Publishing Persistent Grid Computations as WS Resources

Thomas Heinis, Cesare Pautasso, Oliver Deak, Gustavo Alonso
Department of Computer Science, Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{heinist,pautasso}@inf.ethz.ch, odeak@student.ethz.ch, alonso@inf.ethz.ch

Abstract *Grid services can be composed into processes, providing a high level definition of the computations involved in terms of their data exchanges and control flow dependencies. In this paper we show how processes themselves can be efficiently published as Grid services by mapping the persistent state of the process executions to standard compliant interfaces as defined by the Web Services Resource Framework (WS-RF). Mapping processes to resources is a fundamental step to enable recursive Grid service composition, where composite Grid services are themselves published as services. This gives processes a standardized and well-understood interface that enables their management, monitoring, steering and adaptation. Additionally it eases their reusability and simplifies integration into existing Grid applications and portals. In order to determine the mapping's overhead, we include the results of a comprehensive performance evaluation.*

1. Introduction

The idea of process-based Web service composition and more recently process-based Grid service composition has gained widespread acceptance (e.g., [4, 17, 15]). Using processes, already deployed Web (and Grid) services can be reused to build complex service compositions at a high level of abstraction. However, one important open issue that still needs more attention concerns the reusability of such compositions.

Processes can be used to describe ad-hoc Grid computations involving a well-defined set of Grid services that are integrated to perform a certain computation once. Processes can then be generalized by making them parametric, in order for the same Grid service composition to be reused with different input data sets. Furthermore, the same process can be bound to different Grid service providers, allowing for the same algorithm structure (i.e., the process) to be reused with different building blocks (i.e., the services). Another interesting option – the focus of this paper – consists of reusing a Grid service composition by publishing it as another Grid service.

In this paper we propose to map a process composing Grid services to a Grid service interface compliant with the Web Services Resource Framework and Web Services Notification set of specifications (WS-RF and WS-N [6]). A process published as a Grid service can be accessed as a resource through a Web service interface, the WS-Resource, which provides standardized operations. The interface includes support for advanced functionality not found in stateless Web services, such as lifecycle management, property manipulation and event notification.

One important contribution of this paper lies in the definition of a precise mapping between these WS-RF concepts and the persistent state of a service composition. Furthermore, our approach is general and can be used with process-based compositions defined using different modeling languages. Exposing process-based composition of Grid services as a standard-compliant Grid service has several advantages. For example, in addition to basic Grid services, also composite Grid services can be invoked, managed, and monitored from existing Grid-enabled client applications and portals.

The mapping can be outlined as follows. Lifecycle management of the resource provides a simplified way to manage the persistent state of the corresponding process execution. When the lifetime of the associated resource expires, the state can be garbage collected by the underlying process management system. Resource property manipulations are directly mapped to the state of the corresponding process instance, so that clients can for instance access intermediate results during the execution of the Grid service composition and, to some extent, steer and control the computation by setting the values of the properties of the associated resource. Notifications can be used to inform clients about state changes, giving a powerful and efficient mechanism to report the progress of the execution of a composition. Finally, the WS-RF concept of service groups can be used to manage batches of related process instances.

While such an interface for persistent computations greatly simplifies its integration by providing enhanced managing as well as monitoring capabilities, the performance overhead introduced by such a mapping is critical if a potential solution is supposed to scale for a large number of clients and resources. Thus, we need efficient mechanisms

Part of this work is funded by the European IST-FP6-004559 project SODIUM (Service Oriented Development In a Unified framework).

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE

to create (i.e., start computations) and destroy resources, to read and write the state of the resource as well as to manage notifications and subscriptions, i.e., to match events with subscriptions and to send notifications. We have implemented such a WS-RF mapping layer on top of the JOpera autonomic process execution engine [23] and present an extensive performance evaluation of the system scalability and overhead.

The remainder of this paper is structured as follows: in Section 2 we set the context for our work by giving an overview over process-based Grid service composition. Section 3 discusses the mapping between processes and WS-Resources leading to a discussion of its applications in Section 4. An outline of the implementation is given in Section 5 followed by a performance evaluation of the current prototype implementation (Section 6). We conclude the paper by discussing related work in Section 7 and drawing some conclusions in Section 8.

