Functional Adaptivity for Digital Library Services in e-Infrastructures: The gCube Approach

Fabio Simeoni[1], Leonardo Candela[2], David Lievens[3],
Pasquale Pagano[2], and Manuele Simi[2]

[1] Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK
fabio.simeoni@cis.strath.ac.uk
[2] Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo", CNR, Pisa, Italy
{leonardo.candela,pasquale.pagano,manuele.simi}@isti.cnr.it
[3] Department of Computer Science, Trinity College, Dublin 2, Ireland
david.lievens@cs.tcd.ie

Abstract. We consider the problem of e-Infrastructures that wish to reconcile the generality of their services with the bespoke requirements of diverse user communities. We motivate the requirement of *functional adaptivity* in the context of *gCube*, a service-based system that integrates Grid and Digital Library technologies to deploy, operate, and monitor *Virtual Research Environments* defined over infrastructural resources.

We argue that adaptivity requires mapping service interfaces onto multiple implementations, truly alternative interpretations of the same functionality. We then analyse two design solutions in which the alternative implementations are, respectively, full-fledged services and local components of a single service. We associate the latter with lower development costs and increased binding flexibility, and outline a strategy to deploy them dynamically as the payload of *service plugins*. The result is an infrastructure in which services exhibit multiple behaviours, know how to select the most appropriate behaviour, and can seamlessly learn new behaviours.

1 Introduction

gCube[1] is a distributed system for the operation of large-scale scientific infrastructures. It has been designed from the ground up to support the full lifecycle of modern scientific enquiry, with particular emphasis on application-level requirements of information and knowledge management. To this end, it interfaces pan-European Grid middleware for shared access to high-end computational and storage resources [1], but complements it with a rich array of services that collate, describe, annotate, merge, transform, index, search, and present information for a variety of multidisciplinary and international communities. Services, information, and machines are infrastructural resources that communities select, share, and consume in the scope of collaborative *Virtual Research Environments* (VREs).

To gCube, VREs are service-based applications to dynamically deploy and monitor within the infrastructure. To the users, they are self-managing, distributed Digital

[1] http://www.gcube-system.org

M. Agosti et al. (Eds.): ECDL 2009, LNCS 5714, pp. 51–62, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Libraries that can be declaratively defined and configured, for arbitrary purposes and arbitrary lifetimes. In particular, gCube is perceived as a *Digital Library Management System* (DLMS) [5], albeit one that is defined over a pool of infrastructural resources, that operates under the supervision of personnel dedicated to the infrastructure, and that is built according to Grid principles of controlled resource sharing [8]. The ancestry of gCube is a pioneering service-based DLMS [7] and its evolution towards Grid technologies took place in the testbed infrastructure of the Diligent project [4]. Five years after its inception, gCube is the control system of D4Science, a production-level infrastructure for scientific communities affiliated with the broad disciplines of Environmental Monitoring and Fishery and Aquaculture Resources Management[2].

The infrastructural approach to DLMS design is novel and raises serious challenges, both organisational and technical. Among the latter, we notice core requirements of *dynamic service management* and *extensive development support*. The first is the very premise of the approach; a system that does not transparently manage its own services it is not a service-based DLMS. The second requirement constrains how one plans to accommodate the first; a DLMS that achieves transparencies for users but denies them to its developers is prone to error, is hard to maintain, and has little scope for evolution.

In this paper, we comment briefly on service management and development complexity, overviewing relevant parts of the gCube architecture in the process (cf. Figure 1). Our goal is not to present the solutions adopted in gCube, for which we refer to existing literature. Rather, we wish to build enough context to discuss in more detail a third requirement: the ability of the system to reconcile the generality of its services with the specific demands of its communities of adoption.

Functional adaptivity is key to the usefulness of the system; a DLMS that does not serve a wide range of user communities, and does not serve each of them well, is simply not adopted. In particular, we argue that functional adaptivity requires services that: (*i*) can simultaneously manage multiple implementations of their own interface; (*ii*) can autonomically match requests against available implementations; and most noticeably: (*iii*) can acquire new implementations at runtime. Effectively, we advocate the need for services that exhibit multiple behaviours, know how select the most appropriate behaviour, and can *learn* new behaviours.

The rest of the paper is organised as follows. We set the context for the discussion on functional adaptivity in Section 2 and show why a general solution requires multiple implementations of service interfaces in Section 3. We motivate our choice to accommodate this multiplicity *within* individual services in Section 4, and then illustrate our deployment strategy in Section 5. Finally, we draw some conclusions in Section 6.

