design are not public, from the available documentation we can argue that they make use of the processing nodes that store the execution state locally, while store the output results in the data service. They can send for execution tasks on external processing nodes, but provide also elastic processing nodes for implementing elastic tasks. In summary, their architecture may resemble the second alternative presented in the previous section, with the exception that the data service used is inside the PaaS instead of outside. The main limitation is that SWF does not take care of elastically running the workflow. Instead, users of the SWF need to explicitly define the rules to scale and distribute their composite service and tasks that the platform executes at runtime.

Work on CaaS focus on the SaaS offering and proposes rapid modeling tools delivered as a service for the design, deployment and execution of composite services. Thanks to the Cloud-based approach, process fragments and service composition knowledge can be easily shared and reused [26], [4]. CaaS tools offer advanced capabilities for the discovery of services that may fit within an incomplete composition, or have overlapping service specifications. The CaaS provides recommendations to users about their compositions that might include a completed workflow, process fragments, or modifications to the current workflow. Given the higher level of abstraction, also such CaaS tools may benefit from the PaaS architecture for RESTful BPM we have presented in this paper.

VII. Conclusions

Delivering Business Process as a Service (BPaaS) requires to design a specialized form of Platform as a Service (PaaS) which is optimized for elastic and dependable service compositions. In this paper we have discussed the most critical architectural alternatives concerning the management of the

[4]http://aws.amazon.com/swf/

state of the execution and the invocation of the tasks, subprocesses and services within the process. We have shown that the abstractions and concepts of RESTful business process management give a clear design strategy to manage the state of processes, tasks and services deployed in the Cloud and published to clients as resources, each with its unique identifier and each responding to the same uniform interface. These properties are also beneficial within the PaaS infrastructure to support the elastic execution of the processes which requires to efficiently replicate, migrate and consolidate their state. As part of the tradeoff between performance against dependability we have discussed and compared two alternative architectures, one which makes usage of the internal data storage services provided by the same PaaS platform, and another which relies on an external data storage service only used to publish the final results of the process. These are just the extreme cases of a continuum, whose sweet spot can also vary depending on the operational costs and the business model chosen by the particular BPaaS platform provider.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss National Science Foundation under the "Fellowship for Prospective Researches" contract PBTIP2-142337. We would like also to thank Prof. Schahram Dustdar and the Distributed System Group of the Technical University of Vienna for hosting Alessio Gambi and for providing great support for this research.

REFERENCES

[1] R. Alarcón, E. Wilde, and J. Bellido. Hypermedia-driven RESTful service composition. In *Proceedings of the International Conference on Service-Oriented Computing - Workshops*, ICSOC'10 Workshops, pages 111–120, 2010.

[2] T. Anstett, F. Leymann, R. Mietzner, and S. Strauch. Towards BPEL in the Cloud: Exploiting different delivery models for the execution of business processes. In *Proceedings of the Congress on Services*, SERVICES'09, pages 670–677, 2009.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkley view of Cloud Computing. Technical Report No. UCB/EECS-2009-28, University of California at Berkley, 2009.

[4] M. B. Blake, W. Tan, and F. Rosenberg. Composition as a Service. *IEEE Internet Computing*, 14(1):78–82, 2010.

[5] J. Brzeziński, A. Danilecki, J. Flotyński, A. Kobusińska, and A. Stroiński. ROsWeL workflow language: A declarative, resource-oriented approach. *New Generation Computing*, 30:141–164, 2012.

[6] J. C. Damasceno, F. A. A. Lins, R. W. A. Medeiros, B. L. B. Silva, A. R. R. Souza, D. Aragão, P. R. M. Maciel, N. S. Rosa, B. Stephenson, and J. Li. Modeling and executing business processes with annotated security requirements in the cloud. In *Proceedings of the International Conference on Web Services*, ICWS'11, pages 137–144, 2011.

