Mirroring resources or mapping requests: implementing WS-RF for Grid workflows

Thomas Heinis, Cesare Pautasso, Gustavo Alonso
Department of Computer Science
ETH Zurich
8092 Zürich, Switzerland
{heinist, pautasso, alonso}@inf.ethz.ch

***Abstract*— The Web Services Resource Framework (WS-RF) and the Web Services Notification (WS-N) specifications are a crucial component of Grid infrastructures. They provide a standardized interface to stateful services so that they can be managed remotely. There are already several implementations of these specifications and initial performance studies have compared them in terms of the overhead observed by a single client. In this paper we address the problem of implementing the WS-RF and WS-N specifications for large scale systems. In particular, we discuss how to implement WS-RF and WS-N as the management interfaces to a Grid workflow engine. In the paper we describe and compare two different architectures for mapping resources to processes. The first one mirrors the state of the process as a resource. The second one maps the client requests to access the state of a resource embedded into the Grid workflow engine. We include an extensive performance evaluation, comparing the resulting systems in terms of scalability when servicing a large number of concurrent clients.**

I. Introduction

In order to provide Grid workflows with a stateful service-oriented interface, we have used the Web Services Resource Framework (WS-RF) [1] and the Web Services Notification (WS-N) [2] specifications to define the interface to a Grid workflow management system. Processes (or workflows) are used to model the interactions between computational and data services deployed on the Grid [3], [4]. Such interactions are captured in a workflow, which is then deployed for execution into a Grid workflow engine. Through WS-RF and WS-N the resulting workflows can themselves again be published as Grid services. This allows clients to easily integrate workflow-based Grid services into Grid applications and portals.

In [5] we have presented a mapping of a process model common to most Grid workflow languages onto the notion of resource used in the WS-RF and WS-N specifications. This mapping is however not particularly efficient and the resulting system does not scale well when facing large numbers of client requests.

In this paper we discuss an alternative implementation of the WS-RF and WS-N mapping for Grid workflow systems which provides improved scalability when servicing a large set of concurrent clients. To do so, we compare two different architectures. The first uses WS-Core (part of the Globus Toolkit [6]) to mirror the state of the execution of a process into a locally managed resource. That is, the workflow process is mapped to a resource. The second embeds the state of the resource into the Grid workflow engine by mapping the WS-RF requests to the engine's API. Although the first approach lets clients read resource property values faster, the performance of this mirrored architecture is limited due to the redundancies shared between the WS-RF layer and the underlying workflow engine. The second solution, mapping requests rather than mirroring resources, removes unnecessary layers of indirection and scales significantly better. To demonstrate this, we include an extensive performance comparison of the two solutions.

The remaining part of this paper is organized as follows. In Section II we give a brief overview over the relevant aspects of WS-RF and WS-N. In Section III we describe the Grid workflow management system we have used (JOpera [7]) and present the mapping from processes to WS-RF resources. In Section IV we discuss the first solution. Initial measurements will be used to motivate the need for a more lightweight implementation. We then outline the second solution (Section V) and compare the two implementations in Section VI. In Section VII we present related work and in Section VIII we present conclusions. We conclude the paper in Section IX with a discussion of future work.

II. Web Service Resource Framework (WS-RF) and Web Service Notification (WS-N)

The Web Service Resource Framework specification provides Grid computations with a service oriented interface. Although Web services are commonly used to provide legacy applications with a service oriented interface, plain Web services lack the notion of state used in long running Grid computations. The Open Grid Service Infrastructure (OGSI) [8] was originally proposed to address this problem. This specification defines a Grid service to be a Web service conforming to a set of conventions, therewith introducing the concepts of stateful Web service instances, common properties that can be read and written, asynchronous notification of state changes, references to instances of services, and collections of service instances. In early 2004, OGSI was refactored into WS-RF and WS-N [9] in order to harmonize it with the evolving Web service standards. The WS-RF set of specifications comprises features related to

Part of this work is funded by the European IST-FP6-004559 project SODIUM (Service Oriented Development In a Unified fraMework) and the European IST-FP6-15964 project AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)
0-7695-2585-7/06 $20.00 © 2006 IEEE

the basic definition of a Grid service. It defines the implied resource pattern which allows clients to access the stateful resource using a well-defined and well-understood interface as defined in WS-Resource [10]. Additionally it defines operations for lifecycle management, property manipulation and service groups specified in WS-ResourceLifetime [11], WS-ResourceProperties [12] and WS-ServiceGroup [13] respectively. The WS-N set of specifications defines the publish-subscribe interaction patterns in WS-BaseNotification [14], WS-BrokeredNotification [15] as well as in WS-Topics [16].

