An architecture for implementing application interoperation with heterogeneous systems

George Hatzisymeon[1], Nikos Houssos[2], Dimitris Andreadis[3], Vasilis Samoladas[1]

[1]Tech. U. of Crete
{george, vsam}@softnet.tuc.gr
[2] Communication Networks Laboratory, University of Athens
nhoussos@di.uoa.gr
[3] JBoss Europe
dimitris@jboss.org

Abstract. We are concerned with the issues faced by software developers with a certain family of distributed applications; those that connect to and interoperate with a heterogeneous infrastructure, i.e., a large heterogeneous collection of external systems (databases, embedded devices, equipment, internet servers etc.) using different communication protocols. This *product family* includes applications such as e-commerce systems, network management and provisioning applications and Grid-based collaborations. For such applications, implementing the interoperation logic is both challenging and expensive. We discuss the major concerns that contribute to the problem. These include system concerns, such as transaction support, security and management, as well as development concerns, such as integration with workflow or component frameworks. We propose an architecture and related development methodology, based on generative programming, to reduce implementation complexity, allow for rapid application development, ease deployment and manageability.

1 Introduction

In recent years, there is an increasing tendency for automation of complicated distributed processes whose realisation has previously included a significant degree of human intervention. Of particular interest to us are those activities that involve multiple administrative domains, depend on heterogeneous infrastructures and are subject to frequent change. Relevant examples can be found in diverse fields like Business-to-Consumer (B2C) and Business-to-Business (B2B) e-commerce, service management and provisioning in wired and wireless networks, Grid-based collaborations and computer-aided manufacturing, to name but a few. Realising automation essentially amounts to the development of a (typically complex) software application that contains the necessary intelligence (application/business logic).
One of the most challenging aspects of this task is the seamless cooperation of the application logic with a variety of external systems, such as enterprise applications (e.g., ERP, CRM, Billing), databases, Internet/Intranet servers (e.g., web, email, FTP),

and embedded devices (network equipment, sensors, instruments etc.) The development of modules that interoperate with such systems is tedious and time-consuming, since a lot of effort needs to be put on implementing the required communication/access protocols and data transformations.

The aforementioned developer responsibilities are facilitated by tools that enable the programmer to work at a relatively higher level of abstraction. These tools range from simple libraries (e.g., clients for email, FTP or SNMP servers) to powerful middleware technologies (e.g., RMI, CORBA, ODBC, JDBC, COM). However, even with the utilisation of such tools, the task remains challenging, for at least three reasons. First, programmers still need to be aware of a variety of different APIs and technologies, which are irrelevant to the actual task to be implemented. Second, the integration of the external systems into the application frequently requires support for advanced features; dynamic pluggability, transactional execution (when it is possible to undo actions), concurrency control, manageability and configurability. Third, to make external systems available to application logic, they must be made accessible via specialized interfaces: as workflow activities (nodes), web services, component objects (e.g. EJBs or COM components), and so forth. This is achieved via suitable wrappers that can be tedious to compose and maintain by hand.

The present contribution aims to provide a framework for realising interaction with heterogeneous infrastructures that minimises the effort required for the development of the interaction logic. In particular, it defines a component architecture and related mechanisms that provide the following capabilities:

- Rapid development of "one-of-a-kind" components to interoperate with external systems. Development is based on *generative programming* techniques [10] utilizing an active library [27] of access mechanisms/protocols (e.g., telnet, FTP, HTTP, JDBC, SNMP).
- On-the-fly deployment, and integration of components with the underlying transaction, management and security infrastructures of the application.
- Automatic integration of interoperation components into different types of business logic implementations (e.g., workflows, web/grid services) through specially generated wrappers.

To develop our framework, we were guided by the identification of a *product family*, or *domain* (in the sense of [10, 11]), namely that of applications which interoperate with a large, or frequently changing, collection of heterogeneous systems. In this domain, development and maintenance costs of interoperation are comparable, or even dominate, development and maintenance costs of application logic. Our contribution includes a refinement of the semantic content of access mechanisms/protocols (*domain analysis*, in software reuse parlance) and a proposed domain architecture. To validate our approach, an operational prototype has been developed, making use of commercial component frameworks (JBoss [3][4]) and software engineering tools (Eclipse platform [6], Velocity generator [5]). The prototype has been successfully employed to provide application interoperation with relational databases, network elements and Internet servers.

The rest of the current document is organized as follows: Section 2 presents related related work. Section 3 provides an overview of the proposed architecture and elaborates on vital mechanisms such as the task of integrating into an application atomic functional elements and techniques for their template-based, rapid development through predefined adaptors. Section 4 discusses the main choices and trade-offs involved in the design of our solution. The last section of the paper is devoted to summary/conclusions and identification of important elements for further work.