[7] T. Dörnemann, E. Juhnke, and B. Freisleben. On-demand resource provisioning for BPEL workflows using Amazon's Elastic Compute Cloud. In *Proceedings of the International Symposium on Cluster Computing and the Grid*, CCGRID'09, pages 140–147, 2009.

[8] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5):66–71, Sept 2011.

[9] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. The Prentice Hall service technology series. Pearson Education, 2013.

[10] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, and D. Schumm. Capturing Cloud computing knowledge and experience in patterns. In *Proceedings of the International Conference on Cloud Computing*, CLOUD'12, pages 726–733, 2012.

[11] C. Fehling, F. Leymann, J. Rütschlin, and D. Schumm. Pattern-based development and management of Cloud applications. *Future Internet*, 4(1):110–141, 2012.

[12] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[13] K. Görlach and F. Leymann. Dynamic service provisioning for the Cloud. In *Proceedings of the International Conference on Services Computing*, SCC'12, pages 555–561, 2012.

[14] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the Symposium on Cloud Computing*, ACM SOCC'11, page 18, 2011.

[15] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for Cloud platforms. In *Proceedings of the International Conference on Performance Engineering*, ICPE'12, pages 85–96, 2012.

[16] S. Kumaran, R. Liu, P. Dhoolia, T. Heath, P. Nandi, and F. Pinel. A RESTful architecture for service-oriented business process execution. In *Proceedings of the International Conference on e-Business Engineering*, pages 197–204. ICEBE'08, 2008.

[17] OASIS. Web Service Business Process Execution Language Version 2.0 Specification, april 2007. OASIS standard.

[18] G. Pardon and C. Pautasso. Towards distributed atomic transactions over RESTful services. In E. Wilde and C. Pautasso, editors, *REST: From Research to Practice*, pages 507–524. Springer, 2011.

[19] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana. A multi-tenant architecture for business process executions. In *Proceedings of the International Conference on Web Services*, ICWS'11, pages 121–128, 2011.

[20] C. Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[21] C. Pautasso. BPMN for REST. In *Proceedings of the International Workshop on Business Process Model and Notation*, BPMN'11, pages 74–87, 2011.

[22] C. Pautasso and G. Alonso. The JOpera visual composition language. *Journal of Visual Languages and Computing*, 16(1–2):119–152, 2005.

[23] C. Pautasso and E. Wilde. Push-enabling RESTful business processes. In *Proceedings of the International Conference on Service-Oriented Computing*, ICSOC'11, pages 32–46, 2011.

[24] M. Pereira, M. Fernandes, and J. Martins. Web service and business process execution on peer-to-peer environments. In *Proceedings of The International Conference on Advances in P2P Systems*, AP2PS'11, pages 19–26, 2011.

[25] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly Media, Incorporated, 2007.

[26] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards Composition as a Service - a quality of service driven approach. In *Proceedings of the International Conference on Data Engineering*, ICDE'09, pages 1733–1740, 2009.

[27] N. Stojnić and H. Schuldt. OSIRIS-SR: A Safety Ring for Self-Healing Distributed Composite Service Execution. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'12, 2012.

[28] J. Varia. Cloud architectures. White paper, Amazon, 2008.

[29] J. Varia. Best practices in architecting Cloud applications in the aws cloud. In R. Buyya, J. Broberg, and A. Goscinski, editors, *Cloud Computing: Principles and Paradigms*, pages 459–490. Wiley Press, 2011.

[30] M. Wang, K. Y. Bandara, and C. Pahl. Process as a Service. In *Proceedings of the International Conference on Services Computing*, SCC'10, 2010.

[31] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010.

[32] X. Xu, I. Weber, L. Zhu, J. Liu, P. Rimba, and Q. Lu. BPMashup: Dynamic execution of RESTful processes. In *Proceedings of the International Conference on Service Oriented Computing*, ICSOC'12-DEMO, 2012.

[33] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing web services choreography standards: the case of rest vs. soap. *Decis. Support Syst.*, 40(1):9–29, 2005.