2. Modeling Persistent Grid Computations with Processes

In this section we state our assumptions about how processes are used to model computations composed of Grid services. We also define the persistent state associated with such computations as well as their lifecycle, so that it becomes clear what information should be provided by a process management infrastructure in order for our approach to be applicable.

The concept of process is shared among several Grid service composition languages and tools like GSFL [15], Pegasus [7], AGWL [8], Triana [25], ScyFlow [20], GridFlow [3], JOpera [22], xWFE [28] and Karajan [27].

A process is a composition of tasks connected by data flow and control flow. Processes also have input parameters, holding the data passed to the process, and output parameters storing the results of the computation. Tasks (equivalent to jobs in ScyFlow and activities in BPEL, AGWL and GSFL) refer to Grid services which also have input and output parameters associated with them. The data passed to a service is copied into the input parameters whereas the results coming from a service invocation are copied into the output parameters of the task. The data flow defines how the results of a Grid service invocation are copied into the input parameters of the next tasks. The control flow defines the order of invocation of the distinct Grid service invocations. Neither Karajan or xWFE explicitly distinguish between control flow and data flow, but instead derive the control flow from the data flow dependencies. This, however, does not hinder the applicability of our approach. In fact, when we define the mapping between a process and a resource we are mostly interested in its runtime execution state.

The runtime state of a process can be defined to be all data associated with its execution. Since the same process can be executed multiple times, such state is typically structured in several instances that can be active concurrently. Each instance stores a set of state attributes comprising the values of the input and output parameters (referred to as input/output data set in GridFlow, variables in BPEL, data packages in AGWL, parameters in xWFL and input/output files in Karajan) of all its Grid service invocations including the process itself, as well as process and task attributes written by the execution engine. These attributes include meta-data such as execution times of tasks and processes determined by the execution engine, the current execution status of tasks and processes (e.g., waiting, running, finished, failed, etc.) and other execution related information which may differ depending on the engine.

The lifecycle of a process instance begins when a process is instantiated. During the execution of the instance, its state will be accumulated and stored persistently. This implies that all intermediate results, i.e., results of task invocations, remain available after the task has finished and the following tasks have consumed them. The final results will also be saved once the computation is finished. At the end of the lifecycle of a process instance, its state will be removed from the persistent storage. It is worth noting that the end of the lifecycle of the instance and the end of the computation do not necessarily coincide: after the instance has finished executing, the state will still be available, allowing the history of the computation to be read from persistent storage.

3. Mapping Processes to WS-Resources

The specifications that constitute the Web Service Resource Framework have been defined in order to shift from the stateless paradigm of plain Web services to the stateful model of Grid services [5].

To do so, WS-RF [12] loosely couples a Web service with a stateful resource and provides well-defined methods to monitor and manage its state. In this context, a Web service that provides a standardized set of operations to access the state of the resource associated with it is called a WS-Resource. The areas of standardized operations span lifecycle management, property manipulation and service groups as specified in WS-ResourceLifetime [14], WS-ResourceProperties [14] and WS-ServiceGroup [19] respectively. In order to also provide publish-subscribe interaction patterns for Web services, the WS-N set of specifications (WS-BaseNotification [13] and WS-Topics [26]) has been brought forward.

Together, the WS-N and WS-RF set of specifications define the Grid service interface [5]. In the following we give a brief outline of the specifications and also provide a detailed mapping of the specified operations to the persistent state of a Grid computation defined as a process.

3.1. WS-Resource

This specification defines the implied resource pattern [6] as being the relationship between a Web service, the WS-resource, and a resource. A WS-Resource is defined to be a Web service through which clients can access the state of a resource and manage its lifetime. The WS-Resource uses implementation specific means to access the underlying resource. The specification is not very restrictive with respect to what can be considered a resource. The only requirements a resource has to satisfy is that it needs to be uniquely identifiable and that it has properties. Process instances meet these requirements as they are usually associated with unique identifiers and contain state information which can be interpreted as properties.

