A Lightweight, Message-Oriented Application Server for the WWW

Ralf-Dieter Schimkat*
Wilhelm-Schickard Institute for Computer Science
Sand 13
D-72076 Tübingen, Germany
schimkat@informatik.uni-tuebingen.de

Stefan Müller
Wilhelm-Schickard Institute for Computer Science
Sand 13
D-72076 Tübingen, Germany
muellers@informatik.uni-tuebingen.de

Wolfgang Küchlin
Wilhelm-Schickard Institute for Computer Science
Sand 13
D-72076 Tübingen, Germany
kuechlin@informatik.uni-tuebingen.de

ABSTRACT

In this paper, we present a lightweight system that loosely couples and integrates any kind of source systems with information retrieval capabilities. The system provides mechanisms for a rapid integration of source systems into the World Wide Web (WWW), allowing the generation of configurable collections of individual, heterogeneous source systems.
The proposed system is based on a lightweight, message-oriented application server and an object-oriented client framework to provide an uniform, Java-based graphical user interface. The system offers application middleware functionalities that solve issues such as limited bandwidth and scalability of both sides (WWW user clients and backend server systems) in a generic manner.
Already integrated source systems include the system PROGRESS (a method base system for mathematical algorithms with functionalities for server-side computing) and SPECTO (an XML-based, distributed monitoring system).

Keywords

Application server, uniform Web interface, application integration, message bus.

1. INTRODUCTION

During the last five years, a lot of different database and information systems have been developed with an intention to offer their functionalities to a wide range of users via the Internet and especially via the World Wide Web (WWW). Many more information systems do not even provide a Web interface. Efforts are made in the field of Federated Information Systems (FIS)[15] and by Commercial Application Servers (CAS) to integrate those heterogeneous systems.

*Supported by debis Systemhaus Industry.

SAC'00 March 19-21 Como, Italy

FIS accomplish this by a federated model or scheme on top of the systems to be integrated. Our approach differs from the one used in FIS in the way that we do not integrate semantically the backend systems. Instead, we introduce a general data description model which is only responsible for the presentation of data, whereas the semantics is determined by the individual backend system (BES). The data description model is part of a middleware component that encapsulates all communication aspects and therefore can be seen as a means to hide the technical heterogeneity of the BES. Virtually any kind of source system with information retrieval capabilities or database management system can be considered a BES. Our approach should yield a lightweight information system (explained in Section 2.2) with which any BES can be rapidly adapted to our system. After the adaption, each BES remains an autonomous system but additionally offers its functionality to a wide range of Web users through an uniform Web interface. Logical changes to the BES do not affect our middleware because of their autonomity. On the other hand, the missing integrating logical view prevents a direct reuse of data or any combination of the information provided by the BES. We are therefore not concerned with the development of a meta model that describes an integrated view of the incorporated systems. Our "meta model" just determines the messages that must be delivered between the client (the uniform Web interface) and the individual BES.

In contrast to CAS, our system intends to offer a facility for forming collections of individual heterogeneous systems with the help of integration mechanisms. A collection (i.e. each individual system within the collection) can be accessed via a single uniform Web interface because the structure of the transferred messages is general and different messages can be mapped onto the same view, like different XML-documents [3] can be displayed within a single viewer. The communication of most CAS is based on concepts like CORBA [11] or Java's [6] Remote Method Invocation (RMI) with compiled stubs on the client and server side which does not provide an uniform access or view because each stub must be handled individually. CAS provide uniform interfaces only for the management of the integrated systems. Similiar to CAS, our system architecture allows to solve communication problems like scalability and bandwidth. Additionally our system solves the problem of uniform access to the individual BES.