2 Related work

Our work addresses interoperability issues of distributed applications composed by a (possibly dynamic, heterogeneous) collection of external systems. These issues have been addressed before in fundamental work in distributed systems, especially in the area of middleware. The bulk of the work can be cast into two broad approaches: (a) general-purpose, low-level mechanisms, such as basic middleware, and (b) application-specific, high-level techniques.

The first approach, which is typified by traditional middleware (RPC, CORBA, RMI, etc.) has been broadly studied. The general direction of the work is to abstract IPC and networking facilities into a high-level application framework. Recent progress in this area has broadened applicability in challenging cases, such as real-time and embedded applications [13, 17]. Composition of communication protocols has also been studied, notably in the BAST system [16] and in [24, 28]. These techniques are very broadly applicable, but focus on the communication task, and have not been integrated with the higher-level aspects of application frameworks, such as transaction, security and management. The recent introduction of Web Services has advocated a new style of loose integration of autonomous systems, the so-called Service-Oriented Architecture. The platform is currently being augmented with additional conventions related to high-level application aspects (e.g. transactions [8] and resources [12]). It has also been adopted as the standard paradigm for the development of the Grid [13, 14].

The second approach in system interoperability took an application-oriented view of the problem, where the goal was to integrate external systems as close to the application logic as possible. The most notable advances have been in the area of information system integration. The introduction of widely used wrapper technologies (ODBC, JDBC etc.) allowed uniform access to multiple external systems using high-level languages (such as SQL). This has enabled technologies such as mediator-based information system integration [25, 26] over heterogeneous environments, and object-relational mapping technologies (e.g. Enterprise JavaBeans).

What is needed today is the convergence of the two approaches outlined above: general-purpose, high-level system interoperation mechanisms. Ambitious software engineering efforts (notably OMG's Model Driven Architecture [21]) are underway to combine current techniques. At the same time, an array of component-based application frameworks are being developed for web (JSP), client-server applications (J2EE), web and grid services (e.g. Globus [13]), mobile agents (e.g. Cougaar [17]), peer-to-peer systems (e.g. PROST [22]), bringing forward new generations of large-scale distributed systems. In each of these frameworks there is need for high-level interoperability with external systems, integrated with fundamental transactional, security and management mechanisms. Existing technologies to these directions do exist (e.g. the J2EE Connection Architecture [2]), but they are still little more than hooks into the platform functionality.

3 Architecture

In this section we present our framework in considerable detail. First, we focus on the overall system architecture, introducing fundamental concepts and design. Next, emphasis shifts to application lifetime cycle and the issues thereof.

3.1 Overall architecture

Access to external systems is accomplished through *actions*, a semantically high-level interface, whose purpose is to isolate application logic from communication and other access concerns as much as possible. Actions have a signature; their invocations accept and return typed arguments, and raise exceptions. Actions must coordinate: via concurrency control and transactions. They must implement access control, and perhaps obey other security-related constraints. They must be manageable and discoverable. Finally, they must be accessible in a variety of ways: through workflows, embedded script languages, components (e.g., servlets, EJBs), exported as services, and so forth.

Based on these, we concerns suggest a 3-tier architecture to address the problem. The bottom tier encapsulates external system access specifics: communication channels, protocol implementations, session authorization/login, fault tolerance etc. The middle tier's purpose is to integrate with the application framework in use (e.g., J2EE, .NET) and provide synchronized and transactional access to the bottom tier. Finally, the top tier implements different interfaces to the lower tiers, since application logic can be coded in different ways inside the application (workflows, embedded scripting, web service/CORBA/servlet calls, session EJBs etc.). Our proposed architecture along these lines is depicted in Fig. 1.