Thus, we propose to map a process instance storing the state of the execution of a composite Grid service to a resource. As we are going to show, clients can manage a process instance through the standard operations provided by a WS-Resource interface. Because the set of state attributes (i.e., the properties) is identical for all instances of a given process, only one WS-Resource interface per process is required to operate on all instances of the particular process. However, for each process that is published as a Grid service, an additional WS-Resource interface is required.

Furthermore, since the mapping between resources and process instances is a bijection and each process instance already has its own unique identifier, we can reuse the same identifier for the corresponding resource. Thus, the process identifier becomes part of the resource endpoint reference and will have to be included in all messages sent to the WS-Resource in order to correlate the client request with the individual process instance.

3.2. WS-ResourceLifetime

The WS-ResourceLifetime specification defines the management of a resource by providing means to either destroy a resource instance immediately via the **Destroy** operation or to schedule its destruction at a specific point in time by using the **SetTerminationTime** operation. The scheduled destruction time is a property of the WS-Resource and can therefore be queried, set and thereby extended accordingly.

Both operations defined in this specification, immediate and scheduled destruction, are mapped to the lifecycle of the process instance and its state. In addition to discarding the state of a process instance, destruction will also interrupt and terminate the ongoing execution (if the process is still running).

Since the specification does not include a standard mechanism for resource creation, we discuss several alternatives for instantiating a new process upon resource creation. Related to this, we also describe two additional operations to control the state of the process associated with a resource, once it has been created.

The first way to create a resource, is through the **startProcess** operation. This operation instantiates a process and begins its execution after having allocated a new resource for it. In some cases, e.g., for parameter sweep computations [1], the same process is started multiple times with different input parameter values. Calling the **startProcess** operation several times to do so may be expensive. Thus, we also provide the **startBatch** operation which, instead of starting only one process, starts a batch of identical processes that may receive different input data.

Additionally, in order to enable more fine-grained lifecycle management of the process execution we provide the (non-standard) **Suspend** and **Resume** operations which allow the client to pause the execution of a process and to subsequently resume it. The suspend operation amounts to setting a breakpoint before the next task which is to be executed. Execution will be suspended once the breakpoint is reached. These two operations can be used in conjunction with the ability of modifying property values of the associated resource, so that the changes can be applied by assuring the client that the state of the suspended process has not changed and thus ensuring the consistency of the result.

Finally, the **startSuspended** operation is an atomic combination of the **startProcess** and **Suspend** operation which prepares a new process instance for execution but does not start it. This operation can for instance be useful to start a process, subscribe to its topics and only then resume execution, thereby making sure none of the notifications sent due to the subscriptions is missed.

3.3. WS-ResourceProperties

This specification defines how the properties associated with the state of a resource can be accessed using a pull mechanism and how they can be modified. Published properties of a resource are defined in a document associated with the resource. A client can retrieve the properties document from a resource via the **GetResourcePropertyDocument** operation or can query the resource for specific properties by invoking the **QueryResourceProperties** operation and can read or write the values of these properties using the **GetResourceProperty** and **SetResourceProperty** operations respectively.

In the case of resources representing processes, their properties can be directly mapped to the process execution state. Since the persistent state of a process can be modeled as a set of attributes as described in Section 2, each of these can be accessed by clients through the corresponding property of the resource. Thus, each input and output parameter of the process (and its tasks) as well as execution related meta-data (e.g., profiling, debugging, error handling information that is accumulated during the execution of the process) is mapped to a specific property. Given the data flow structure of a process, it is possible to automatically generate the corresponding resource's property document by enumerating its state attributes.

In particular, we distinguish between read-only and read-write properties. There is no need to be able to write properties mapped to intermediate results of tasks as well as to final results of the process (i.e., mapped to output parameters). Also, writing properties that are mapped to instance attributes set by the execution engine should not be allowed. In fact, doing so would invalidate the state of the process. Additionally, there are also constraints regarding the time when such properties can be read. That is, they can only be read after having been initialized by the execution engine. Violating these rules leads to a fault message sent to the client.

In contrast to the read-only properties there is also a set of read-write properties. Input parameters to tasks and also the process are defined to be read-write properties. However, also in this case there is no need to be able to write to properties that map to input parameters that have already been used for the computation since it will not have any influence on the process execution. Writing such a property will therefore also lead to a fault message.