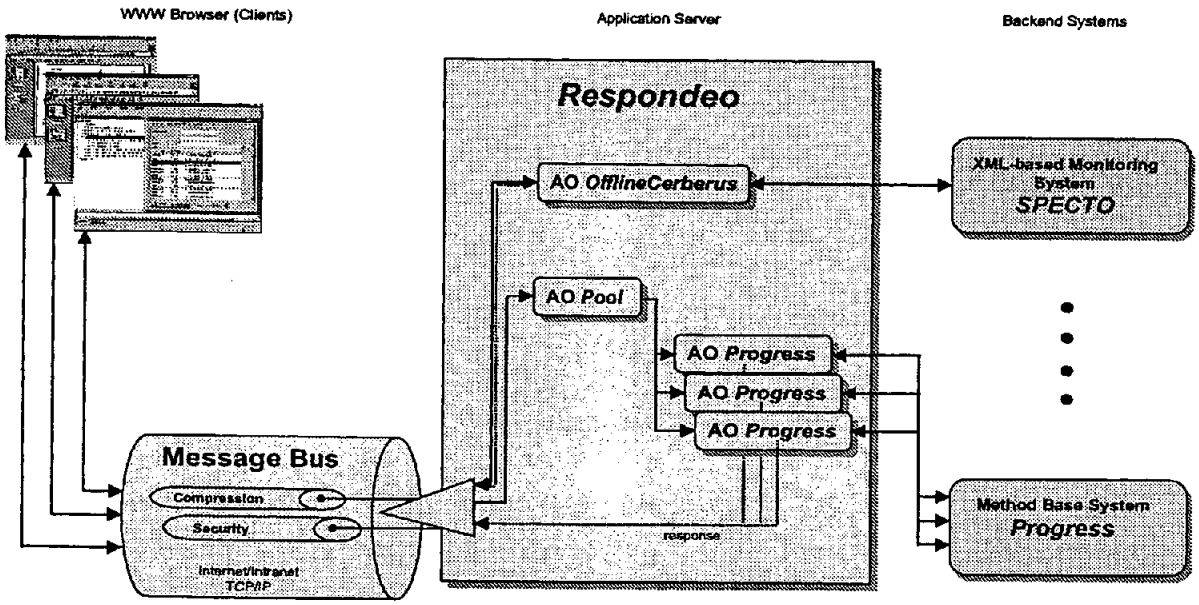

Figure 1: Architecture of the application middleware

All these problems arise in an environment where a large number of users want to interact with a large number of different information servers, which is the typical situation on the Internet and the WWW. All the problems of an uniform interface, of scalability and of bandwidth are explained and discussed, together with the design and implementation of our system, in the next two sections.

Section 2 describes the architecture of our lightweight, message-oriented system. Section 3 explains how the different communication problems mentioned above are solved by the system, and Section 4 shows the integration of a scientific method base system (PROGRESS) and an XML-based monitoring system (SPECTO) as examples of how users can interact with our system. The article concludes with a short summary.

2. PROPOSED ARCHITECTURE

In this section, we discuss the main components of the object-oriented framework architecture for the proposed system. A framework is a reusable, semi-complete application that can be specialized to produce custom applications [4]. It is characterized by an abstract set of classes and interfaces which, taken together, establish a generic software architecture for a family of related applications [8; 9]. An interface defines an object-oriented class where all methods are abstract. Thus it supports a clear distinction between the specification of functionalities and their actual implementations.

Mainly due to the scalability issues (a large and unknown number of clients and BES) we propose a strictly decoupled multi-tier client-server architecture, as illustrated in Figure 1. A sophisticated lightweight, multithreaded application server in the middle tier loosely couples WWW clients and BES in the backend tier. Each tier is encapsulated into an uniform object-oriented interface which guarantees stable interfaces over time even for future BES which might be integrated into the system. Client and application server are connected via a message bus which serves as a standardized point-to-point communication channel within the WWW. Each client request is wrapped in a message and forwarded to the application server, which in turn is responsible for finding the appropriate application object (AO) to handle the incoming request. Each BES is encapsulated within one or a pool of dedicated application objects. Generally, aspects such as communication protocols and distribution of objects are completely hidden behind uniform framework interfaces. Thus, while guaranteeing application services at the framework level, all instances of the framework, namely all BES which are integrated into the framework, benefit from the overall infrastructure in an uniform and predictable way. Therefore, the proposed system can be seen as an application middleware for connecting BES to the WWW. The integrated BES form a loosely coupled information system.

We have adopted an object-oriented model, which is implemented in Java. However, legacy BES components do not have to be written in Java, as long as there are interfaces to glue Java and the programming language in question together, e.g. C/C++ via Java Native Interface.

2.1 The Message Bus

Our architecture includes a messaging component that serves as an uniformly accessible communication channel between each client and the application server. The proposed message bus is characterized by point-to-point communication between the clients and the application server based on the request/response model. The data delivery protocol can be synchronous or asynchronous depending on the application's profile. Each client request is wrapped in a message and transmitted to the application server. According to certain meta entries of the message, the application server dispatches the incoming message to the appropriate application object regardless of the actual content of the message.