A. Web Service Resource Framework (WS-RF)

1) WS-Resource: This specification defines the representation of a Grid service as a WS-Resource. It does so by defining the implied resource pattern according to which a Web service, referred to as WS-Resource, is used to access the state of the corresponding resources. One WS-Resource can be used to access several different resource instances. In order to do so, the endpoint reference defined in the WS-Addressing [17] specification is used to identify: 1) the WS-Resource using a URI, and 2) the resource instance using a custom opaque identifier.

2) WS-ResourceLifetime: The WS-ResourceLifetime specification defines two mechanisms for ending the lifetime of a resource. The first destroys the WS-Resource immediately. The second method allows the scheduled (lease-based) destruction of a resource. Scheduled destruction is done by setting the lifetime property associated with each resource.

Although resource instantiation is an integral phase of the lifecycle of a resource, it is not covered by this specification. The reason is that the mechanism used to construct a new resource instance is highly dependent on the specific kind of resource.

3) WS-ResourceProperties: This specification defines how properties of a resource are defined and how they can be accessed (by using a pull mechanism) and be modified through the WS-Resource. The properties of a resource are defined in a XML document which can be retrieved and queried by clients.

4) WS-ServiceGroup: The WS-ServiceGroup specification defines how service groups can be used to simplify the management and discovery of groups of WS-Resources. A service group is defined to be a set of WS-Resources conforming to a criterion associated with the group. Groups can then be queried in order to find all their members.

B. Web Service Notification (WS-N)

1) WS-Topics: This specification defines how to describe the topics a client can subscribe to. These topics are defined as properties and can therefore be found in the same XML property definition document.

2) WS-BaseNotification: The WS-Notification specification defines a push mechanism by which clients can be informed about events (e.g., state changes) occurring at the resource using an asynchronous event-notification pattern. To do so, clients subscribe to topics and will subsequently receive notifications from the WS-Resource. Clients cannot unsubscribe from a topic but can cancel their subscription using the same soft-state mechanisms as for resources.

III. Grid Workflows with JOpera

In this section we provide a high level description of a Grid workflow engine. We do so in order to describe the constraints imposed by the architecture of Grid workflow engines and to illustrate the mapping between processes and Grid services. We take JOpera [7] as an example.

A. Process Design and Execution

Using JOpera, developers compose Grid services into processes which are then automatically published as Grid services. The processes are visually composed out of different heterogeneous tasks (mixing coarse grained Grid and Web service invocations with fine grained Java snippets) which are linked by a control flow and a data flow graph [18]. The control flow defines the partial order of invocation of the tasks while the data flow is a directed graph which defines the data to be copied between the tasks, i.e., what data is copied between output and input parameters of tasks.

For each execution, a new instance of a process is created. The runtime state of a process instance consists of all data associated with the execution. This includes all the values of the input and output parameters of the tasks and the process, as well as process and task attributes written by the execution engine (e.g., execution status, debugging, profiling and lineage tracking information, and other execution related metadata). The state of the computation can be stored persistently in a database.

B. Execution Engine Architecture

The JOpera workflow execution includes of following components (Figure 1).

Execution Engine API: The JOpera execution engine provides an API for clients to issue commands and interact with the engine and the processes deployed therein. Once a process is deployed, a client can request to start it. To do so, the engine instantiates the process and begins with the execution. The API provides clients with the ability to start, to stop and to manage processes and their instances.

Execution Engine: The execution engine is in charge of executing process instances. It follows the control flow to determine what tasks to execute and the data flow for moving data between tasks. The tasks are executed through the service invocation adapters. For each process instance the engine stores intermediate and final results in the persistent storage.

Service Invocation Adapters: The engine dispatches task executions to the Grid service adapters. These adapters carry out the actual invocation of the Grid service. The execution engine can use different adapters for different kinds of services.

C. Mapping Processes to Grid Services

The mapping of a process and its state into a WS-Resource is as follows [5]. The persistent state of a process instance is mapped to the state of a resource. This means that the