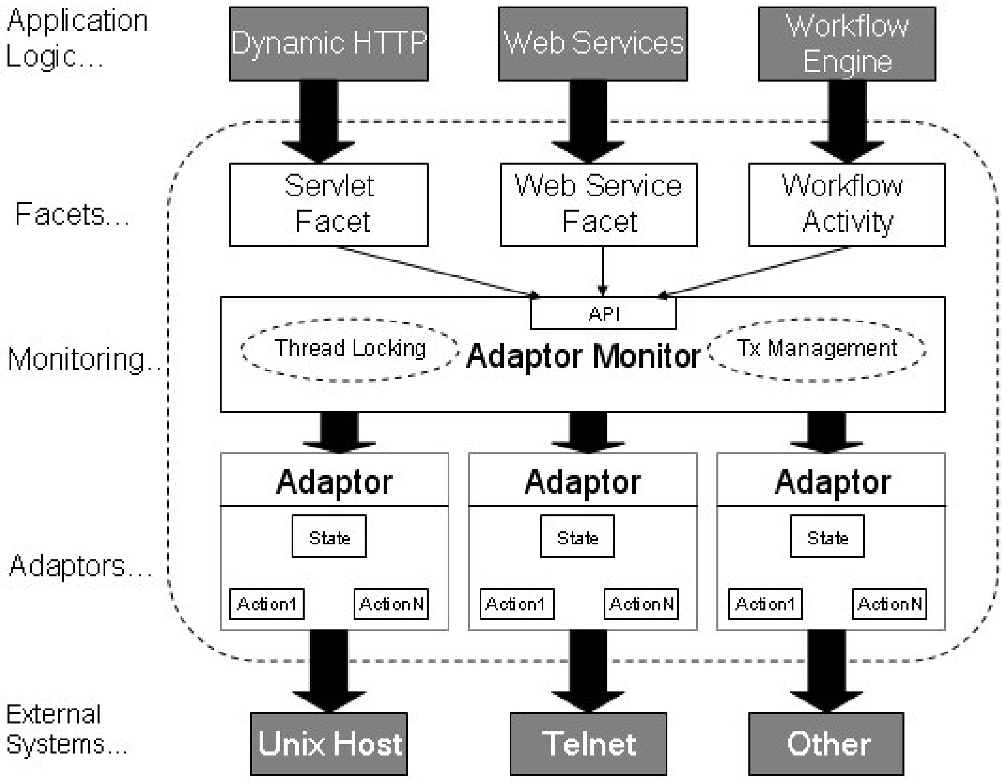

Fig. 1. Making external systems (grey boxes on the bottom) available to application logic (grey boxes on the top): The overall architecture.

Adaptors. The logic for application connectivity to external systems is embedded within *adaptors*. An adaptor is a component, which encapsulates the necessary connection state and logic to one or more external systems. Adaptors possess two distinct interfaces. The first is a transactional, high-level interface, consisting of *actions.* This interface is used by application logic, through the facets (the top layer) interaction with external systems. It integrates with the underlying application logic framework, i.e., transaction processing, concurrency control and authorization. The second is a non-transactional, low-level interface, which is only available to the Adaptor Monitor (the middle layer). This interface is used to integrate the adaptor with the underlying infrastructure, i.e., resource and connection management and monitoring, security, auditing, on-line configuration and lifecycle management.

Adaptors can relate to external systems or services in a variety of ways. For example, an adaptor may encapsulate a telnet-based connection to a remote Unix host, a TCP/IP multicast group, a Kerberos-authenticated database session, an SNMP-managed device, etc. As a general principle, adaptors are protocol-oriented; they derive from protocol templates, specialized and refined appropriately to comply with application requirements.

Actions. Actions correspond to operations on external systems. Each action is contained within a specific adaptor. Actions are stateless components, that is, they only encapsulate the behaviour necessary for an indivisible element of functionality. Their invocations are atomic with respect to application transactions; thus it is desirable that they map to atomic operations on the external system. Each action is specified by one or two procedures, the first procedure implementing the operation, and the second, which is optional, reversing the operation. These two procedures correspond to the well-known DO-UNDO transactional protocol [17].

In contrast to adaptors, which relate closely to the external system, actions relate to the application logic. Consider for example an adaptor encapsulating a telnet session to a Unix host. The adaptor is responsible for communication-level properties, such as IP address and port, session authentication (login/password exchange), configuration of the conversational exchange (e.g., recognising the session's command prompt), etc. Actions related to this particular adaptor however, are totally application specific. For example, if the purpose of connecting to this Unix host is to perform user management on it, sample actions for this adaptor would include `adduser`, `deluser`, `chgpass`, and `chgshell`. The developer would be responsible for implementing these actions (and their reversals) as required by the host, e.g., compose the command line necessary to add/delete a user, and parse the command output. The adaptor will only provide a protocol-specific API (e.g., in our example, an `execute` function, accepting a command line and returning a stream of the command output).

Adaptor Monitor. Actions are invoked only via the *Adaptor Monitor*. This module constitutes the middle layer of our architecture and is responsible for application-wide adaptor integration:

- TP monitor: It is a Transaction Processing monitor for action invocations. It logs the information needed to reverse the sequence of actions performed in the context of an aborted transaction. Naturally, only those actions which support reversal are reversed. This module integrates closely to the application framework.
- Concurrency control: It is implemented by a lock manager, supporting two types of locking:
  - *Synchronization locking*, ensuring mutual exclusion among action invocations from multiple threads. Mutual exclusion can be configured to lock either individual actions or the whole adaptor. These locks are automatically acquired and released upon action invocation and return.
  - *Consistency locking*, where transactions can explicitly obtain long-term locks on specific actions, that will preclude other transactions from invoking these actions, until the locks are released. This mechanism can be used to implement transaction scheduling policies, such as serialization [17].

The lock manager coordinates closely with the TP monitor.