Each message is composed of a header, a set of properties,

and a body. The header contains values used by both the clients and the application server to describe the structure and the content of messages in a general manner. In addition, each message provides a set of optional and extensible properties which are used by BES to specify dedicated application protocols by adding application-specific fields to a message. The body contains the data items themselves, which can be of any type that is serializable in Java.
By using messages as the exclusive communication paradigm between WWW clients and the application server, each call to remote functionalities is dispatched to the appropriate method call of the application server. Since the communication interfaces between clients and the application server do not change over time regardless of the number and type of BES which are or might be integrated in the system, no additional communication-related code has to be provided, such as compiled static interfaces like stubs and skeletons. Instead, dedicated application protocols wrapped in messages are used as generic interfaces to initiate remote requests and to transmit the corresponding response back from the application server.

2.2 RESPONDEO - A Lightweight Application Server

RESPONDEO is a lightweight application server for managing BES wrapped in application objects. It provides an object-oriented framework for specialized communication and application protocols which facilitates the connection of various legacy BES to the WWW. RESPONDEO is lightweight in the sense that it is small in code size, it offers well-defined interfaces for interaction and communication via the message bus, and it provides stateless application objects only. It manages application objects in a generic way: it does not need to know anything about the application-specific behaviour and implementation of each BES. The abstraction from these BES-specific details offers management functionalities at a higher level of abstraction and in an uniformly accessible manner. Within the proposed system architecture, RESPONDEO forms the object-oriented framework-based middle tier which is implemented in Java JDK 2.

2.2.1 *Application Objects*

An application object encapsulates all application-specific logic for initialization and for handling incoming requests. Since application objects are stateless, the number of available application objects for processing client requests can be restricted to a predefined maximum number.
An application object can be configured by attribute values which can be defined in a special property file called *ApplicationObjects.properties*. In the upper part of Figure 2, an extract of the file with two application objects is depicted. In the lower part of Figure 2, the hierarchical structure of the property file is illustrated. Basically there are attribute values related to RESPONDEO and to application objects in general. The second parameter (*class name*) is the class name of an application object which is loaded dynamically into RESPONDEO during the process of initialization. The rest of the parameters are application-specific name-value pairs which are passed through to the specified application object as a black-box. This strategy can be applied iteratively as shown for the application object *Pool*. A pool is a special application object managing a set of application objects of the same type. It has to have two parameters called *member* and *poolsize*, which respectively specify the type of the application object and the number of application objects in the pool.
Each parameter is accessed by a general function which provides a stable and unique interface to all application objects. The actual management of application-specific parameters is left up to the application object's implementation. In addition, the ability to obtain parameter information via a general function enables the development of flexible collections of application objects that can be changed and adapted even at run-time by applying the initialization mechanism as described above.
RESPONDEO provides a naming service that maps a symbolic name to a set of application objects. The symbolic name, shown in Figure 2 as the first property value (*Debis-Cerberus1*, *RandomProgressAlgorithms*) of each application object, provides independent ways to bind to an application object, i.e. for availability, improved performance and for reusing existing application objects in various application contexts. The symbolic name can change over time and is accessed via the general function mentioned above. In Section 4 examples of application objects are illustrated.

2.2.2 *Application Management Functionalities*

RESPONDEO offers several management functionalities to each application object regardless of the actual type of BES. For each BES, it manages a single instance or pool of instances of the corresponding application object. When an incoming request has been processed, the active application object is enqueued into the provided pool for future client requests. If there is no application object available for processing an incoming request, the request is blocked until the application object pool can offer the object. The size of the pool can be configured and changed at run-time in order to adapt the scheduled number of application objects to the currently active number of WWW users and the processor's load.
By controlling the entire communication process between client and backend servers through an additional level of indirection, RESPONDEO shields the backend servers from the total load. By providing only stateless objects, all resources (number of application objects for each BES) are configurable, controllable and predictable regardless of the number of WWW clients trying to access the BES.
RESPONDEO strictly separates application- and communication-related logic by wrapping application objects and client requests in standardized messages. Thus, the communication level of the proposed architecture has to deal with messages only regardless of their application specific content, i.e. the application logic. Our architecture provides various kinds of communication channels which can be switched dynamically at run-time, e.g. the compression and the security channel. In the former one, all messages are compressed in order to reduce bandwidth. The latter one provides a secure data transfer on the message bus between clients and RESPONDEO. As far as RESPONDEO is concerned, communication channels can be aggregated and plugged together in order to build customized channels which communicate down a single network connection.