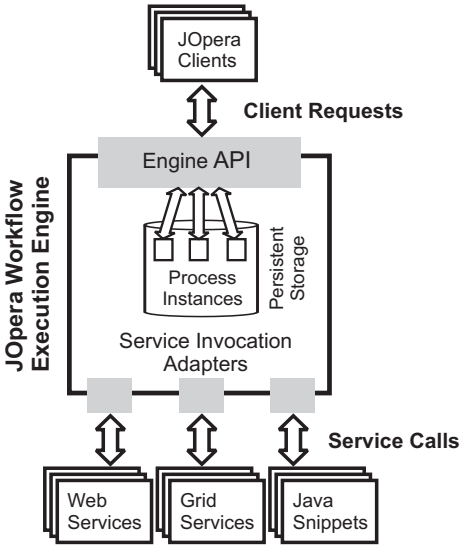

Fig. 1. Architecture of the JOpera workflow execution engine

WS-Resource interface is used to access and control the execution of the process corresponding to the resource. When a resource is created, a new process instance begins execution. When a resource is destroyed, the execution of the process is interrupted and its state discarded. Properties of a resource are directly mapped onto the elements of the state of the corresponding process instance. Thus, all data parameters and attributes can be read (and updated) by clients. Similarly, all elements of the process state are considered topics to which clients can subscribe. Once the particular element of the state (e.g., the execution status of a task) changes, the client will receive a notification. In this way, Grid clients can initiate the execution and track the progress of a long running Grid process using a standardized mechanism.

IV. Mirrored Architecture

In this section we describe a first implementation of the mapping of processes to resources using the WS-Core implementation of WS-RF and WS-N. WS-Core provides a complete implementation of the WS-RF and WS-N specifications [19]. Being written in Java, it can easily be integrated with the JOpera engine. At the end of this section we illustrate the problems that stem from this approach and motivate the need for a more lightweight implementation.

A. Architecture

The mirrored architecture (Figure 2, left) builds on the idea of using the WS-Core implementation and to follow its programming model. This prescribes to develop a Web Service, the WS-Resource, as well as a resource. Web service and resource are linked by the *ResourceHome* which is used to create, find, manage and potentially persist resources. Following this model, we have mapped JOpera processes into resources, developed the necessary WS-Resource and used the *ResourceHome* accordingly. With this, the state of the resource (located in the WS-Core hosting environment) reflects the state of the corresponding process instance (located inside the underlying Grid workflow engine). Furthermore, only one

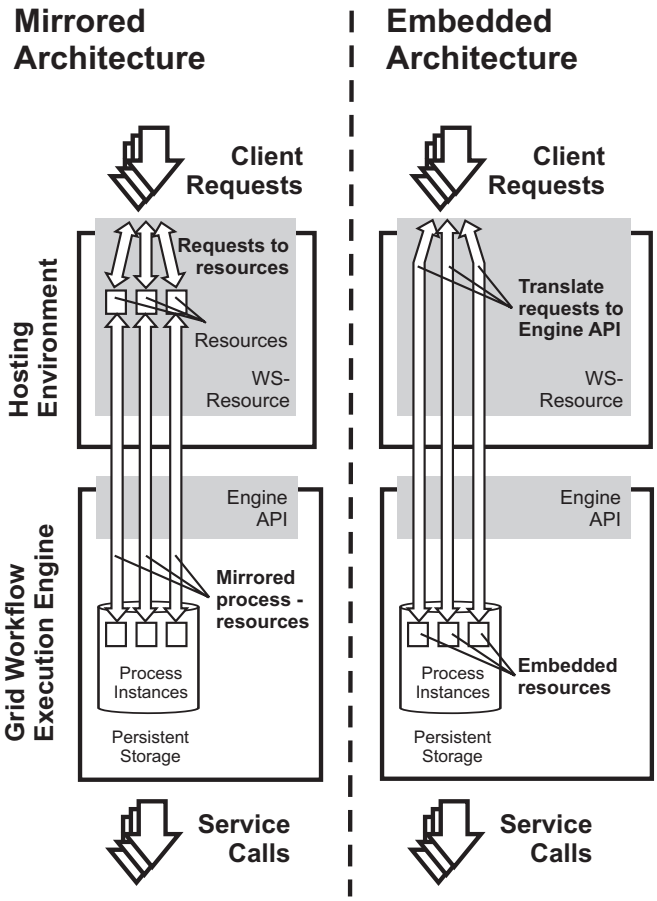

Fig. 2. Mirrored Architecture (left) discussed in Section IV and Embedded Architecture (right) presented in Section V

WS-Resource needs to be deployed for each process. This interface provides access to all of its resource instances as they are addressed by the endpoint reference sent along with each client request.

In the following, we present how each part of the standardized interface has been implemented in this mirrored architecture.

1) Lifecycle management: When receiving a client request to instantiate a process, the WS-Resource will first create a resource in the hosting environment. Doing so will trigger the resource to also create a process instance in the underlying execution engine using the engine's API. Since resource instance and process instance are tightly coupled, the resource instance is identified by the process instance identifier, which is returned by the WS-Resource to the client.