3.4. WS-ServiceGroup

With this specification, groups are used as a classification mechanism to simplify the discovery and management of sets of WS-Resources. WS-Resources are not allowed to freely join groups, but must meet certain criteria defined for each group. Groups can then in turn be queried to find all members.

We map the concepts defined in the WS-ServiceGroup specification to the execution of batches of related processes. To do so, service groups are defined so that membership is restricted to only allow resources representing process instances belonging to the same execution batch to join the group.

3.5. WS-BaseNotification / WS-Topics

The WS-BaseNotification and WS-Topics specifications define the push mechanism used by clients to be informed about changes occurring at the resource. With these, clients are notified using an asynchronous event-notification interaction pattern. A client can subscribe to topics defined by a WS-Resource using the **Subscribe** operation and will receive the corresponding notifications from it. The various topics provided by a resource are defined as one of its resource properties and can thus be queried for.

Similar to the resource properties, we define the attributes of the state of a process instance to be available as topics. This means that whenever a state attribute changes, subscribed clients will receive a notification. The notification sent to the clients also includes the new value of the attribute, in order to reduce the load on the WS-Resource server.

As an extension, we also include the **startSubscribed** operation, which atomically instantiates a resource and subscribes to its changes (the topic is passed as a parameter). This way, clients are guaranteed not to loose notification messages.

4. Applications

In this section we discuss how the previously described mapping improves the handling of persistent computations in the areas of lifecycle management, monitoring, and steering of one or a batch of processes.

4.1. Lifecycle Management

Mapping WS-Lifetime to the lifecycle of processes and their state provides a useful and elegant technique to deal with the problem of managing the accumulated state of past executions of a process. This allows clients to define during what time frame they are interested in the computation's results to remain stored persistently. If the computation terminates within this window, its results will be kept as part of its persistent state as long as the resource instance is not destroyed. Otherwise, the computation will be aborted upon the expiration of the corresponding resource's lifetime and its state will be discarded at a well-defined and predictable time. This is an improvement with respect to other systems that resort to ad-hoc techniques to manage the results of past computations (e.g., [16]).

4.2. Monitoring and Steering

Using property manipulation a client is able to read and write properties that map to attributes of the underlying state. More specifically, by reading properties a client can poll for the current state. Thus, a resource can be queried to find out what task has been reached by the process, to retrieve intermediate results or to download the final results once the process execution has completed. However, if a client requires to be informed about a change of a state attribute mapped to a property, with this property-based mechanism it still needs to periodically poll the resource to find out about the new value. An alternative method to monitor the process and its execution is thus provided by the topics to which the client can subscribe and about which it will receive a notification once the value of the corresponding property changes. This subscribe/notify form of interaction enables a push model relieving the client from having to poll for the changed value and the service provider from the overhead introduced by polling clients. A client can subscribe to execution state changes of individual tasks of the process and is thereby able to track the progress of the process execution.

Steering of processes can be achieved by proactively setting properties to different values. With this, for instance, a client becomes able to reset the value of an input parameter of a task depending on the result or the value of an

output parameter of a different task. Thus, it can steer the execution by following different paths in the control flow and adapting the data flow of the running process. Since partial results are sent to the client through notifications, a client can use this information to perform a form of error recovery: if results indicate that an exception or failure has occurred, the client can take corrective actions and reset input parameters of tasks yet to be executed. To do so in a safe way in order to avoid inconsistencies, the client should suspend the execution of the process by interactively setting a breakpoint on a specific task or by pausing the execution of the entire process immediately. The client is informed with a notification when the breakpoint is reached. After the values of the selected properties have been corrected, the client can resume execution (**Resume**).

4.3. Managing process batches

We use the concept of service groups to manage process batches. With this, a client can add all started processes belonging to a batch to a service group. Clients can then read the properties of the service group to find all process instances of the batch. Using the concept of the service group in this context therefore provides a convenient way of grouping process instances so that they can be managed and monitored as a whole. For example, processes can be grouped by a caller-callee relationship, so that the state of a set of nested processes is treated as a single resource. This in turn can be used to keep track of the execution of the entire set of processes and to garbage collect multiple (but related) process instances when the corresponding resource is destroyed.