2.3 The Client

In our architecture, the main component of the client tier is an object-oriented framework which provides both an uniform graphical user interface (GUI) for various applications

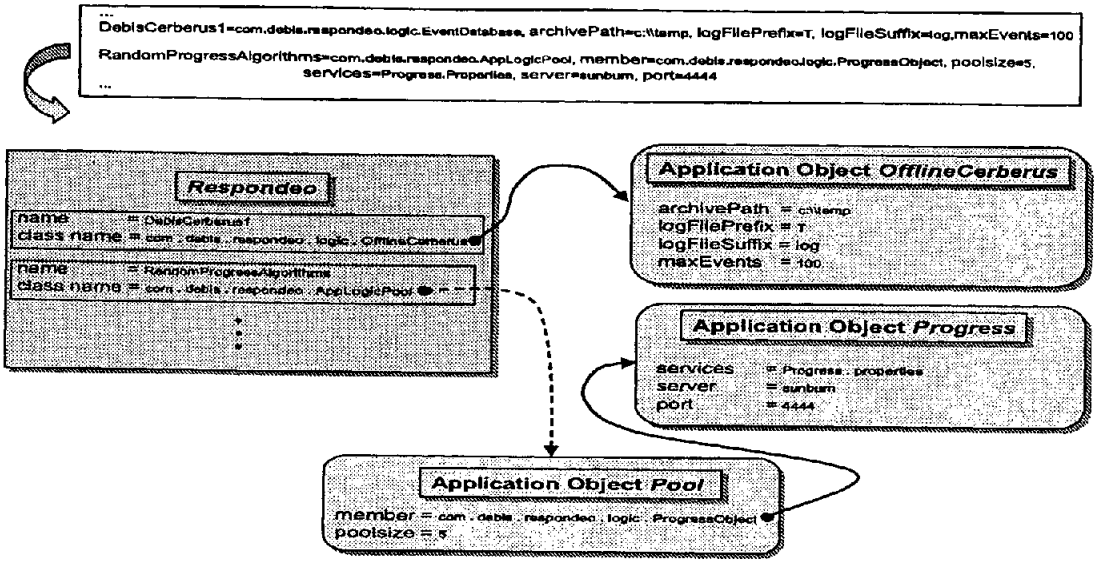

Figure 2: Initialization process of application objects

and a message-based communication component. The framework is based on JIMA which is presented in [13]. In order to access a BES from the client framework, the corresponding application-specific logic has to be provided by subclassing so-called hotspots [12] of the framework. Each data entry of a BES is described by a so-called general *InputComponent*. This component does not provide a common data model. Instead it describes a data entry of the BES by three attributes. The attribute *name* assigns a name to a data entry, the attribute *type* describes the type of a data entry (e.g. Integer, List, Picture, ...), and the attribute *value* is a link to the actual data. As far as the GUI is concerned, it only needs to access the entries in an uniform manner, whereas application-specific logic on the client-side is responsible for interpreting the respective *InputComponents*. The visualization of the data depends on the value of the type attribute. This means that the visual representation is identical for a given type independent of the application-specific representation. Therefore the user interface is uniform with respect to a given type. In Section *4.1.1* an example of an *InputComponent* is given. Generally, the client framework offers facilities to build user-specific collections of BES at the client-side. A collection is basically a group of BES which provides an uniform WWW interface to the entire bundle of BES. The client framework can be used transparently as an applet via the Internet or as a common application which is installed locally on a computer.

The GUI separates presentation and interaction logic in distinct layers at the framework level. The presentation layer provides uniformly accessible interfaces for the visualization of tables, lists, text components, menus, etc. The interaction layer notifies the application logic of changes initiated by the user. However, both layers are coupled loosely by the command pattern [5] which provides a general callback facility within the framework. The GUI-related framework components are largely reused by each BES.

The communication component is a message-based manager for handling client requests. There is one single instance of it for all integrated BES. This client-based manager forms the communication end-point to RESPONDEO, which resides in the middle tier. Each request that is initiated by the GUI component is wrapped in a message and transferred to the communication manager. According to the execution strategy of the BES, the message is sent to RESPONDEO synchronously or asynchronously. When transmitting messages asynchronously, a callback handler has to be provided in order to handle the results appropriately. Additionally, the communication manager is responsible for choosing the right communication strategy, such as the compression or security channel described above.

3. SUPPORT SERVICES

In this section, we discuss how the goals of scalability, reduced bandwidth, and a sophisticated user interface are achieved by the system introduced in Section 2. Generally, our application middleware supports application services which provide solutions to the described goals at the design level as opposed to the implementation level. Therefore, these services apply to an entire set of related BES which do not provide functionalities and accessibility features for the Web themselves.