2 Context

In a DLMS that commits to service-orientation, managing resources is tantamount to managing services that virtualise or manipulate resources. In such system, services do not run in pre-defined locations and are not managed by local administrators. Rather, they are resources of the infrastructure and they are dynamically deployed and redeployed by the system that controls it, where and when it proves most convenient. As an

[2] http://www.d4science.eu

implication of dynamic deployment, the configuration, staging, scoping, monitoring, orchestration, and secure operation of services become also dynamic and a responsibility of the system. Essentially, the baseline requirement is for a state-of-the-art *autonomic* system.

In gCube, the challenge of autonomicity is met primarily in a layer of *Core Services* that are largely independent from the DL domain and include:

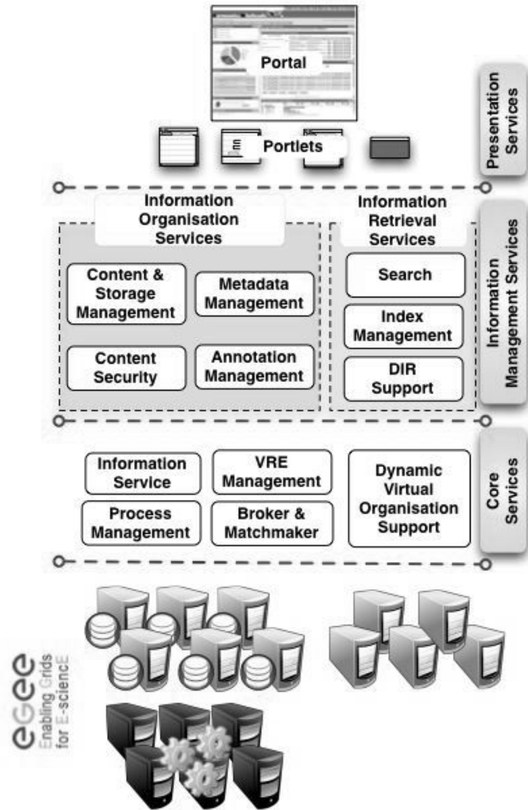

Fig. 1. The gCube Architecture

– *Process Management Services* execute declaratively specified workflows of service invocations, distributing the optimisation, monitoring, and execution of individual steps across the infrastructure [6];
– *Security Services* enforce authentication and authorisation policies, building over lower-level technologies to renew and delegate credentials of both users and services;
– *VRE Management Services* host service implementations and translate interactive VRE definitions into declarative specifications for their deployment and runtime maintenance [2];
– *Brokering and Matchmaking Services* inform deployment strategies based on information about the available resources that a network of *Information Services* gathers and publishes within the infrastructure.

The need for autonomicity bears on a second problem, the complexity of service development and the impact of the associated costs on the scope and evolution of the system. A system that aspires to manage its own services needs to raise non-trivial requirements against their runtime behaviour, and thus on how this behaviour is implemented. This adds to the complexity already associated with service development, whether generically related to distributed programming (e.g. concurrency, performance-awareness, tolerance to partial failure) or specifically introduced by open technologies (e.g. reliance upon multiple standards, limited integration between development tools, inadequate documentation). At highest risk here are the services of a second layer of

the gCube architecture, the *Information Management Services* that implement DL functionality, including:

- a stack of *Information Organisation Services* rooted in a unifying information model of binary relationships laid upon storage replication and distribution services. Services higher up in the stack specialise the semantics of relationships to model compound information objects with multiple metadata and annotations, and to group such objects into typed collections.
- a runtime framework of *Information Retrieval Services* that execute and optimise structured and unstructured queries over a federation of forward, geo-spatial, or inverted indices of dynamically selected collections [13].

gCube offers a number of tools to tame the complexity of service development, most noticeably a container for hosting gCube services and an application framework to implement them. These are the key components of *gCore*[3], a minimal distribution of gCube which is ubiquitously deployed across the infrastructure [9]. Built as an ad-hoc extension of standard Grid technology[4], gCore hides or greatly simplifies the systemic aspects of service development, including lifetime, security, scope, and state management; at the same time, it promotes the adoption of best practices in multiprogramming and distributed programming. This guarantees a qualitative baseline for gCube services and allows developers to concentrate on domain semantics. Equally, it allows changes related to the maintenance, enhancement, and evolution of the system to sweep transparently across its services at the rhythm of release cycles.