5. Implementation

In order for our approach to be implemented, an execution engine needs to store the state of the processes persistently and provide an API with the following functionality: instantiate, start, suspend, resume and kill processes as well as a mechanism to read and write their persistent state and listen for notifications to its changes.

In this section we describe how we implemented the mapping of processes to resources in the context of JOpera for Eclipse [21]. This tool can be used to graphically define Grid workflows and it includes an execution engine that satisfies the aforementioned requirements. We first give an overview over the execution environment's architecture and will then describe the implementation of the WS-RF and WS-N interfaces in more detail.

5.1. Architecture

The architecture of the JOpera execution environment can be described as a set of layers: the grid clients, the grid service container, the execution engine (logically including the persistent storage), its API and an open set of service invocation adapters (Figure 1).

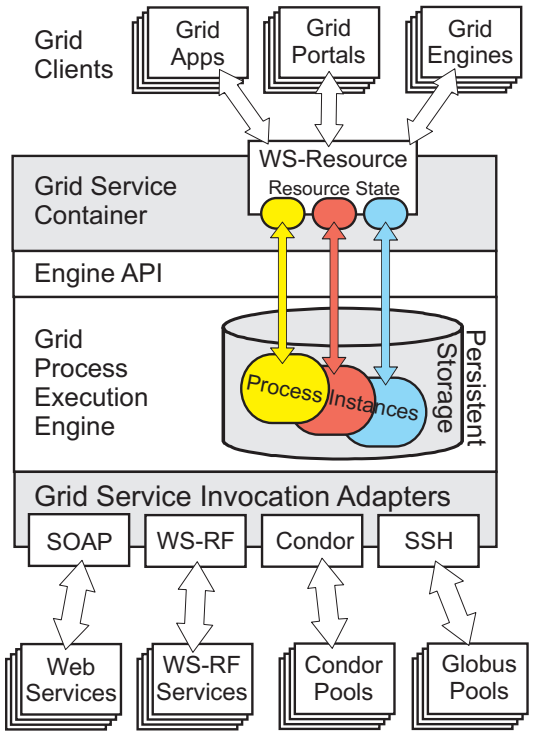

Figure 1. Layered architecture of the JOpera Grid service composition engine

Thanks to its standards compliant WS-RF interface, various *Grid Clients* can connect to the system. These are, e.g., Grid-enabled applications offloading some computationally intensive process to the process execution engine; Grid portals providing users with an interface to monitor the progress of the processes; other Grid service composition engines, thereby enabling recursive Grid service composition.

The *Grid service container* layer implements the mapping between the WS-Resource abstraction and the process abstraction. By correlating the specific resource endpoint reference passed by clients, it is able to address each process instance and interact with it using the engine's API.

The *Execution Engine API* provides an interface for clients to manage and run processes in the JOpera execution engine. With it, the Grid service container can instantiate, run and stop processes and can also access and manipulate their state.

The *Execution Engine* is mainly concerned with the execution of the processes. It does so by invoking each Grid service in the order defined by the control flow and by copying data between these invocations as specified in the data flow. Intermediate results as well as execution related information is stored in the *Persistent State Storage*. The persistent state storage is implemented using a database.

The *Grid Service Invocation Adapters* are used by the execution engine in order to call Grid services using the appropriate mechanism and protocol. Adapters for Grid services as well as different kinds of services can be plugged into the JOpera execution engine. Using such adapters, JOpera is currently able to invoke WS-RF compliant services (e.g., Globus 4.0 services), plain Web services, and to efficiently submit jobs to Condor [18] pools as well as jobs in a Globus [9] command-line based environment accessed through the SSH protocol.

5.2. Implementing the WS-RF and WS-N Interfaces

As depicted in Figure 1, Grid Clients access processes using the WS-Resource hosted in the Grid service container. Once a process has been deployed in the execution engine, the corresponding WS-Resource will be deployed in the Grid service container above it and a description of it is made available as a WSDL. We have implemented the WS-Resource using the WS-RF, Pubscribe and Addressing libraries developed as part of the Apache Web Service project [2]. Although only one WS-Resource is deployed per process, it provides access to all the instances of this process by mapping different endpoint references to their corresponding instance.