3.1 Scalability

The central issue of scalability is the system's ability to support both growing numbers of clients and backend servers. Below we will focus on some important design considerations which support scalability in different contexts.

3.1.1 WWW User Scalability

The number of WWW users connected concurrently to our information system is not predictable and varies from time to time. In addition, the collections of integrated BES on the client-side should be customizable by each user. We therefore propose a multi-tier client-server architecture with the standardized message bus between client and application server, as described in Section 2. By decoupling the clients from the backend servers, the application server in

the middle tier manages the overall workload in a controlled manner:

- The size of each pool of application objects is determined by a configurable value. If the number of active requests exceeds the number of application objects in a pool, a client request is blocked until one of the stateless application objects is ready for processing the request. This allocation strategy guarantees that there is a well-defined number of concurrent application objects which in turn determine the maximum workload the BES are exposed to. The upper bound is independent of the overall number of concurrent WWW users. In addition, the statelessness of application objects results in a constant number of allocated objects in memory regardless of the actual workload.

- Since the presence of only one application server is a potential bottleneck within the system, we have introduced the concept of interconnected groups of application servers to our middleware. Thereby, one dedicated instance of the group serves primarily as a message router, in order to balance the load among the remaining group of application servers appropriately. Since such routing objects are specialized application objects which forward messages to specified locations, the entire set of application management functionalities apply to them. Various sophisticated routing strategies can be implemented in order to fulfill potential load-balancing requirements because application objects can be configured at run-time.

The message bus provides an uniformly accessible communication channel which is completely independent from any specific collection of BES representation at the client-side. Therefore, the interaction of multiple applications through dedicated application protocols which are wrapped in messages, makes no assumptions about the communication-related code of applications apart from the common message bus interface. This scalable approach enables the access to any collection of BES representations at the client-side.

3.1.2 Backend Server Scalability

The scalability issues related to the potentially huge number of different BES, such as relational, object-relational, or XML-based database management systems, are critical to our application middleware. The point to note here is that the application server provides the right place to encapsulate different kinds of BES and to give them an uniformly accessible and manageable interface. Thus, our middleware solves the application-related scalability issues at a higher level of abstraction:

- Each BES is encapsulated within general application objects which hide the different kinds of BES and their implementations respectively.

- For each application object, a pool can be configured which is managed by the general application server. A pool basically synchronizes the access to and manages the pooling of application objects, i.e. connection pooling for fast and efficient handling of database connections.

- Various configuration policies can be applied even at run-time to each pool of application objects in an uniform manner since the architecture strictly separates between the management of application objects and their implementations, as discussed in [14].

- Our naming scheme enables a scalable approach to reuse existing application objects in different application contexts by providing different symbolic names to the application objects.

In general, solving scalability issues at a higher level of abstraction leads to an exhaustive reuse of the provided middleware services and speeds up the overall integration process tremendously.

3.2 Bandwidth

The issue of limited bandwidth concerns network applications which assume that the reliability and capacity of the underlying network is not sufficient. Thus, one goal is to keep the size of the transmitted data as small as possible by providing adequate data encoding schemes. When applying sophisticated data compression schemes, the availability of local resources for the encoding and decoding of the data has to be kept in mind. In general, since there is a tradeoff between managing bandwidth and the availability of local resources, a flexible strategy has to be provided which can be changed at run-time. For example, when the volume of the data is large and local decoding resources are available, apply adequate encoding mechanisms. Especially on the Web, the additional time for the decoding of the transmitted data has only little affect on the overall communication time. When the volume of the data is small, transmit the data without any additional encoding. This is also the default strategy.

Generally, scalability affects the limited bandwidth problem directly due to the increased volume of data, since an increasing number of clients demands an increasing throughput of method or object calls. The application middleware provides a solution to these problems by encapsulating the entire communication process and data transfer within the uniformly accessible message bus which offers a compressing, a secure and a default communication channel for transmitting messages. The strategy for choosing the appropriate communication channel is changeable at run-time.

3.3 User Interface

Traditional HTML-based Web interfaces are simple and tailored to the requirements of a certain BES. Since the communication paradigm is based on the stateless protocol HTTP, this kind of user interface supports only basic interaction facilities between client and server. Additionally, it lacks sophisticated integration mechanisms due to the common use of proprietary communication protocols and of application-specific look and feel.