3 Functional Adaptivity

The infrastructural approach to DLMS design emphasises the generality of services, i.e. the need to serve user communities that operate in different domains and raise different modelling and processing requirements. Communities, on the other hand, expect the infrastructure to match the specificity of those requirements and are dissatisfied with common denominator solutions. Accordingly, the DLMS needs to offer generic DL services that can adapt to bespoke requirements. In gCube, we speak of the functional adaptivity of services and see it as an important implication of the autonomicity that is expected from the system.

Abstraction, parameterisation, and composition are the standard design principles in this context. One chooses generic data models to represent arbitrary relationships and generic formats to exchange and store arbitrary models. One then defines parametric operations to customise processes over the exchanged models and provides generic mechanisms to compose customised processes into bespoke workflows. In gCube, content models, format standards, service interfaces, and process management services follow precisely these design principles (cf. Section 1).

Often, however, we need novel algorithmic behaviour. In some cases we can specify it declaratively, and then ask some service to execute it. Data transformations and distributed processes, for example, are handled in this manner in gCube, through services

[3] http://wiki.gcore.research-infrastructures.eu
[4] http://www.globus.org

that virtualise XSLT and WS-BPEL engines, respectively. In other cases, parameterisation and composition do not take us far enough.

Consider for example the case of the *DIR Master*, a gCube service used to optimise the evaluation of content-based queries across multiple collections [12]. The service identifies collections that appear to be the most promising candidates for the evaluation of a given query, typically based on the likelihood that their content will prove relevant to the underlying information need (*collection selection*). It also integrates the partial results obtained by evaluating the queries against selected collections, thus reconciling relevance judgements based on content statistics that are inherently local to each collection (*result fusion*). For both purposes, the service may derive and maintain summary descriptions of the target collections (*collection description*). Collectively, the tasks of collection description, collection selection, and result fusion identify the research field of (content-based) *Distributed Information Retrieval* (DIR) [3].

Over the last fifteen years, the DIR field has produced a rich body of techniques to improve the effectiveness and efficiency of distributed retrieval, often under different assumptions on the context of application. Approaches diverge most noticeably in the degree of cooperation that can be expected between the parties that manage the collections and those that distribute searches over them. With cooperation, one can guarantee efficient, timely, and accurate gathering of collection descriptions; similarly, selection and merging algorithms can be made as effective as they would be if the distributed content was centralised. Without cooperation, collection descriptions are approximate and require expensive content sampling and size estimation techniques; as a result, collection selection and fusion are based on heuristics and their performance may fluctuate across queries and sets of collections.

In gCube, we would like to cater for both scenarios, and ideally be able to accommodate any strategy that may be of interest within each scenario. However, DIR strategies may diverge substantially in terms of inputs and in the processes triggered by those inputs. As an example, cooperative fusion strategies expect results to carry content statistics whereas uncooperative strategies may expect only a locally computed score (if they expect anything at all). Similarly, some collection description strategies may consume only local information and resources (e.g. results of past queries); others may require interactions with other services of the infrastructure, and then further diverge as to the services they interact with (e.g. extract content statistics from indices or perform query-based sampling of external search engines).

It is unclear how these strategies could be declaratively specified and submitted to a generic engine for execution. In this and similar contexts within the infrastructure, functional adaptivity seems to call for the deployment of multiple implementations, a set of truly alternative interpretations of DIR functionality. As we would like to preserve uniform discovery and use of the DIR functionality, we also require that the alternative implementations expose the same DIR Master interface.

4 The Design Space

Once we admit multiple implementations of the same interface, we need to decide on the nature of the implementations and on the strategy for their deployment. There are at

least two desiderata in this respect. Firstly, we would like to minimise the cost of developing multiple implementations, ideally to the point that it would not be unreasonable to leverage expertise available within the communities of adoption. Secondly, we would like to add new implementations dynamically, without interrupting service provision in a production infrastructure. The convergence of these two goals would yield an infrastructure that is open to third party enhancements and is thus more sustainable.

4.1 Adaptivity with Multiple Services

The obvious strategy in a service-oriented architecture is to identify multiple implementations with multiple services, e.g. allow many 'concrete' services to implement an 'abstract' DIR Master interface. gCube would standardise the interface and provide one or more concrete services for it; the services would publish a description of their distinguishing features within the infrastructure; clients would dynamically discover and bind to services by specifying the interface and the features they require. The approach is appealing, for gCube already supports the dynamic deployment of services and gCore cuts down the cost of developing new ones.