- Directory: It implements a directory service over the deployed adaptors and actions. Apart from name-based discovery it also provides metadata services for adaptors and actions, both in human-readable form (e.g., to be used by interactive management tools), and API-based (i.e., reflection descriptors of adaptor and action interfaces).

- Lifecycle management: The adaptor interface includes four mandatory operations: `init`, `start`, `stop` and `destroy`. These methods are only accessible via the monitor, and typically they are automatically invoked upon particular management operations (e.g., adaptor deployment, redeployment, system exit, system start). During lifecycle operations, the Adaptor Monitor take into account dependencies among adaptors; that is, global sequencing constraints that must be followed in order to set up all adaptors properly. Such dependency information is provided at adaptor design time.
- Adaptor-specific management: is the entry point for invoking additional management operations that are specific to the external system (e.g. SNMP calls).
- Authorization: Actions are not aware of application authorization policy. The necessary checks and rights management are performed here for all action invocations.

Facets. Facets are responsible for application-wide action integration. The Adaptor Monitor has a standard interface for all adaptor and action-related operations, which may not be convenient to call directly from application logic. Some useful types of facets include:

- Workflow facets. Make actions available as *activities* (workflow nodes) to a workflow engine executing in the application.
- Services facets. Actions/sets of actions become available as Web Services, CORBA or RMI objects etc. to the application and its clients.
- Script facets. Actions become available to application-embedded script languages (e.g., Visual Basic, Python).
- Unit testing facets. Interfacing to the testing and debugging tools.
- Servlet, Javabean, and any other component kind, implementing application logic.

Facets are generated automatically from adaptor specifications, using specialized tools for each facet type.

3.2 Implementing Adaptors

Depending on the system they target, adaptors can be very complex components. Their implementation is in most cases knowledge-based (in the terms of the targeting protocol/domain/language). If it were to be attempted by hand, it would likely be time-consuming and difficult to achieve. Instead, we develop a generative approach that allows for rapid, simplified implementation. Our approach is based on the development of an active library [27] of protocols, i.e., a collection of protocol implementation templates, which encapsulate most of the required connection knowledge, and can be customized and refined through a graphical tool.

Our adaptor development process is depicted in Fig. 2. The first stage is adaptor design, and is performed graphically using the Adaptor Designer. It includes three steps: (a) selection of a protocol template, (b) protocol customization, to configure connection, deployment, lifecycle, authentication and auditing aspects, and (c) implementation of the actions required by the application, which includes, for each action, definition of its signature, implementation of the DO-UNDO logic, locking specification and documentation. The result of this process is an (XML-coded) Adaptor Descriptor, which is used to drive code generation.

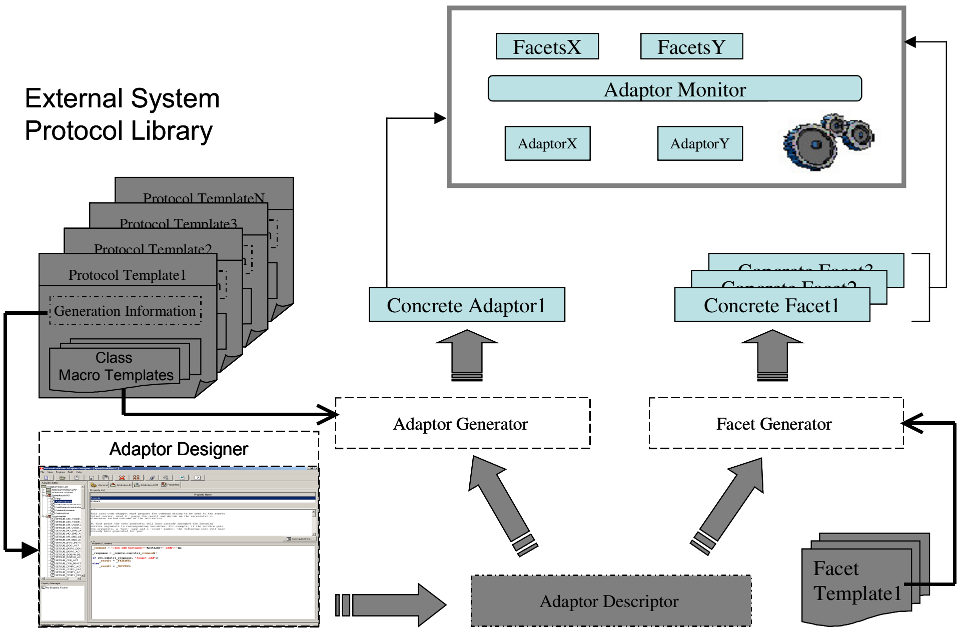

Fig. 2. Adaptor and facet development process.