Our client-based framework overcomes these shortcomings by providing uniformity along two dimensions:

- Uniform look and feel of the user interface by encapsulating visualization components within an object-oriented framework implemented in Java. Each collection of BES representation at the client-side can reuse these framework components.

- Uniform accessibility of the user interface by providing a client-based message manager which handles different application protocols wrapped in messages.

4. EXAMPLES

As mentioned above, a wide variety of different BES can be integrated rapidly into the system enabling an uniform usage of their services via the WWW. As examples of how to integrate BES and their functionalities, two different kinds of BES, namely the method base system PROGRESS and an XML-based monitoring system, will show the potential use of our middleware.

4.1 The Integration of the Method Base System PROGRESS

The method base system PROGRESS [1; 2] facilitates the provision and usage of computational services via the Internet. The basic idea behind PROGRESS is to build an environment that renders the combination of "implementation languages" like C or C++ with tools for modeling and Internet support. PROGRESS provides a simple untyped scripting language. It facilitates the definition and processing of complex and nested structures (list, tuple, etc.). Algorithms (e.g. sorting algorithms) implemented in a host language (e.g. C) can be integrated with a small wrapper that transforms the input and output parameters into the PROGRESS language. The PROGRESS server (basically a PROGRESS language interpreter) connects the system with the Internet so that the integrated algorithms can be used remotely via a PROGRESS shell or a special Web interface. PROGRESS also provides a Java package with which PROGRESS language constructs can be modelled, and by which a Java application can communicate transparently with a PROGRESS server [7]. This package was created to build more powerful user interfaces and in our context is the basis for the integration of a PROGRESS server into our system.

4.1.1 *The* PROGRESS *Application Object and the* PROGRESS *Input Component*

In order to integrate a PROGRESS environment (respectively a PROGRESS server), we have to implement a PROGRESS Application Object (PAO). This PAO contains the name of the remote PROGRESS server, the server port and the name of the service. In addition, it contains a PROGRESS Input Component Implementation (PICI), which implements an interface of the application server to deal with the data that have to be exchanged between the remote server and the WWW client.

The PICI extends the framework's concept of the general input component so that each object of the PROGRESS language is handled as a general object with the three attributes name, type and value. The PROGRESS language provides recursive data structures (list of lists, etc.), therefore the PICI has an array of sub-PICIs. The above mentioned Java package for the communication with a PROGRESS server provides all neccessary methods for the construction of general PROGRESS objects and the extraction of the data stored in PROGRESS objects. The actual implementation of the PICI must provide methods to transform the internal attributes to a PROGRESS object (for a request) and vice versa (for a response), but on the basis of the Java package these methods can be implemented as simple case statements.

4.1.2 *Initialization*

The PAO is totally configurable, which means that every time a PAO is instantiated by the application server, its attributes (name, service, server, port and PICI) are provided by a special PROGRESS property file. The parametric instantiation of a PAO is similar to the initialization process of general application objects as described in Section *2.2.1*. The initialization of a PAO is divided into two steps. The first step is setting the basic attributes (name, service, server and port) which are simple objects (strings and integers) that can be coded directly within the property file. The second step is the instantiation of PICI which is a bit more complicated. Therefore, we code an example input value for the denoted service as a PROGRESS language string in the property file. On the one hand, this string provides the structure of the request arguments (which corresponds to the structure of the PICI) and on the other hand it is a valid input useful in a first execution. Because we do not want to implement a PROGRESS language parser within a PAO, we use a special service (str2pg) that each PROGRESS server supports. This service simply transforms a string given in the PROGRESS language into a corresponding PROGRESS object. The above mentioned Java package of PROGRESS then allows us to construct the actual PICI for this PAO.

Now that the PAO has been constructed, it offers the underlying PROGRESS service transparently to the application server (respectively the clients) through a general interface (the PICI). Figure 3 schematically shows the structure of the PROGRESS property file.

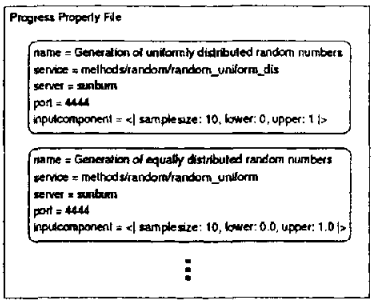

Figure 3: Scheme of the Progress property file

4.1.3 *Handling of PAOs Within the Client*

All PAOs and PICIs are mapped onto internal components of the client framework (list, table, etc.). Result input components (components that deliver the result of a PROGRESS server computation) are dispatched to the presentation layer, whereas the PAO interface and the request input components (the PICI that is instantiated within the PAO at bootstrap time) are dispatched to the interaction layer of the GUI. The use of internal framework components leads to the uniform look and feel of the user interface.