There are complications, however. Firstly, full-blown service development is overkill when implementations diverge mildly and in part. Many DIR strategies, for example, share the same approach to collection description but diverge in how they use descriptions to select among collections and to merge query results. Server-side libraries can reduce the problem by promoting reuse across service implementations [11]. Yet, service development remains dominated by configuration and building costs that may be unduly replicated.

Secondly, multiple services must be independently staged. This is inefficient when we wish to apply their strategies to the same state and becomes downright problematic if the services can change the state and make its replicas pair-wise inconsistent.

Thirdly, we would like the system to discover and select implementations on behalf of clients. This is because they make no explicit choice, or else because their choice is cast in sufficiently abstract terms that some intelligence within the system can hope to resolve it automatically. In the spirit of autonomicity, the requirement is then for some form of *matchmaking* of implementations within the infrastructure.

In its most general form, matchmaking concerns the dynamic resolution of processing requirements against pools of computational resources. Grid-based infrastructures use it primarily for hardware resources, i.e. to allocate clusters of storage and processors for high-performance and high-throughput applications; in gCube, we employ it routinely to find the best machines for the deployment of given services (cf. Section 2).

Matchmaking of software resources is also possible. There is an active area of research in *Semantic Web Services* that relies upon it to flexibly bind clients to services, most often in the context of service orchestration. The assumption here is that (*i*) clients will associate non-functional, structural, and behavioural requirements with their requests, and (*ii*) distinguished services within the system will resolve their requirements against service descriptions available within the infrastructure [10,15].

In practice, however, the scope of application for service-based matchmaking remains unclear. We would like to use it for arbitrary interactions between services, not only in the context of process execution. We would also like to use it as late as possible,

based on evidence available at call time rather than mostly-static service descriptions. Its impact on the operation of the infrastructure and the design of the system that controls raise also some concerns. The ubiquitous mediation of a remote matchmaker at runtime is likely to increase latencies; similarly, the necessity to distribute its load would complicate considerably its design and model of use. Finally, just-in-time descriptions seem costly to produce and publish, and the expressiveness of these descriptions is likely to introduce novel standardisation costs. Service-based matchmaking remains an interesting option for gCube, but the invasiveness of the approach and the current state of the art does not encourage us to deploy it in a production infrastructure.

4.2 Adaptivity with Multiple Processors

We have explored another strategy for functional adaptivity which promises a smoother integration with the current system. The underlying assumption is different: the diverse implementations of a given functionality are mapped onto multiple components of a single service, rather than onto multiple services. These components are thus local *processors* that compete for the resolution of clients requests on the basis of functional and non-functional requirements.

From a service perspective, the strategy may be summarised as follows (cf. Figure 2). A distinguished *gatekeeper* component receives requests and passes them to a local *matchmaker* as implicit evidence of client requirements. The matchmaker cross-references this evidence with information about the available processors, as well as with any other local information that might bear on the identification of suitable processors. As there might be many such processors for any given request, the matchmaker ranks them all from the most suitable to the least suitable, based on some quantification of its judgement. It then returns the ranking to the gatekeeper. The gatekeeper can then apply any selection and dispatch strategy that is compatible with the semantics of the service: it may dispatch the request to the processor with the highest rank, perhaps only if scored above a certain threshold; it may broadcast it to all processors, it may dispatch it to the first processor that does not fail, and so on.

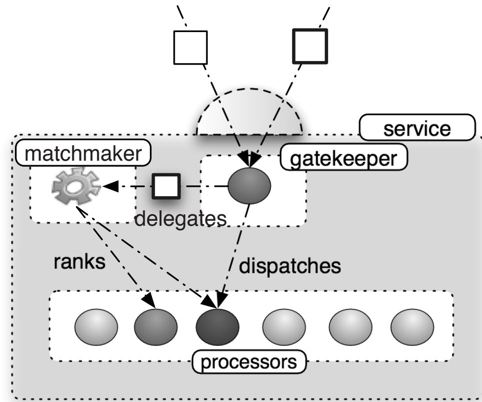

Fig. 2. Matchmaking requests and processors

Placing multiple implementations within the runtime of a single service is unconventional but yields a number of advantages. The definition of processors is framed within a model of local development based on the instantiation of service-specific application frameworks. The model is standard and incremental with respect to overall service behaviour; compared with full-blown service development, it promises lower costs on average, and thus lends itself more easily to third-party adoption. Problems