Should a client request the immediate termination of a process instance, the WS-Resource will destroy its resource instance which in turn destroys the process instance. In case

of a scheduled destruction, this implementation relies on the mechanisms provided by WS-Core to destroy the resource instance in due time. Also in this case, destruction of the resource instance will lead to the destruction of the process instance.

2) Properties: The WS-Resource provides clients with access to the process instance properties. To do so, the WS-Resource maintains a list of all properties that can be read and written by clients. This list is the same for all instances of a process and can therefore be defined when the process is deployed. A resource instance maintains a copy of all properties and synchronizes each of these with the properties of the process instance. This is very efficient for read operations where the value need not be read from the process instance but can be returned immediately upon request. If, however, the value of a state element of a process instance changes, this method incurs the overhead of having to update the cached property value in the resource instance. Similarly, changes of resource properties triggered by clients (e.g., through a set property operation) need to be forwarded to the underlying engine's API.

3) Notifications: As mentioned earlier, the topics to which clients can subscribe are the elements of the execution state of a process instance. Once a client is subscribed to a particular topic, it will receive a notification once the value of the corresponding process state element changes.

In order to send out notifications, the resource instance registers listeners with the engine for the property values the clients have subscribed to. When the value of a topic changes, the engine calls back the resource instance and informs it about the change. The resource instance will then notify the WS-Resource which in turn will send out notification messages to subscribed clients.

B. Initial measurements

The goal of this experiment is to perform a basic evaluation of the mirrored architecture and study the design problems that limit scalability.

This initial experiment has been carried out by running the Grid service provider 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 50 dual processor (1Ghz) nodes connected with a 100Mb/s local area network. Up to two clients were run on each node. In order to reduce the size of the configuration space, the WS-Core installation was configured to use a pool of 100 threads to handle client requests and all security mechanisms were disabled.

First, we measured the throughput of the raw hosting environment. By deploying a simple Web service and invoking it with up to 100 concurrent clients, we observed that this amount of clients is not enough to saturate the Web service (Figure 3). Unfortunately, this result does not hold for the WS-Resource used to create resources by calling the Grid workflow engine. For this case, our results indicate that the throughput reaches the maximum at 36 resource creations per second with only 5 clients. Clearly, this performance is not enough in large scale applications.

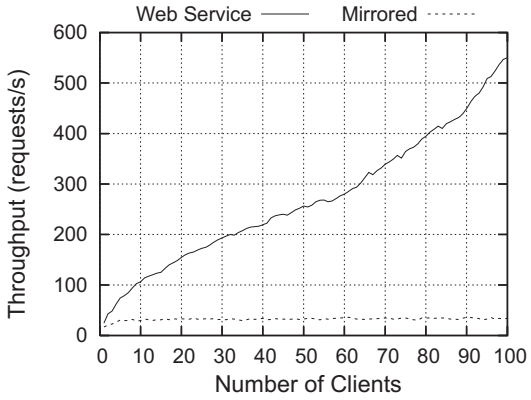

Fig. 3. Comparison of the throughput when calling a raw Web service and the create resource operation of the WS-Resource

In order to find the bottleneck, we have analyzed the execution profile of resource creation requests. WS-Core builds on its own hosting environment. Client requests are accepted by a *ServiceDispatcher* which uses a pool of *ServiceThread*s to service them. In our experiments, this pool was configured to use up to 100 threads, therefore we believe that the scalability problem is not due to a misconfiguration of the system. Moreover, as the throughput of the plain Web service indicates, the loss in performance happens past this point.

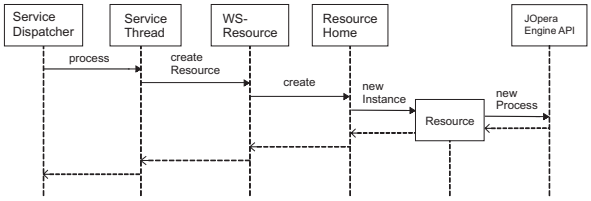

Fig. 4. Sequence diagram of the resource creation operation

Following the profile shown in Figure 4, a *ServiceThread* locates the deployed Web service (in our case the WS-Resource) and calls it to execute the requested operation on the resource. To do so, first the resource object is located (or created), one of its methods is called and the results are returned to the client. In this architecture, resources are managed by a central component, the *ResourceHome*. Thus, when a resource is created, the WS-Resource delegates this operation to the *ResourceHome*, which instantiate a new *Resource* object. As part of the implementation of our mapping, the constructor of the *Resource* class instantiates a process by calling the engine's API.