Instance lifecycle management: Upon receiving a request to instantiate a process, the WS-Resource will trigger the corresponding instance creation in the underlying engine. The engine returns an identifier for the instance back to the WS-Resource, which then passes this to the client as an endpoint reference. Conversely, if a client requests immediate resource destruction, the WS-Resource will use the execution engine API to stop the execution and to discard the state of the corresponding process instance. The same mechanism is used for scheduled destruction, where the engine is notified by the container whenever the resource is about to be destroyed.

Property manipulation: The WS-Resource keeps a list of all properties that can be read and written by clients. The set of properties is the same for all instances of a process and is defined at deployment time of the process. In order to implement the operations to set and get properties, the WS-Resource keeps a copy of all relevant values for each process instance. This is especially efficient for reading operations. It however bears the overhead of always having to update the copy of the value once the value in the persistent state has changed. The operation used to set properties directly uses the engine API to update the corresponding state attribute values.

Notifications: As discussed earlier, the topics are defined to be the attributes of the persistent state. Using the *Subscribe* operation, the client is able to subscribe to specific topics and will receive notifications once the value of the subscribed topic has changed. In order to implement the property change notification, the service container registers listeners with the execution engine. These will receive notifications once a state change occurs. Upon receipt of such a notification, the new values are copied from the persistent state to the WS-Resource in order to notify subscribed clients. In order to keep track of the subscriptions, the WS-Resource maintains a table of which client is subscribed to what topic. Once it is informed about a change, it checks which client is subscribed for these events and sends out notifications.

6. Evaluation

In this section we evaluate the performance of the WS-RF enabled process execution engine. We focus on determining the cost of several resource creation alternatives and showing that the system can scale to handle a very large number of resources.

The experiments have been carried out by installing the JOpera Grid service composition engine on a server running Linux (RedHat AS 4), equipped with two AMD Opteron 2.4GHz CPUs and 2GB of memory. The clients were running on a cluster of 64 dual processor (1Ghz) nodes connected with a 100Mb/s local area network.

6.1. Resource Creation

In a first set of experiments, we measured the time required to instantiate a process and to subscribe to one of its properties in 4 different ways. We did the first using the **startProcess** (**S**) and the combination of **startSuspended** and **Resume** (**SR**). The latter was done with a combination

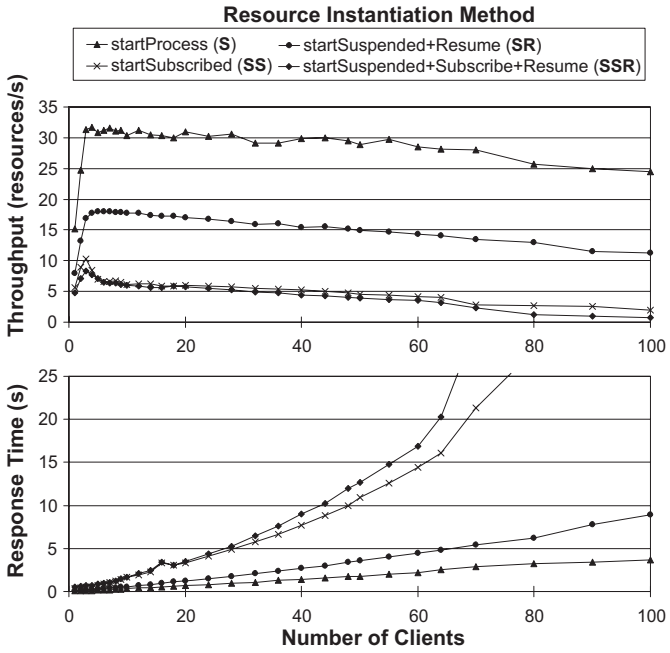

Figure 2. Throughput and response time of different resource instantiation methods

of the three operations **startSuspended**, **Subscribe** and **Resume** (**SSR**) and also with these three operations merged into one, the **startSubscribed** (**SS**) operation. We ran the experiments with up to 100 concurrent clients. The response time and the throughput are shown in Figure 2.