of staging and state synchronisation across multiple services are also alleviated, as the application of multiple strategies over the same state can occur in the context of a single service. Finally, matchmaking has now a local impact within the infrastructure and can easily exploit evidence available at call-time, starting from the request itself. In particular, it can be introduced on a per-service basis, it does no longer require the definition, movement, and standardisation of service descriptions, and it does not raise latencies or load distribution issues. Overall, the requirement of functional adaptivity is accommodated across individual services, where and when needed; as mentioned in Section 2, the infrastructure becomes more autonomic because its individual services do. Indeed, the strategy goes beyond the purposes of functional adaptivity; the possibility of multi-faceted behaviours enables services to adapt functionally (to requests) but also non-functionally (e.g. to processor's failures, by dispatching to the next processor in the ranking).

As to the matchmaking logic, the design space is virtually unbound. In previous work, we have presented a matchmaker that chooses processors based on the *specificity* with which they can process a given request [14]. In particular, we have considered the common case in which processors specialise the input domains of the service interface, and are themselves organised in inheritance hierarchies. We summarise this approach to matchmaking here and refer to [14] for the technical details. The matchmaker compares the runtime types that annotate the graph structure of request inputs and those that annotate the graph structures of *prototypical* examples of the inputs expected by the available processors. A processor matches a request if the analysis concludes that, at each node, the runtime type of the input is a subtype of the runtime type of the prototype (cf. Figure 3). The analysis is quantified in terms of the distance between matching types along the subtype hierarchy, though the combination of distances is susceptible of various interpretations and so can yield different metrics; in [14], we show a number of metrics, all of which can be injected into the matchmaking logic as configuration of the matchmaker.

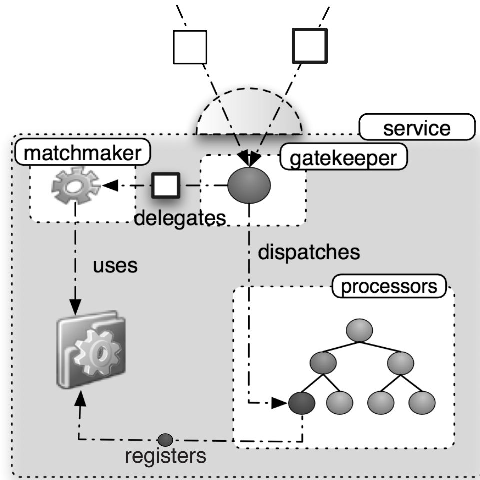

Fig. 3. Matchmaking for hierarchies of processors of increased specificity

5 Service Plugins

Services that are designed to handle multiple processors can be seamlessly extended by injecting new processors *in their runtime*. This challenges another conventional

expectation, that the functionality of the services ought to remain constant after their deployment; not only can services exhibit multiple behaviours, their behaviours can also grow over time. This amplifies the capabilities of both the service — which can serve in multiple application scenarios — and the infrastructure — which maximises resources exploitation.

To achieve this in an optimal manner, we need to look beyond the design boundary of the single service, and consider the role of the infrastructure in supporting the development, publication, discovery, and deployment of *plugins* of service-specific processors. In this Section, we overview the high-level steps of a strategy whereby service plugins become a new kind of *resource* that the infrastructure makes available for the definition of Virtual Research Environments (VREs).