We measured the time needed for each of these steps in case of 1 client and 20 concurrent clients (Figure 5). The time spent in the *create* method is much bigger than the time actually required for instantiating the resource in case of 20 clients. The relative difference between these two times is not as big in case

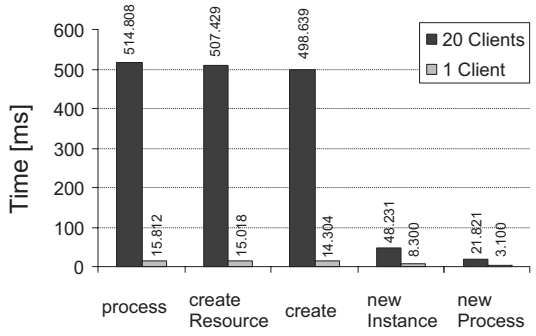

Fig. 5. Average time required for each of the methods called when creating a resource (as shown in Figure 4) in case of 1 and 20 client(s)

only one client is creating resources. This clearly indicates a contention problem in the *create* method preventing multiple clients from creating resources simultaneously.

From this analysis it can be concluded that while linking the WS-Resource and the resources using the *ResourceHome* adds flexibility (in terms of finding, managing and making resources persistent), it also involves synchronized access to the resources. This synchronization is thus the main reason for the increased time required to create resources when multiple clients are doing so concurrently. This bottleneck leads to the performance loss observed in Figure 3.

V. Embedded Architecture

As shown in the previous section, synchronized access to the *ResourceHome* can amount to a performance penalty when a large number of clients accesses a resource simultaneously.

In the context of publishing processes as resources, the mechanisms provided by the *ResourceHome* are not strictly needed. Equivalent functionality to the one provided by the *ResourceHome* is available in the vast majority of Grid workflow engines. As a consequence, the API of the engine can be directly used to find, manage, destroy and create the process instances. In this architecture, the WS-Resource only provides a translation of all client requests in terms of the engine API. This way, the state of the resources becomes embedded into the engine. Although this involves accessing the engine for all resource operations, it removes unnecessary redundancies and, as we are going to show, provides much better scalability.

Figure 2 (right) depicts the embedded architecture of the WS-RF specification implementation in the context of JOpera. Clients interact with the WS-Resource hosted on top of the execution engine API. The WS-Resource uses the API to access the resources stored in the persistent storage. Different processes designed and run in JOpera are mapped to different WS-Resources. Still, there is only one WS-Resource made accessible in the hosting environment. This WS-Resource uses the URI to map the request to a given process and the endpoint in order to map it to a particular instance of the process.

Upon arrival of requests, the WS-Resource will directly use the engine's API to access the persistent storage in order to store and retrieve information about the state of instances. With this solution access to resources is synchronized on the lowest possible level: only read and write access to the same resource is synchronized by using thread-safe data structures.

1) Lifecycle management: When receiving a request to create a resource, the WS-Resource reads the process name out of the URI from the request received and instantiates a process. The identifier of the process is returned as the ID of the resource. Destruction, be it immediate or scheduled, removes the accumulated state of the process instance and if necessary also interrupts process execution.

2) Properties: Properties in the embedded implementation are handled as follows: when a client requests to read a property of a resource instance, the WS-Resource uses the engine API to retrieve the value from the persistent storage. In case of a write property request, the WS-Resource will also use the engine's API to write the value directly into the persistent storage.

3) Notifications: Subscriptions to topics are also considered to be resources according to the specification [14]. In the embedded implementation this would mean to map them to a process and to create a process instance every time a client subscribes. Subscriptions however do not require the flexibility provided by processes and it is therefore more efficient and simpler to store them in a list located in the WS-Resource. To be able to send out notifications once changes in the state of the resource occur, we use a mechanism provided by the JOpera execution engine. The WS-Resource registers listeners with the engine and will receive notifications once state changes occur. It will then match subscriptions with the state changes and will send out the corresponding notifications.

VI. Comparison

The goal of this second set of measurements is to show and compare the performance under heavy load induced by an increasing number of clients concurrently accessing a resource through the WS-Resource interface using the mirrored and the embedded architecture. The same setup that has been used for the initial measurements presented in Section IV-B has also been used for these experiments.

A. WS-Resource Creation

Although resource creation is not specified in the WS-RF set of specifications, it remains a very important operation in a Grid infrastructure. In our case, it represents the submission of a new computation to be started by the Grid workflow engine. In a first series of experiments we have therefore measured the response time and the throughput when a variable number of clients simultaneously initiates the execution of a Grid process instance by creating the corresponding resource.

