Adaptation of Legacy Software to Grid Services

Bartosz Baliś[1,2], Marian Bubak[1,2], and Michał Węgiel[1]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
{balis,bubak}@uci.agh.edu.pl, mwegiel@student.uci.agh.edu.pl

Abstract. Adaptation of legacy software to grid services environment is gradually gaining in significance both in academic and commercial settings but presently no comprehensive framework addressing this area is available and the scope of research work covering this field is still unsatisfactory. The main contribution of this paper is the proposal of a versatile architecture designed to facilitate the process of transition to grid services platform. We provide thorough analysis of the presented solution and confront it with fundamental grid requirements. In addition, the results of performance evaluation of a prototype implementation are demonstrated.

Keywords: Legacy software, grid services, design patterns

1 Introduction

In this paper we intend to propose a versatile framework enabling for semi-automated migration from legacy software to grid services environment [1,8]. A detailed analysis of the presented solution can be found in [2] which contains rationale supporting our design decisions and justification for rejection of other approaches. Here, we outline the most recent stage of evolution of our concept and demonstrate the results of performance evaluation of its prototype implementation.

The need for a framework enabling for cost-effictive adaptation of legacy software to grid services platform is a widely recognized issue. Nonetheless, it is still addressed inadequately as compared to its significance. We have summarized the related research, which originates from both academia and industry, in [2]. Recent experiences in this area were presented at the GGF9 Workshop [3]. Currently no comprehensive framework facilitating migration to grid services environment is available and, as discussed in [2], existing approaches [4,5] possess numerous limitations.

2 System Structure

In the proposed architecture, we can distinguish three main components: **back-end host**, **hosting environment** and service **client**. As depicted in Fig. 1, they

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 26–33, 2004.

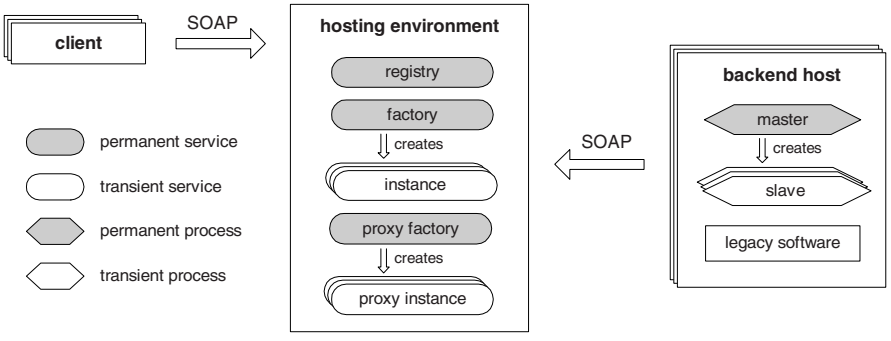

Fig. 1. System architecture

are potentially located on separate machines. The presented diagram reflects the configuration for a single legacy system which is exposed as a grid service.

Communication between components takes place by means of SOAP messages. Data is exchanged by remote method invocations performed on services deployed within the hosting environment.

2.1 Backend Host

Backend host represents the machine on which legacy software is installed and executed. In order to enhance performance and scalability as well as improve reliability and fault tolerance, we may decide to deploy several redundant copies of a single legacy application on different computing nodes, possibly in various geographic locations. For this reason, in the context of a particular service, we can end up having multiple backend hosts constituting dynamic pool of available processing resources.

We propose the following approach to maintenance and utilization of such a conglomerate.

- Backend hosts do not fulfill the function of network servers. Instead, they are devised to act as clients. Legacy applications are not required to be directly accessible from the outside.
- We employ registration model. Backend hosts are expected to volunteer to participate in computations and offer their capabilities of serving client requests.

Each backend host comprises two flavors of processes: **master** and **slave** ones. The master process is a permanent entity responsible for host registration and creation of slave processes. Whenever the load of a particular backend host in comparison with its resources allows it to serve a new client, the master process is obliged to report this fact by calling the registry service. Along with this invocation, the estimated processing capability (in the sense of a certain metric) and validity timestamp are provided. The call blocks until the given time

expires or one of the pending clients is assigned. In the latter case, a new slave process is spawned in order to take over further processing. There is only one master process per host. Slave processes are transient – as clients come and go, the number of concurrently running slave processes changes respectively. These processes are in charge of direct collaboration with legacy software on behalf of the clients that they represent. This cooperation can assume various forms and range from local library usage to communication over proprietary protocols. The primary activity performed by each slave process is intercepting subsequent client invocations, translating them into legacy interface and delivering the obtained results. Slave processes communicate with the service container by means of blocking method invocations performed on the proxy instance.