Figure 4 shows a snapshot of the GUI after the PROGRESS service for the generation of normally distributed random numbers has been executed.

4.2 The Integration of the XML-based Monitoring System SPECTO

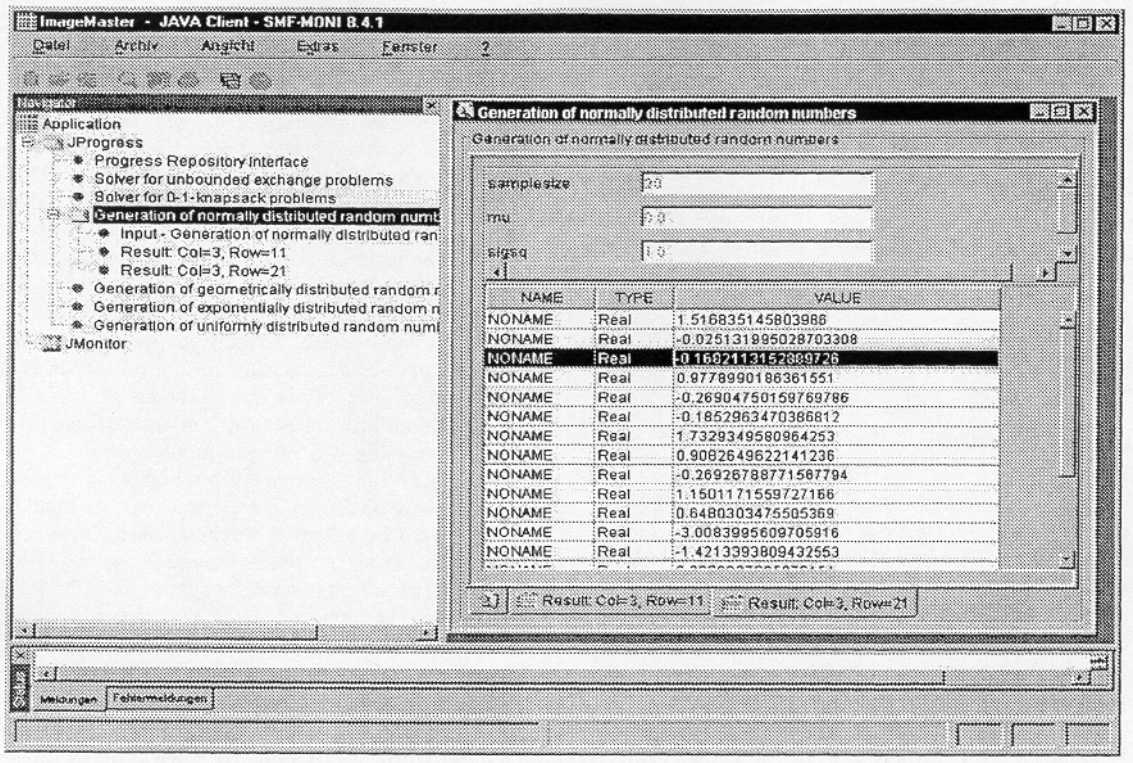

Figure 4: Generation of normally distributed random numbers

The monitoring system SPECTO[1] [10] facilitates the management of log information in distributed systems. A log information is captured as an event which is encoded as an XML application [3]. SPECTO provides two kinds of servers for online and offline monitoring. Whereas online monitoring is characterized by pushing log information to the user permanently, offline monitoring is more like a request/response communication process where log information is requested by the user. Each SPECTO server is embedded into RESPONDEO such that clients send their request messages to the application server which in turn dispatches each message to the specified application object. Each SPECTO application object is responsible for managing either a single XML-based log file or a directory of such files. For online monitoring of system components, another middleware component, called MITTO, is introduced in SPECTO, which provides a scalable infrastructure for asynchronously pushing log information to the user. However, in what follows, we focus on how the offline monitoring server of SPECTO is integrated into RESPONDEO.