Figure 6 shows the number of resources created per second when using both architectures serving up to 100 clients at a time. Each client creates 1000 resources as fast as possible. The mirrored architecture reaches its peak throughput of 36 new resources/second with only 5 clients. We have investigated this problem and presented our findings in Section IV-B. The embedded architecture does not suffer from this limitation and

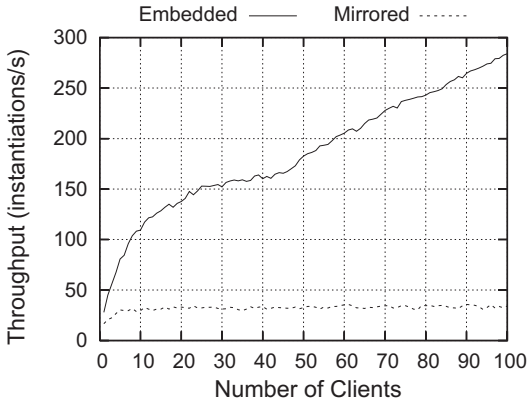

Fig. 6. Throughput of the create resource operation

it reaches a throughput of over 250 resources/second with 100 clients. In this case, we were not able to saturate the system.

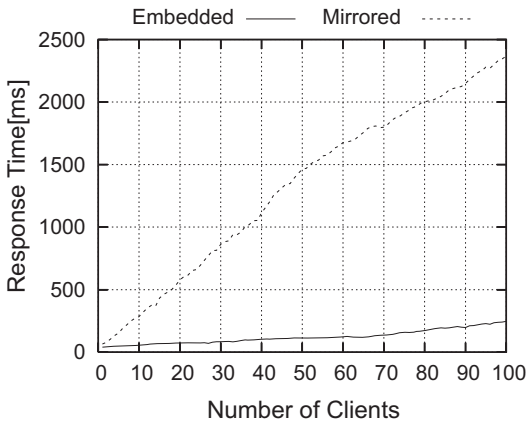

Fig. 7. Response time of the create resource operation

Figure 7 shows the response times for resource creation. With an increasing number of clients concurrently creating resources, the response time increases as is to be expected. In case of the mirrored architecture, the response time grows very high, to up to 2.4s in case of 100 concurrent clients. This is because this implementation is only able to serve up to 36 clients requests per second while additional requests will have to wait. The response time in case of the embedded architecture grows one order of magnitude less (0.25s), as the system has enough capacity to deal with 100 clients.

B. WS-Properties

In these experiments an increasing number of clients reads the property of a single resource 1000 times. The response time for both architectures only grows slowly with an increasing number of clients (Figure 8). The throughput increases linearly when more clients read the properties in case of both architectures.

In this case, the mirrored architecture outperforms the embedded one. This can be explained by a caching effect. The

Fig. 8. Response time of the read property operation

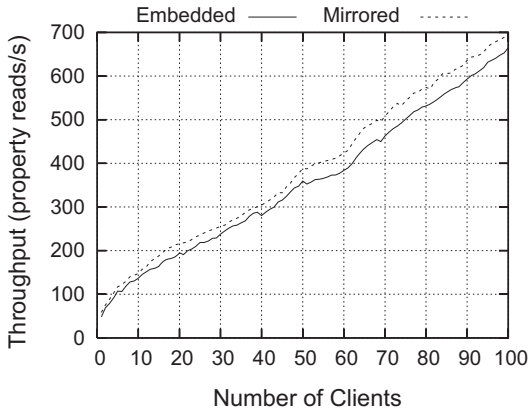

Fig. 9. Throughput for the read property operation

state of the resource is kept closer to the clients and is thus faster to access. Given that all clients were reading from the same resource, the cache in the *ResourceHome* is able to fulfill all requests. In case of the embedded architecture, requests of reading property values have to be mapped to queries to the underlying engine API, which makes their execution path about 10ms longer with 100 concurrent clients. Nevertheless, this overhead seems acceptable in view of the advantages in other performance measurements.

C. WS-Notification

In a next series of experiments we have measured the different implementations of WS-Notification. In the first experiment we have measured the time it takes a client to subscribe to a certain topic. To do so, an increasing number of clients (1 to 100) subscribes to 1000 topics as fast as possible. We have measured the throughput and the response time for both architectures. In case of the mirrored architecture, we used two configurations of WS-Core. The first stores subscriptions in memory, the second one makes them persistent on disk.