Response Time: Resource creation with **S** has a lower response time than **SR** because in the former case only one operation is invoked compared to two in the latter case. The response time for instantiation requests in both cases scales linearly with the number of clients. Also **SS** performs faster than **SSR**. This can again be attributed to the fact that the former is executing only one operation whereas the latter executes three. From the results it can also be observed that the operations involving a subscription (**SS** and **SSR**) are significantly slower than the others, even in the case of **SS**, where instantiation and subscription are done atomically. Thus, the time required to subscribe to a topic outweights the time required for message transfer.

Throughput: The throughput (Figure 2 bottom) for the different methods of resource creation gives a similar picture as the response time: **S** has the highest throughput, followed by **SR**. Again, the two methods that include a subscription to a topic, **SS** and **SSR**, have the lowest throughput. This shows that subscription is costly, as multiple concurrent clients must be synchronized to access the shared resources subscription table. For the operations involving a subscription, we have also measured the throughput of the notifications sent by the Grid service container to the clients, observing that at most 26 notifications/second could be sent for 6 clients, each creating and subscribing to 100 resources.

Process batch instantiation: In order to motivate the need for the **startBatch** operation, we have also compared the time it takes to instantiate process batches of different sizes by calling the **startProcess** operation repeatedly and the **startBatch** operation once. As can be seen in Figure 3, starting the process batch with the **startBatch** operation is drastically faster. With it, creating 10^5 resources takes less than a second, compared to almost 3 hours with the **startProcess** operation.

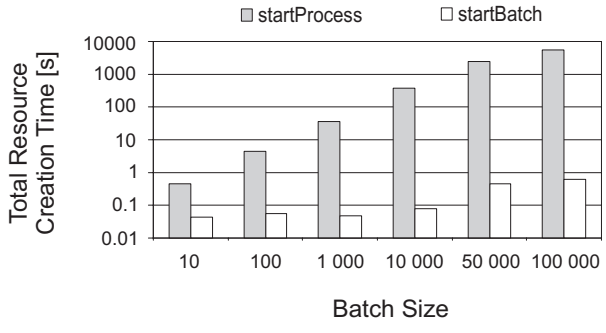

Figure 3. Comparison of different methods for starting process batches

6.2. Querying Properties Overhead

In this experiment we used one client to create a resource and get the value of one of its properties and measured the time it requires to execute the **getResourceProperty** operation. The results shown in Figure 4 indicate that the time required for the operation increases linearly with the number of resources instantiated starting at 40ms and growing to only 70ms when 100'000 resources have been created. This is because before the value of the property can be read and be sent back, also here the resource must be found first. The process of finding a resource of course takes longer with an increasing number of instantiated resources.

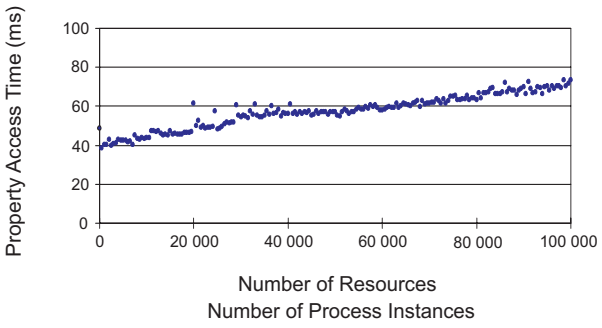

Figure 4. Property access overhead with an increasingly large number of resources

7. Related Work

Only few research results are available in the context of turning applications or service compositions into Grid services. Initial work [10] was concerned with providing distributed applications with a Web service interface. The interface was mainly used to authenticate and authorize users as well as instantiate applications. Similar work [11] was carried out in order to wrap command line-tools in Web services to make them available for remote users. The limitations of wrapping stateful resources with stateless Web services however have been well-understood in the meantime [5]. In [24] the authors describe a system which turns scientific legacy applications into Grid services and publishes these in a Grid portal. To do so a generic application independent WS-RF layer is added on top of the legacy application without having to generate additional code. This approach is similar to ours insofar as they turn applications into Grid services. Their implementation of the WS-RF layer does however not exploit the full potential of the WS-RF specifications by only implementing a small subset of the operations specified in WS-RF, thereby lacking the support for lifetime, property and notification management.