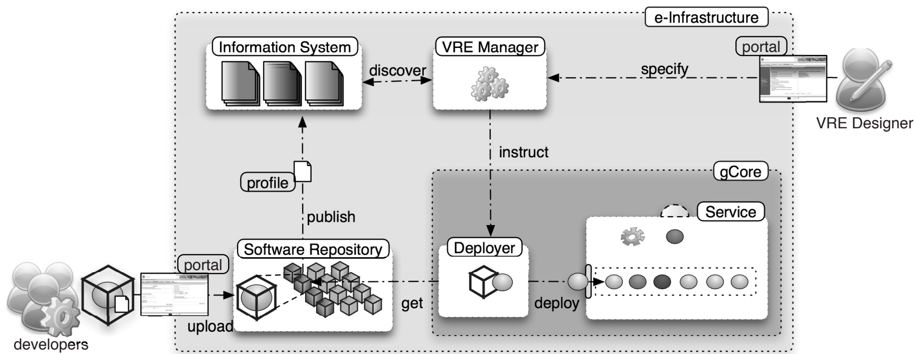

Fig. 4. Plugins publication, discovery and deployment

With reference to Figure 4, the lifecycle of a plugin begins with development and packaging phases. Development is supported by programming abstractions included in gCore and is framed by service-specific libraries (e.g. define service-specific plugin and processor interfaces). Packaging is dictated by infrastructural requirements; plugin code and dependencies are collected in a *software archive* that includes all the metadata needed by the Core Services to manage later phases of the plugin's lifetime. Like for services, in particular, the software archive includes a *profile* that identifies the plugin and its target service, and that describes its logical and physical dependencies.

The plugin enters the infrastructure when it is uploaded to the *Software Repository*, a distinguished Core Service that hosts all the software resources available in the infrastructure. The Software Repository verifies the completeness and correctness of the software archive and then publishes the plugin profile with the *Information Services*, so as to inform the rest of the infrastructure of the availability of the new resource.

The deployment of the plugin may then occurs asynchronously, in the context of the definition or update of a VRE. The process originates in the actions of a distinguished user, the *VRE Designer*, that interacts with a portal to select the content and functionality required for the VRE. gCube translates the human choice into declarative VRE specifications and feds them to dedicated Core Services for the synthesis of a deployment plan. The plan includes the identification of the software, data, and hardware

resources that, collectively, guarantee the delivery of the expected functionality at the expected quality of service [2]. In this context, the plugin is additional functionality to directly present to the VRE Designer or to rely upon when translating the choice of a higher-level functionality into a deployment plan (e.g. the functionality can be implemented by deploying the service augmented with the plugin).

The actuation of the deployment plan is responsibility of another Core Service, the *VRE Manager*. The VRE Manager may inform services that already operate in the scope of existing VREs that their functionality ought to be made available in a new scope. Equally, it may instruct the deployment of new software archives on selected nodes. The instructions are carried out by the *Deployer* service, a Core Service that is distributed with gCore to deploy and undeploy software locally to any node of the infrastructure. The Deployer obtains the archives from the Software Repository and subjects them to a number of tests to ascertain their compatibility with the local environment (i.e. dependency and version management). Upon successful completion of the tests, the Deployer unpackages the archives and deploys their content in the running container.

When a software archive contains a plugin, however, the Deployer performs an additional task and informs the target service of the arrival of additional functionality. In response, the service performs further tests to ensure that the plugin has the expected structure and then stores its profile along with other service configuration, so as to re-establish the existence of the plugin after a restart of the node. If this process of activation is successful, the processors inside the plugin are registered with the matchmaker and become immediately available for the resolution of future requests. The service profile itself is modified to reflect the additional capabilities for the benefit of clients.

Crucially to the viability of the strategy, the behaviour of the service in the exchange protocol with the Deployer is largely pre-defined in gCore. The developer needs to act only upon plugin activation, so as to register the processor the plugin with its matchmaker of choice. Further, the type-based matchmaker summarised in Section 4.2 is also pre-defined in gCore, though service-specific matchmakers defined anew or by customisation of pre-defined ones can be easily injected within the protocol for plugin activation.

6 Conclusions

e-Infrastructures that provide application services for a range of user communities cannot ignore the diversity and specificity of their functional requirements. Yet, the ability to functionally adapt to those requirements, as dynamically and cost-effectively as possible, has received little attention so far.

We have encountered the problem in gCube, a system that enables Virtual Research Environments over infrastructural resources with the functionality and transparencies of conventional Digital Library Management Systems. In this context, adaptivity requires the coexistence of multiple implementations of key DL functionality, the possibility to add new implementations on-demand to a production infrastructure, and enough intelligence to automate the selection of implementations on a per-request basis.

In this paper, we have argued that such requirements can be conveniently met by embedding multiple implementations within individual services, i.e. by making them

polymorphic in their run-time behaviour. In particular, we have shown how mechanisms and tools already available in gCube can be employed to, respectively, reduce the cost of developing such services and enable their extensibility at runtime.

Acknowledgments. This work is partially funded by the European Commission in the context of the D4Science project, under the 1st call of FP7 IST priority, and by a grant from Science Foundation Ireland (SFI).