The mirrored architecture relies on WS-Core to manage its subscriptions. WS-Core follows the WS-N specification

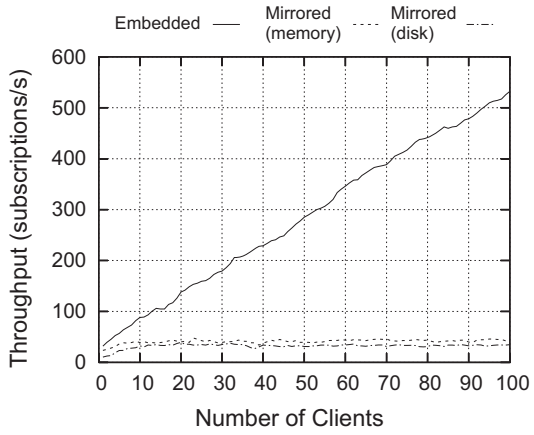

Fig. 10. Throughput for the subscribe operation

and treats subscriptions as resources. Thus, it uses a *SubscriptionHome*, which inherits the scalability problems of the *ResourceHome* implementation. As our measurements indicate (Figure 10), the throughput reaches a maximum at 50 subscriptions/second. As expected, persistent subscriptions are more expensive than volatile ones (Figure 11).

As previously described, the embedded implementation relies on the engine's API to maintain the client subscriptions and only maps the WS-N topics to the addressing mechanism used by the engine to identify each element of a process execution state which may change. This greatly speeds up subscribing to a topic, as we observed from the throughput which grows linearly with the increasing number of concurrent subscribers.

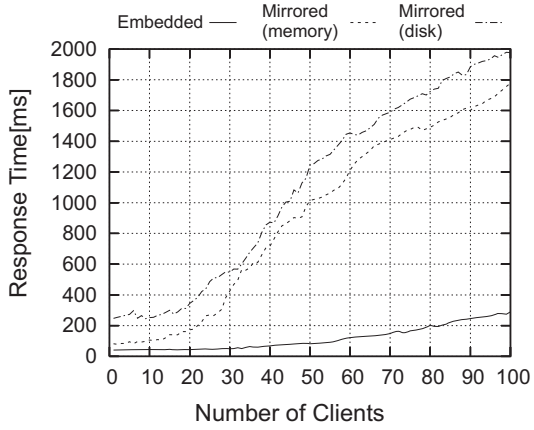

Fig. 11. Response time for subscribe operation

In the next experiment we measured the throughput of the two architectures in terms of sending out notifications. Although there are many combinations of the number of subscribers, number of resources and number of subscriptions to topics that could affect this performance metric, due to space limitations we focus on one setup. Each client (from 1 to 100) subscribes to one topic of the same shared resource. This resource goes through 100 state changes, which will be reported to all subscribed clients by sending them 100 notification messages. Thus, in the experiment, the Grid service sends from 100 notifications (with 1 client) up to 10'000 notifications (with 100 clients).

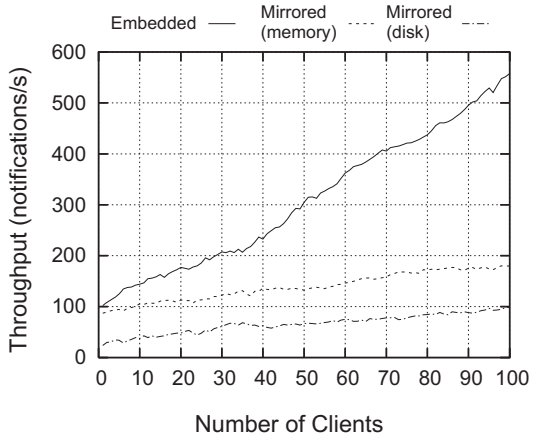

Fig. 12. Throughput for matching events with subscriptions and sending notifications

For both architectures, the throughput does not saturate with 100 clients. However, the mirrored architecture is only able to send 180 notifications/second (with volatile subscriptions) and 100 msg/s (persistent subscription). The embedded architecture sends out notifications at a higher rate (550 msg/s). The reason for this performance improvement lies in the fact that in the mirrored implementation the local copy of the state needs to be updated whenever a notification from the engine is received. Furthermore, this implementation relies on WS-Core to call back subscribed clients, and it appears that each notification is sent out sequentially. Instead, in case of the embedded architecture this is not a limitation as clients are notified in parallel.

VII. Related Work

In addition to the one presented in this paper, there are currently five more implementations of WS-RF and WS-N. They differ mostly in the programming model as well as the implementation language. The most prominent implementation is the Java version of WS-Core which we used to implement the mirrored architecture and is distributed with the Globus Toolkit [6]. This code is also used in the two Apache projects Hermes (WS-N) and Apollo (WS-RF). An additional implementation of the standards which is part of the Globus Toolkit is implemented in C and lets one develop Grid services in C. pyGridWare [20] is also part of the Globus Toolkit. It allows the user to rapidly develop Grid services in Python. Similarly, WSRF.NET [21] is used to develop Grid services in any .NET language. With this implementation developing Grid services is not much different than programming Web services: the developer only needs to annotate what parts of

the service should be made persistent. Lastly there is also a Perl implementation of the standards called WSRF::Lite [22].