Each protocol template consists of :

1. A number of class macro-templates, which contain templatized source code to be fed into the generator (we have used the Velocity generator in our implementations), and
2. The Generator Information (GI), an XML-coded descriptor of the protocol template.

The GI is used to customize the Adaptor Designer to the specification needs of the target adaptor. The elements that comprise a GI descriptor are the following:

- *Protocol information:* defines a protocol name, documents the protocol implementation's API and provides deployment information (e.g., external software libraries used)
- *Adaptor properties:* typed attributes exposed to the adaptor API. These can be the mandatory attributes that maintain the state of the adaptor or additional information required to configure protocol-related operation (connection, resource and lifecycle management, authentication etc.) Properties have name, type, an optional default value and a human readable description, presented by the Adaptor Designer to the developers.

- *Action types:* In order to simplify implementation of actions, each adaptor can support a number of action types. Each action type is specified by a name, a collection of class macro-templates and human-readable documentation of the contract that must be fulfilled when implementing actions of this type. The contract of an action type consists a list of mandatory in-out arguments, and a list of guidelines that act as a reference for coding an action's DO-UNDO procedures, and per action action-type. Action types can provide utilities assisting the most common types of interaction processing (text/XML/URL parsers, data transformers, macro expanders etc.)

The output of Adaptor Designer is an **Adaptor Descriptor** (AD), an XML document holding the specification of a concrete adaptor. It is used by the code generators to parameterize the instantiation of code macro-templates. The elements that comprise an AD are as follows:

- *Protocol information:* mostly copied from the GI.
- *Adaptor information:* name and documentation (provided during adaptor design).
- *Adaptor properties:* copy of all properties defined in the GI, with appropriate values provided by the developer. During adaptor design, additional properties can be specified. These can be methods, or attributes. An attribute property has name, a human readable description, type and an initial value. Properties can be used on action method implementations, or can be part of the non-transactional adaptor interface.
- *Action specifications:* an action specification consists of the following: action type, name, action signature, a human-readable documentation of the action interface and semantics, implementations for the DO-UNDO procedures, specification of locking behavior, and deployment information (e.g., dependency on external libraries).
- *Additional metadata:* the adaptor and individual actions can be annotated with arbitrary XML-coded metadata, whose semantics are opaque to our framework. This metadata can be accessed both during facet generation, and at runtime through the Adaptor Monitor.

The final step in adaptor implementation is automated by a code generation tool which accepts the Adaptor Descriptor, and uses the code macro-templates of the protocol template to produce source code, deployment information, scripts etc. Facets are also generatively produced; the adaptor descriptor enters the **facet generator**, together with a list of appropriate class macro-templates, drawn from the active library.

3.3 Prototype implementation

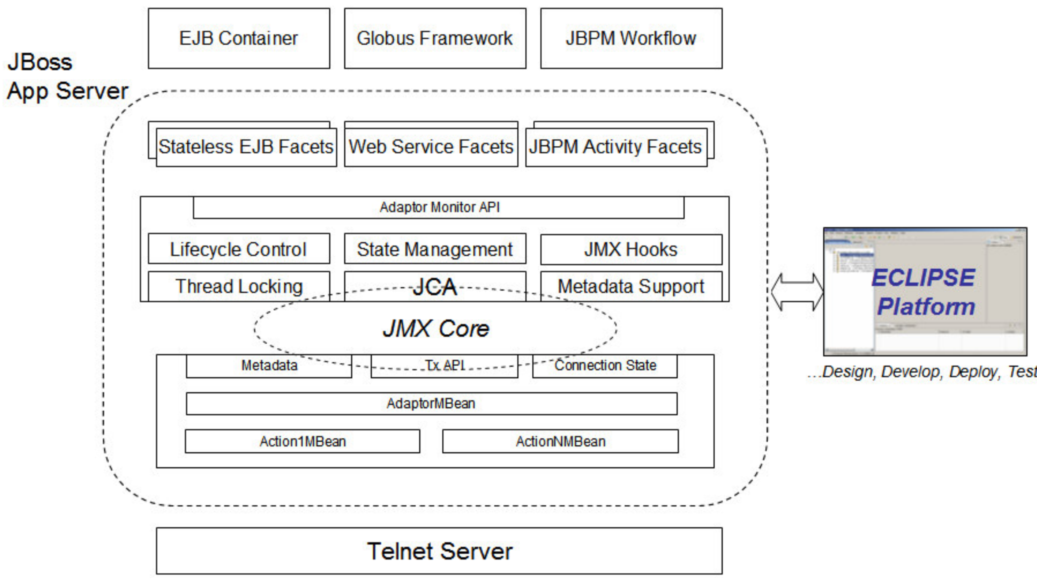

Fig. 3. Architecture of prototype implementation.