2.2 Hosting Environment

The hosting environment contains a collection of grid services that jointly fulfill the role of shielding clients from unmediated interaction with backend hosts. For each legacy system, there are three permanent services deployed: **registry**, **factory** and **proxy factory**. Depending on the number of simultaneously served clients, we have varying number of transient services, two types of which exist: **instance** and **proxy instance**.

Access to all services is authorized and restricted to subjects holding adequate identities. In case of internal services, such as registry, proxy factory and proxy instance, permission is granted on the basis of host certificates. For the remaining services, namely factory and instance, we employ user certificates. Clients are not eligible to use internal services. Moreover, both types of instances can be accessed exclusively by their owners.

Registry is a service that controls the mapping between clients and backend hosts, which offered their participation in request processing. It is obliged to assign consecutively appearing clients to the chosen backend hosts. For the purpose of selection, registry maintains priority queue to which pending master processes are inserted. The ordering criterion is processing capability advertised during each registration.

The remaining services form two pairs, each consisting of factory and the corresponding instance. We discriminate two types of factories and instances: ordinary and proxy ones. The former are designated for clients whereas the latter are auxiliary entities providing mediation facilities. Proxy entities are logically separated because they serve internal purposes and should be transparent to the clients.

3 System Operation

Let us now turn our attention to the scenarios that take place during typical client-service interaction. From the client's perspective, the following sequence of actions normally is performed: (1) new service is created, (2) certain methods are invoked, (3) the service is destroyed. On the internal system side, each of the

above-mentioned steps involves a number of activities. They are schematically presented in Fig. 2 as a time sequence diagram. The prerequisite for all depicted scenarios is an operative environment with permanent services up and running. Moreover, before any client interaction takes place, at least one master process should be registered in the context of the considered service. It is essential since otherwise no actual processing would be feasible.

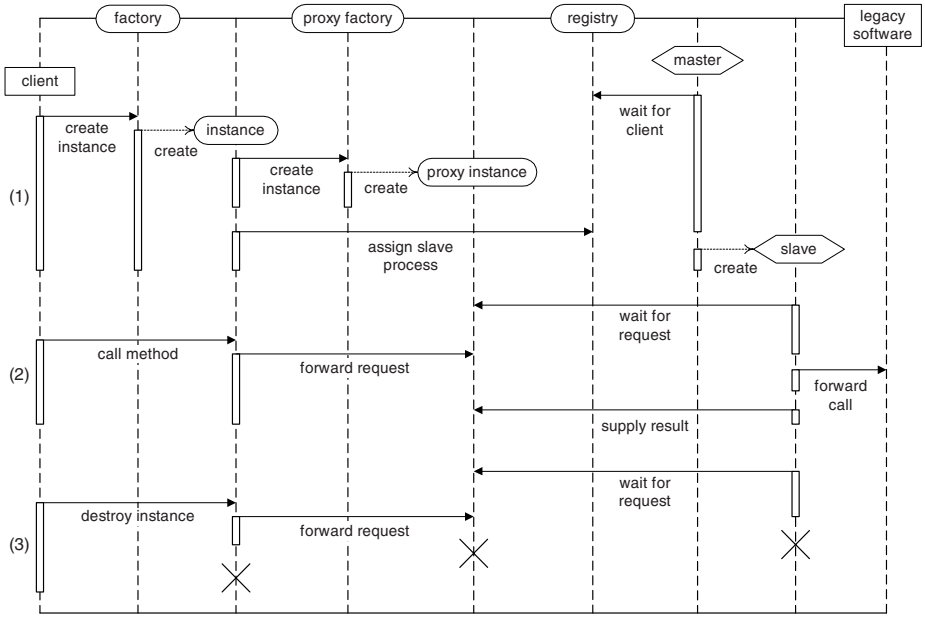

Fig. 2. Scenarios: (1) service construction (2) method invocation (3) service destruction

All scenarios are triggered by clients. Procedures accompanying service construction and destruction are executed by container's callback methods connected with lifetime management.