These five implementations have been compared in terms of functionality and performance in [19]. This comparison however only focused on evaluating the performance of a single client setup (both, distributed on two machines and co-located). In our experiments we have used a setup with increasing numbers of clients thereby showing how the system copes with a larger number of concurrent requests.

VIII. Conclusion

In this paper we present and compare two different strategies for implementing the WS-RF and WS-N standards for process based systems. Our work extends the mapping presented in [5], which defined how to bridge two different levels of abstractions: the WS-Resource and the Grid workflow.

We first consider an architecture where this mapping is implemented by mirroring the persistent state of the execution of a Grid workflow as the state of the published resource. This has the advantage, as our experimental results show, that clients can directly access such resources without the overhead of going through additional layers of the system. Furthermore, wrapping the API of a Grid workflow engine within WS-Core also reduces the development effort required to make the Grid workflow engine standard compliant, as the Grid infrastructure providing the implementation of such standards can be reused. The disadvantage of this approach lies in the redundancy introduced in the system, where the state of the resources is duplicated across different components. This has the drawback that concurrency control is performed early in the request processing pipeline and the scalability of the system suffers when facing a large number of concurrent clients.

As an alternative, we present a second architecture, where the state of a resource is embedded into the underlying process engine and the mapping involves a direct translation of the client requests from the standardized WS-RF interface to the engine API. Although with this solution all requests have to be serviced through the engine, incurring in slightly higher response time, the embedded architecture scales much better as the concurrency control is performed on a more fine grained level within the Grid workflow engine.

IX. Future Work

As part of future work we plan to implement WS-BrokeredNotification [15] as well as security mechanisms for client authentication (using X.509 signing of messages), authorization and message encryption (using HTTPS).

References

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

[2] P. Nibblet and S. Graham, "Events and service-oriented architecture: The OASIS Web Services Notification specifications," *IBM Systems Journal: Service-Oriented Architecture*, vol. 44, no. 4, pp. 869–886, 2005.

[3] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, "Mapping Abstarct Workflows onto Grid Environments," *Journal of Grid Computing*, vol. 1, no. 1, 2003.

[4] 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*. Washington, DC, USA: IEEE Computer Society, 2003, p. 198.

[5] T. Heinis, C. Pautasso, O. Deak, and G. Alonso, "Publishing Persistent Grid Computations as WS Resources," in *Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.

[6] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, Summer 1997.

[7] C. Pautasso, "JOpera: Process Support for more than Web services," http://www.jopera.org.

[8] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, D. Snelling, and P. Vanderbilt, "Open Grid Services Infrastructure (OGSI) Version 1.0." 2003, http://www.globus.org/alliance/publications/papers/Final_OGSI_Specification_V1.0.pdf.

[9] 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.

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

[11] L. Srinivasan and T. Banks, "Web Services Resource Lifetime 1.2," OASIS, June 2004, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-pr-01.pdf.

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

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

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

[15] D. Chappell and L. Liu, "Web Services Brokered Notification 1.3," OASIS, 2005, http://www.oasis-open.org/committees/download.php/13485/wsn-ws-brokered_notification-1.3-spec-pr-01.pdf.

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

[17] D. Box and F. Curbera, "Web Services Addressing (WS-Addressing)," W3C, August 2004, http://www.w3.org/Submission/ws-addressing/.

[18] C. Pautasso and G. Alonso, "The JOpera Visual Composition Language," *Journal of Visual Languages and Computing*, vol. 16, no. 1–2, pp. 119–152, 2004.

[19] M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, J. Bester, J. Gawor, S. Lang, I. Foster, S. Meder, S. Pickles, and M. McKewon, "State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, Research Triangle Park, NC, USA, 2005.

[20] "pyGridWare: Python Web Services Resource Framework," http://dsd.lbl.gov/gtg/projects/pyGridWare/.

[21] M. Humphrey, G. Wasson, M. Morgan, and N. Beekwilder, "An Early Evaluation of WSRF and WS-Notification via WSRF.NET," in *2004 Grid Computing Workshop (associated with Supercomputing 2004)*, Pittsburgh, PA, USA, 2004.

[22] "WSRF::Lite – Perl Grid Services," http://www.sve.man.ac.uk/Research/AtoZ/ILCT.