For an initial implementation of our platform we chose the Java 2 Enterprise Edition platform (J2EE) and the JBoss application server. JBoss provides robust pluggable implementations of Java Management eXtensions (JMX) [1] and the Java Connector Architecture (JCA) [2] (Fig. 3). Adaptors are implemented as standard MBeans according to the JMX specification. The primary requirement is that they provide an interface (and an implementation of that interface) accessible through JMX.

We have used the Velocity generator to implement both the Adaptor Generator and the Facet Generator. Velocity provides an intuitive macro language that adds only marginal complexity to the coding of protocol templates. Our Adaptor Designer is currently a stand-alone Java application, although we plan to implement a new version inside the Eclipse IDE.

We have implemented a moderate library of protocols, including most Internet services (telnet, FTP, http, SMTP, SNMP), as well as three facet types: an Enterprize Java Bean (EJB) facet, where actions are available to EJBs as methods, a Web Service facet, where actions are exposed as Web Services by JBoss, and a jBPM workflow facet, where actions are available as workflow activities.

4 Discussion

The present section provides a discussion on the architecture proposed in this paper. First, we consider its applicability in two different domains: service provisioning in wired and wireless networks and grid-based applications. Then, we elaborate on important choices and trade-offs that we faced in the course of the system design.

4.1 Application areas

Service provisioning. The proposed architecture is particularly suited for service provisioning applications, in the general area of telecommunications Operation Support Systems (OSS). The goal of seamless service provisioning is to automate the provisioning (and un-provisioning) of telecommunication service across different technology domains (e.g. traditional land line phone service, internet access, mobile access, etc.), thus improving the end-user experience, reduce costs for the operator, and allow the fast introduction of new technologies, products and services [9].

Some of the major challenges of service provisioning and the way they are tackled with the proposed architecture are as follows:

- *The multitude of underlying technologies:* modern providers offer services over a variety of telecommunication equipment and technologies. A typical provisioning scenario may involve interaction with a dozen different devices or management systems. It is of particular importance to be able to encapsulate and control access to those systems using adaptors whose implementation features may change independently from the main provisioning business logic. The usage of generic APIs ensures that to the higher order systems (e.g. workflow logic) invoking on different adaptor looks and feels the same.
- *The problem of consistency:* activation failures are common in complex systems and they can easily result to wasting valuable network resources if a multi-step activation scenario fails at some intermediate point. Our architecture not only ensures rollbacking to the initial state, but it does so in a way that it is friendly to the application developer (i.e. hook into the programmatically available transaction APIs, like JTA), thus considerably reducing the error-prone rollback logic that would have to be coded by hand, otherwise.
- *Constant change.* The strategy of telecommunications operators is not only dictated by technology but by marketing needs, too. Every so often, the marketing department will come up with yet another bundle of services sold as a package, at which point the activation flow will need to be adapted or extended. The proposed architecture matches those requirements because it allows extreme flexibility in introducing new actions, altering implementation of old ones, or just reusing and combine them in arbitrary ways.

Grid computing. Grids [14, 13] constitute virtual computation platforms, promising to make available unparalleled levels of computing, storage and communication resources to scientific, engineering and business applications. To fulfil this promise, Grid technology must be able to harness the resources contributed by the participants of a virtual organization. These resources form a heterogeneous infrastructure, called the *grid fabric*, which must be made accessible to Grid development and application frameworks through a uniform interface, the *Grid middleware*.

Most Grid-related research has been concerned with the grid middleware and higher-level components: resource management, brokering, semantic discovery, etc. There is relatively little work on integrating fabric resources to grid middleware. In real Grids this is done in ad-hoc ways and is considered a source of significant cost. Our proposed architecture can, we claim, reduce this cost significantly, by exposing the Grid fabric to the Grid middleware through adaptors and actions. Thus we can benefit in several ways. Access to computational, storage and network resources, application services and datasets, can automatically integrate with transactional, concurrency, semantic/metadata and security mechanisms of the Grid middleware. Semantic issues are particularly important; most brokering and planning performed by Grid middleware utilizes metadata repositories which provide a semantic description of the grid fabric. Suitable facets can be used to easily populate these repositories with minimum effort.

4.2 Design choices and trade-offs

A principal goal of our solution is achieving separation of concerns with regard to development of interoperation logic (adaptors). This is accomplished through the complementary contribution of three types of actors:

- The framework developer implements basic intelligence common to all adaptors as well as important supporting mechanisms (e.g., generation of custom adaptors from templates). This is actually the functionality introduced in this paper and corresponds to the bulk of the work for interconnection to external systems. Once the framework is deployed and made available for use by the other types of developers, there is practically no need for subsequent modification. Naturally, this does not preclude the (infrequent) updates/new releases of the framework.
- A “connectivity expert” develops specific templates, used to generate a particular family of adaptors. The implementation of these templates is quite tedious and requires specialised knowledge of communication and access protocols (e.g., SMTP, FTP, TELNET, JDBC). It is expected that new adaptor templates are continuously added to the framework, albeit with moderate frequency.
- The application domain expert is responsible only for the application-specific intelligence, that is the implementation of individual actions. This task is normally the easiest and less costly in terms of effort and time. Actions are constantly updated/added to the system, possibly at a high frequency.