8. Conclusions

In this paper we presented our approach describing how to bridge the gap between two abstractions: resources and processes. We did so in order to enable recursive composition of Grid services, where the process defining how Grid services are composed, is published itself as a Grid service. This mapping has been described in terms of the concepts defined by the Web Service Resource Framework (WS-RF) and Notifications (WS-N) and is applicable to several Grid service composition languages and tools sharing the notion of a process. With it, Grid computations modeled with a process at design-time can be managed at run-time through a standardized interface provided by the corresponding resource. By reporting on the implementation of such a mapping, we have shown the feasibility of using a WS-Resource interface to initiate, monitor, steer, suspend, resume and delete the persistent execution state of a Grid computation by creating and destroying the associated resource and reading, writing and subscribing to its properties. Finally, the results of our experimental performance evaluation indicate that the overhead introduced by the mapping layer is minimal and that the system scales well to manage the lifecycle of hundreds of thousands of resources representing process instances in the underlying execution engine.

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: killer application for the global grid? . In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 520–528, Cancun, Mexico, 2000.

[2] Apache Software Foundation. Apache Web Services Project. `http://ws.apache.org`.

[3] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGRID '03: Proc. of the 3st International Symposium on Cluster Computing and the Grid*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[4] F. Casati and M.-C. Shan. Dynamic and Adaptive composition of e-services. *Information Systems*, 26:143–163, 2001.

[5] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From Open Grid Services Infrastructure to WSResource Framework: Refactoring & Evolution, 2002. `http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf`.

[6] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework, June 2005. `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`.

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow management in griphyn. pages 99–116, 2004.

[8] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proc. of Cluster Computing and Grid (CCGrid)*, Cardiff, UK, 2005.

[9] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[10] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Services and Application Factories*. Wiley, 2002.

[11] C. Goble, C. Wroe, and R. Stevens. Grid Project: Services, Architecture and Demonstrator, 2003. `http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/128.pdf`.

[12] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, and I. Sedukhin. Web Services Resource 1.2, June 2005. `http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-pr-01.pdf`.

[13] S. Graham and B. Murray. Web Services Base Notification 1.2, 2004. `http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf`.

[14] S. Graham and J. Treadwell. Web Services Resource Properties 1.2, June 2004. `http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-pr-01.pdf`.

[15] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A Workflow Framework for Grid Services, 2002. `http://www-unix.mcs.anl.gov/~laszewsk/bib/papers/vonLaszewski--gsfl-a.pdf`.

[16] C. Letondal. A Web interface generator for molecular biology programs in Unix. *Bioinformatics*, 17(1):73–82, 2001.

[17] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[18] M. Litzkow, M. Livney, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of the 8th International Conference on Distributed Computing Systems*, 1988.

[19] T. Maguire and D. Snelling. Web Services Service Group 1.2, June 2004. `http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-pr-01.pdf`.

[20] K. M. McCann, M. Yarrow, A. DeVivo, and P. Mehrotra. ScyFlow: An Environment for the Visual Specification and Execution of Scientific Workflows. In *Proc. of Workflow in Grid Systems at GGF10*, Berlin, Germany, 2004.

[21] C. Pautasso. JOpera: Process Support for more than Web services. `http://www.jopera.org`.

[22] C. Pautasso and G. Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing*, 16(1–2):119–152, 2004.

[23] C. Pautasso, T. Heinis, and G. Alonso. Autonomic Execution of Web Service Compositions. In *Proc. of the International Conference on Web services 2005*, Orlando, FL, USA, 2005.

[24] S. Sanjeepan, A. Matsunaga, L. Zhu, H. Lam, and J. A. B. Fortes. A Service-Oriented, Scalable Approach to Grid-Enabling of Legacy Scientific Applications. In *Proc. of the International Conference on Web services 2005*, Orlando, FL, USA, 2005.

[25] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[26] W. Vambenepe. Web Services Base Topics 1.2, June 2004. `http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf`.

[27] G. von Laszewski. Java CoG Kit Workflow Concepts for Scientific Experiments, 2005. `http://www-unix.mcs.anl.gov/~laszewsk/papers/vonLaszewski-workflow-taylor-anl.pdf`.

[28] J. Yu and R. Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, USA, 2004.