References

1. Appleton, O., Jones, B., Kranzmuller, D., Laure, E.: The EGEE-II project: Evolution towards a permanent european grid inititative. In: Grandinetti, L. (ed.) High Performance Computing (HPC) and Grids in Action. Advances in Parallel Computing, vol. 16, pp. 424–435. IOS Press, Amsterdam (2008)
2. Assante, M., Candela, L., Castelli, D., Frosini, L., Lelii, L., Manghi, P., Manzi, A., Pagano, P., Simi, M.: An extensible virtual digital libraries generator. In: Christensen-Dalsgaard, B., Castelli, D., Ammitzbøll Jurik, B., Lippincott, J. (eds.) ECDL 2008. LNCS, vol. 5173, pp. 122–134. Springer, Heidelberg (2008)
3. Callan, J.: Distributed Information Retrieval. In: Advances in Information Retrieval, pp. 127–150. Kluwer Academic Publishers, Dordrecht (2000)
4. Candela, L., Akal, F., Avancini, H., Castelli, D., Fusco, L., Guidetti, V., Langguth, C., Manzi, A., Pagano, P., Schuldt, H., Simi, M., Springmann, M., Voicu, L.: Diligent: integrating digital library and grid technologies for a new earth observation research infrastructure. Int. J. Digit. Libr. 7(1), 59–80 (2007)
5. Candela, L., Castelli, D., Ferro, N., Ioannidis, Y., Koutrika, G., Meghini, C., Pagano, P., Ross, S., Soergel, D., Agosti, M., Dobreva, M., Katifori, V., Schuldt, H.: The DELOS Digital Library Reference Model - Foundations for Digital Libraries. In: DELOS: a Network of Excellence on Digital Libraries (February 2008) ISSN 1818-8044 ISBN 2-912335-37-X
6. Candela, L., Castelli, D., Langguth, C., Pagano, P., Schuldt, H., Simi, M., Voicu, L.: On-Demand Service Deployment and Process Support in e-Science DLs: the DILIGENT Experience. In: ECDL Workshop DLSci06: Digital Library Goes e-Science, pp. 37–51 (2006)
7. Castelli, D., Pagano, P.: OpenDLib: A digital library service system. In: Agosti, M., Thanos, C. (eds.) ECDL 2002. LNCS, vol. 2458, pp. 292–308. Springer, Heidelberg (2002)
8. Foster, I., Kesselman, C.: The Grid 2: Blueprint for a new computing infrastructure, 2nd edn. Morgan Kaufmann Publishers, San Francisco (2004)
9. Pagano, P., Simeoni, F., Simi, M., Candela, L.: Taming development complexity in service-oriented e-infrastructures: the gcore application framework and distribution for gcube. Zero-In e-Infrastructure News Magazine 1 (to appear, 2009)
10. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 333–347. Springer, Heidelberg (2002)
11. Simeoni, F., Azzopardi, L., Crestani, F.: An application framework for distributed information retrieval. In: Sugimoto, S., Hunter, J., Rauber, A., Morishima, A. (eds.) ICADL 2006. LNCS, vol. 4312, pp. 192–201. Springer, Heidelberg (2006)
12. Simeoni, F., Bierig, R., Crestani, F.: The DILIGENT Framework for Distributed Information Retrieval. In: Kraaij, W., de Vries, A.P., Clarke, C.L.A., Fuhr, N., Kando, N. (eds.) SIGIR, pp. 781–782. ACM Press, New York (2007)

13. Simeoni, F., Candela, L., Kakaletris, G., Sibeko, M., Pagano, P., Papanikos, G., Polydoras, P., Ioannidis, Y.E., Aarvaag, D., Crestani, F.: A Grid-based Infrastructure for Distributed Retrieval. In: Kovács, L., Fuhr, N., Meghini, C. (eds.) ECDL 2007. LNCS, vol. 4675, pp. 161–173. Springer, Heidelberg (2007)
14. Simeoni, F., Lievens, D.: Matchmaking for Covariant Hierarchies. In: ACP4IS 2009: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software, pp. 13–18. ACM Press, New York (2009)
15. Sycara, K.P.: Dynamic Discovery, Invocation and Composition of Semantic Web Services. In: Vouros, G.A., Panayiotopoulos, T. (eds.) SETN 2004. LNCS (LNAI), vol. 3025, pp. 3–12. Springer, Heidelberg (2004)