The above-mentioned distinction of roles enables new pieces of connectivity logic (at the action or even the adaptor template level) to be easily added to an application, so that the interoperation requirements of the latter are rapidly satisfied.

Another important design choice is the dual interface exported by the adaptors, as elaborated in section 3.1. Actions comprise the high-level portion of the adaptor interface, supporting advanced features like transactions, concurrency and authorisation. The rest of the interface includes operations that are too low-level for the application to be aware of; they are therefore available only to the Adaptor Monitor and pertain to functions like resource monitoring and lifecycle management. Features like transactionality and concurrency are not supported for these operations; this would considerably complicate matters without any significant benefit (since such operations are not exposed to applications). There is ample precedent justifying our choice, e.g., an analogous design feature exists in database systems, where the

Data Manipulation Language is transactional, while the Data Definition Language is not.

The ultimate objective of the framework is to enable application logic to invoke actions. An action encompasses only the logic that needs to be executed at the external resource; it does not care how the connectivity is obtained. Furthermore, actions are atomic; they do not encapsulate any further nested actions that can be handled as distinct functions from a transactional point of view. Thus, they need not maintain any state information. Support for transactional behavior is optional for actions. Actions are therefore extremely lightweight components; the simpler among them may consist of only a few lines of code. This leads to:

- Minimal effort and time required from the part of the action developer.
- Minimal overhead for the execution of actions. In case of non-transactional actions, the overhead is further smaller, since even the adaptor preserves no relevant history. Importantly, a particular action can be subject to a large number of invocations by a multiplicity of different applications, so this characteristic can be important for maintaining high performance levels.

A conscious choice we made was not to support a refinement hierarchy for adaptors (i.e., support class-like “inheritance”). We do not view actions as “public methods” of adaptors. This choice greatly simplified our adoption of generative programming, and what benefits we forfeit, we can obtain in different ways. A notable capability provided by the framework is the case of adaptors that do not contain any actions; instead they are used as services by other adaptors, through their non-transactional interface. This “service oriented architecture” of adaptor collections can support the rapid development of complex interactions among external systems, and thus achieve composite overall behaviour. As an example, consider composing a new authentication mechanism to an existing protocol. To access an external system, one would create two adaptors for this system: an action-less one to handle authentication interactions and a second to handle the subsequent session interactions via actions.

In designing the transaction support for the Adaptor Monitor, we chose to select DO-UNDO semantics, instead of the more powerful DO-UNDO-REDO semantics. Thus, it will be difficult to implement advanced buffering/caching/coalescing behaviour at the action level. This choice limits performance in a few cases. For example, implementing object-relational mapping of an external data source through our framework may be less efficient. On the other hand, we gain in simplicity: for most external systems, the meaning of REDO is not obvious in general. A related concern is our choice of locking semantics. We chose not to constrain the user to a specific protocol (e.g, an obvious choice would be two-phase locking) but instead allow application logic to control locking explicitly. If more constrained behaviour is desirable for some adaptors, it can in principle be supported by special facets.

A concern we faced during the design of the overall architecture is the management of events that originate from the underlying infrastructure and are of interest to the application. Relevant issues have been the subjects of extensive research efforts in areas related to distributed systems [29][30]. The approach adopted by our framework so far does not include an explicit mechanism for event propagation towards the application. However, this can be achieved: A polling paradigm may be employed by the application logic for receiving important information from external systems, through suitable actions. The corresponding events can then be generated, subsequently triggering the appropriate behaviour. Apparently, according to this model, an event loop is implemented within applications. Future work will focus on incorporating the support for event management into the framework; this can be useful especially in the case of loose coupling between the application and external resources. A relevant non-trivial question that can be identified pertains to achieving the seamless cooperation of actions with publish-subscribe mechanisms.

5 Summary – future work

In this paper, we have presented an architecture for implementing application interoperation with heterogeneous infrastructures. Our contribution mostly pertains to software systems, which have extensive and frequently changing requirements for connection to external resources. We have introduced a framework for making such resources available to several types of applications. Moreover, we have described a basic adaptor structure for encapsulating all aspects of an external entity and a generative technique for rapidly introducing adaptors into an application. Our architecture promotes separation of concerns in the development of interconnection functionality, with a bias in the direction of reducing the burden on the developer of application logic.