3.1 Service Construction

The scenario of service construction encompasses several actions. In the first place, proxy factory is contacted and the corresponding proxy instance is created. Following this, registry service is invoked. As a result of this call, one of the registered master processes is selected and provided with the location of the newly deployed proxy instance. The chosen master process spawns a new slave process which in turn starts to wait for client requests using the created proxy instance.

3.2 Method Invocation

After a successful initialization, three transient entities are created: instance, proxy instance and slave process. Since they cannot be shared, we can treat them as a private property of a particular client. Whenever a client invokes a method, its complete description, together with passed parameters, is forwarded to the proxy instance. The request is then further propagated to slave process. The call blocked on waiting for client action returns and slave process translates the obtained invocation to its legacy equivalent. Next, legacy processing is started. As soon as it is finished, the obtained results are supplied to the proxy instance via a separate asynchronous method invocation. This in turn causes that control is transferred back to service instance and after that to the client from which it originated.

Effective scenario of method invocation is critical to the system performance due to the frequency with which it occurs. The communication between instance and proxy instance takes place within the borders of a local container so it should not pose a bottleneck. Apart from this, we have two internal remote invocations per client call.

3.3 Service Destruction

Grid services support explicit and soft state mode of destruction. The former takes place on client demand whereas the latter is executed automatically when instance lifetime expires. The scenario of service removal is relatively simple. The request of destruction is forwarded to proxy instance. Once it is delivered, it is intercepted by the slave process, which terminates its execution. In the next steps, proxy instance and instance are deleted by the container.

4 System Features

Having examined individual components and scenarios comprising the proposed architecture, we will now characterize it from a wider perspective and confront its properties with requirements that have to be satisfied in a grid environment.

4.1 Security

We can point out two major aspects concerning the security of our infrastructure.

- There is no need to introduce open incoming ports on backend hosts.
- It is possible to obtain the identity of machines with which we cooperate and verify that the processing is delegated only to trusted nodes.

We owe both these advantages to the fact that backend hosts act as clients rather than servers. As we stated earlier, we perform authentication and authorization procedures. If needed, communication integrity and privacy can be ensured by means of digital signatures and encryption. In our solution security configuration

can be thought of as two lists of identities. First for clients that are entitled to use our service, and second for hosts that are permitted to perform processing for our service. In consequence, maintenance of security policy should not involve excessive administrative effort.

4.2 Scalability

The combination of several factors enables to achieve high scalability of our architecture. Since actual processing is always delegated to backend hosts, computations can be heavily distributed across a large number of nodes. Furthermore, service instances residing in the container do not incur large resource consumption. Their activity is in fact reduced to message forwarding. Moreover, thanks to registration model we earn automatic load balancing. Backend hosts decide about client acceptance themselves and volunteer only when they can guarantee that processing can be finished within reasonable amount of time. This brings high responsiveness to unexpected changes in utilization. In addition, master processes advertise temporal capabilities of machines on which they are executed. This can be used for resource reservation, which is essential for the assurance of the quality of service.

4.3 Fault Tolerance

The proposed solution offers a high degree of resilience to component failures. The most important reason for this is the fact that processes are not bound to any specific endpoint. Being clients, they can arbitrarily change their location between subsequent method calls without any serious repercussions. Thus, process migration is supported. In case of legacy systems which maintain internal state or work in a transactional manner, process migration has to be disabled or automatic repetition of all preformed method invocations should be allowed.

Immunity to sudden changes of configuration is ensured by the eager registration model. At any point in time, it is not necessary to poll backend hosts in order to determine operative nodes. We have up-to-date system image sustained continuously.

4.4 Portability

Unquestionably, an important advantage of our architecture is the fact that we make no assumptions as regards programming language or platform on which both backend host and hosting environment are based. Our solution is versatile enough to accommodate a variety of legacy systems and container implementations. Furthermore, legacy software can remain in the same location where it was initially installed. There is no necessity of moving programs between machines or changing their configuration.

5 Performance Evaluation

In order to estimate the communication overhead introduced in the proposed framework, we developed a simple benchmark – echo grid service exposing a single method which repeated the string passed as its parameter. There were two functionally equivalent implementations of the above service. Measurements were performed on the client side and embraced solely method invocation scenario. No security mechanism was employed. In consequence, the results reflect the overhead introduced merely by our core architecture. The influence of different security modes on efficiency of Globus I/O library is quantified in [6].

We performed two types of tests: bandwidth- and latency-oriented. The former used large messages whereas the latter transmitted small amount of data.

The experiment was carried out on a single-processor desktop machine running Linux operating system. It simultaneously played the role of the hosting environment, backend host and service client. As for software configuration, Globus Toolkit 3.0 [9] together with gSOAP 2.1 [10] were used. We performed time measurement for message payload being 1 and 10^5 bytes. Number of method calls ranged from 1 to 256 and 8192 for bandwidth and latency measurement, respectively.

The obtained measurement results are presented in Fig. 3. In order to enhance clarity, logarithmic scale (base 2) is used. We observe a linear dependence between the number of method calls and the time needed for their execution. Thereby, the introduced overhead can be computed as a quotient of directional coefficients of linear approximations of individual functions.

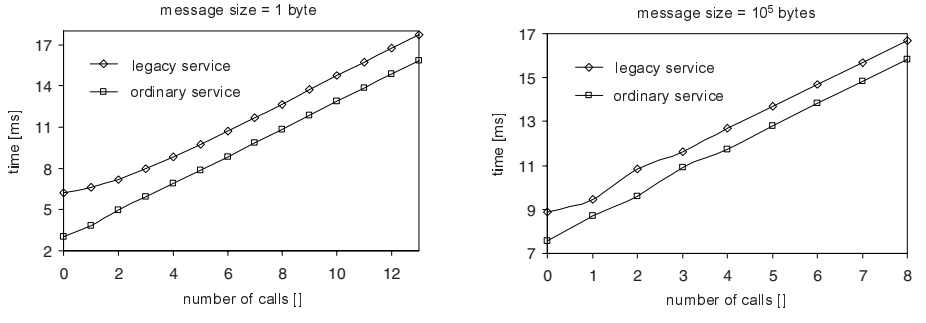

Fig. 3. Results of performance evaluation

In case of transmitting small SOAP messages, the obtained ratio was 3.65 whereas for relatively large ones we got the proportion equal to 1.79. This phenomenon can be explained if we take into account that the impact of communication overhead diminishes along with the message enlargement.

On the basis of the conducted experiments, we can draw a conclusion that in our architecture, in average case, we should expect approximately threefold decreased communication efficiency.

6 Concluding Remarks

Implementation of initial version of the presented framework took place while porting grid monitoring system, OCM-G [7], to grid services environment. This project served as a proof of the concept of the overall solution. Since then the architecture has undergone a number of refinements. Its current form is to a large extent stable. Our work is now focused on developing additional test cases in order to evaluate the proposed architecture under a variety of circumstances. The anticipated direction of project evolution is implementation of tools facilitating the application of our framework. This includes, among others, utilities allowing for automatic interface and code generation.

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid. http://www.globus.org/research/papers/ogsa.pdf
2. Baliś, B., Bubak, M., Węgiel, M.: A Framework for Migration from Legacy Software to Grid Services. To be published in Proc. Third Cracow Grid Workshop, Cracow, Poland 2003.
3. Contributions to GGF9 Workshop *Designing and Building Grid Services*. http://www.gridforum.org
4. Huang, Y., Taylor, I., Walker, D., Davies, R.: Wrapping Legacy Codes for Grid-Based Applications. In Proc. International Workshop on Java for Parallel and Distributed Computing, Nice, France, 2003.
5. Kuebler, D., Eibach, W.: Adapting Legacy Applications as Web Services. http://www-106.ibm.com/developerworks/webservices/library/ws-legacy
6. Baliś, B., Bubak, M., Rząsa, W., Szepieniec, T., Wismüller, R.: Two Aspects of Security Solution for Distributed Systems in the Grid on the Example of OCM-G. To be published in Proc. Third Cracow Grid Workshop, Cracow, Poland 2003.
7. Baliś, B., Bubak M., Funika, W., Szczepieniec, T., Wismüller, R., Monitoring Grid Applications with Grid-enabled OMIS Monitor. In Proc. First European Across Grids Conference, Santiago de Compostela, Spain, February 2003. To appear.
8. Open Grid Services Infrastructure Specification. http://www.gridforum.org/ogsi-wg/
9. Globus Project homepage: http://www.globus.org
10. gSOAP Project homepage: http://www.cs.fsu.edu/~engelen/soap.html