With regard to future work, the top priorities that can be identified are the following:

- Definition of an event management mechanism to be incorporated into the framework,
- Utilisation of the framework to interconnect to external resources diverse types of applications as JBoss-based workflows and grid applications (especially on the Globus platform).
- Investigation of the architecture implementation based on platforms other than J2EE, such as the Cougaar agent framework.

References

1. Java Management Extensions White Paper: Dynamic Management for the Service Age. http://java.sun.com/products/JavaManagement, 1999.
2. J2EE Connector Architecture Specification, Version 1.5, Nov. 2003.
3. JBoss Open Source Application Server, http://www.jboss.org.
4. M Fleury, F Reverbel, The JBoss Extensible Server, International Middleware Conference (Middleware 2003), Brazil, June 2003.
5. Velocity Template Engine, http://jakarta.apache.org/velocity.
6. Eclipse Integrated Development Environment, http:// www.eclipse.org.
7. A. Beugnard, Communication Services as Components for Telecommunication Applications, In Proc. Objects and Patterns in Telecom Workshop (in ECOOP'00), 2000.
8. L. F. Cabrera, G. Copeland, M. Fwingold et al. Web Services Atomic Transaction (WS-AtomicTransaction), Nov 2004.

9. A. Clemm, F. Shen and V. Lee, Generic Provisioning of Heterogeneous Services—a Close Encounter with Service Profiles. Computer Networks 43 (2003), 43-57.
10. K. Czarnecki and U. W. Eisenecker, Components and Generative Programming. In Proc. 7th European Software Eng. Conf. (held jointly with 7th SIGSOFT Int'l Symp. on Foundations of Software Eng.), 1998.
11. A. Egyed, N. Mehta and N. Medvidovic, Software Connectors and Refinement in Family Architectures. In Proc. 3rd Int'l Workshop on Development and Evolution of Software Architectures for Product Families (ARES III), LNCS 1951, 96-105, 2000.
12. I. Foster, J. Frey, S. Graham et al. Modelling Stateful Resources with Web Services. Preliminary whitepaper version 1.1, 3/5/2004.
13. I. Foster, C. Kesselman, J. Nick, S. Tuecke. Grid Services for Distributed System Integration. Computer, 35(6), 2002.
14. I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3), 2001.
15. A. Gokhale and D. C. Schmidt, Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems. In Proc. of INFOCOM '99, 1999.
16. B. Garbinato and R. Guerraoui, Flexible Protocol Composition in Bast, In Proc. Int'l Conf. on Distributed Computing Systems (ICDCS), 1998.
17. J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques. 1993, Morgan Kaufmann Pub.
18. A. Helsinger, A. Thome and T. Wright. Cougaar: A Scalable, Distributed Muti-Agent Architecture. In Proc. IEEE Conf. on Systems, Man and Cybernetics (SMC), 2004.
19. A. Krishna, D. C. Schmidt and R. Klefstad, Enhancing Real-Time CORBA via Real-Time Java. In Proc. 24th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS), 2004.
20. N. Mehta, N. Medvidovic and S. Phadke, Towards a Taxonomy of Software Connectors. In Proc. Int'l Conf. on Software Engineering, 178-187, 2000.
21. J. Miller and J. Mukerji (Eds.), Model Driven Architecture (MDA). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 2001.
22. M. Portmann, S. Ardon, P. Senac, A. Seneviratne, PROST: A Programmable Structured Peer-to-peer Overlay Network, In Proc. IEEE Int'l Conf. on Peer-to-peer Computing (P2P), 2004.
23. Y. Smaragdakis and D. Batory, Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. Software Engineering and Methodology 11(2), 215-255, 2002.
24. B. Spitznagel and D. Garlan, A Compositional Approach for Constructing Connectors. In Proc. Working IEEE/IFIP Conf. on Software Architecture (WISCA), 2001.
25. S. Thakkar, C. A. Knoblock and J. L. Ambite, A View Integration Approach to Dynamic Composition of Web Services. In Proc. ICAPS Workshop on Planning for Web Services. 2003.
26. S. Thakkar, J. L. Ambite and C. A. Knoblock, A data Integration Approach to Automatically Composing and Optimizing Web Services. In Proc. ICAPS Workshop on Planning for Web Services. 2004.

27. T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In Proc. SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), 1998.
28. D. M. Yellin and R. E. Strom, Interfaces, Protocols and the Semi-Automatic Construction of Software Adaptors. In Proc. Object-Oriented Programming, Systems, Languages and Architectures (OOPSLA), 176-190, 1994.
29. J. Bacon et al., Generic support for distributed applications, IEEE Computer, March 2000.
30. R. Meier, Taxonomy of Distributed Event-Based Programming Systems, 1st International Workshop on Event-Based Systems (DEBS 2002), July 2002, Vienna, Austria.