4.2.1 *Initialization of* SPECTO*'s Application Logic*

There is a single, configurable application object, called *EventDatabase*, which encapsulates the monitoring-related logic of SPECTO, as shown in Figure 5. This object contains the path to the directory of XML-files (*archivePath*), the

[1] SPECTO is used in several projects by debis Systemhaus Industry (Competence Center Document and Workflow Management), which is the document and workflow management group in the Engineering division of debis Systemhaus. debis Systemhaus is the IT subsidiary of debis which in turn is the service company of DaimlerChrysler.

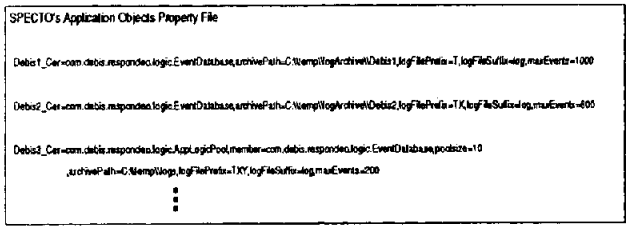

Figure 5: Scheme of the SPECTO application objects

prefix (*logFilePrefix*) and the suffix (*logFileSuffix*) of the log files and the maximum number of log entries (*maxEvents*) which are sent back to the client. In Figure 5, three application objects are specified (*Debis1_Cer*, *Debis2_Cer*, *Debis3_Cer*) which reuse the object *EventDatabase* in different application contexts. Whereas *Debis1_Cer* and *Debis2_Cer* provide only one instance of *EventDatabase* to process client requests, *Debis3_Cer* offers a pool of ten *EventDatabase* objects which are synchronized and managed by the special application object *Pool*. The entire set of application objects is multithreaded and configurable at run-time since they take full advantage of the management facilities described in Section 2.2.2.

4.2.2 SPECTO*'s Offline Application Logic* - EventDatabase

During the initialization of the application object *EventDatabase*, each XML-file is parsed in order to gather some statistical information such as the number, type (error, warning, etc.), and date of log entries. When a request is sched-

uled for *EventDatabase*, the specified XML-file is parsed using the client's parameters specified in the request message. As long as the maximum number of log entries is not exceeded, a result is sent back to the client containing all log entries as serialized Java objects in the body of the message.

5. SUMMARY

We presented a lightweight system that enables the access of various heterogeneous BES via the WWW (e.g. a Web browser). The object-oriented design establishes a framework that separates all communication aspects from the individual BES and provides an uniform GUI with general interfaces to the underlying BES and several application management facilities. Communication problems (scalability and limited bandwidth) are solved at the framework level in order to enable excessive reuse of the system's components for a rapid integration of BES. The communication components of the framework together with the loosely coupled BES lead to a lightweight, message-oriented information system that is especially useful in an environment like the WWW where an unpredictable number of users want to interact with various information sources.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge Andreas Ludwig and Manuel Geiger for their contributions on various aspects of RESPONDEO.

7. ADDITIONAL AUTHORS

Additional authors: Rainer Krautter, debis Systemhaus Industry (Competence Center Document and Workflowmanagement) CU DMS / PP, Erich-Herion-Strasse 13, D-70736 Fellbach, Germany. Email: `Rainer.Krautter@debis.com`).

8. REFERENCES

[1] P. Becker. A framework for providing and using algorithms and algorithmic meta knowledge on the Internet. In S. Ram and M. Jarke, editors, *Proceedings of the 5th Annual Workshop on Information Technologies & Systems (WITS'95)*, number 95–15 in Aachener Informatik-Berichte, pages 2–11, 1995.

[2] P. Becker. An embeddable and extendable language for large-scale programming on the Internet. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 594–603, 1996.

[3] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Available at http://www.w3.org/TR/1998/REC-xml-19980210, Feb. 1998.

[4] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10), Oct. 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[6] J. Gosling and K. Arnold. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[7] T. Hermann. Communication between Progress and Java. Internal report, Wilhelm-Schickard Institute for Computer Science, University of Tübingen, 1997.

[8] R. Johnson and B. Foote. Designing reusable classes. *Object-Oriented Programming*, 1(2):22–35, 1988.

[9] R. Johnson and V. Russo. Reusing object-oriented design. Technical Report 91-1996, University of Illinois, 1991.

[10] M. Häusser. XML-based monitoring in distributed systems. Master's thesis, Wilhelm-Schickard Institute for Computer Science, University of Tübingen, 1999.

[11] Object Managemant Group (OMG). *The Common Object Request Broker: Architecture and Specification*, Feb. 1999. Revision 2.3.

[12] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Massachusetts, 1994.

[13] R. Schimkat, W. Küchlin and R. Krautter. An object-oriented framework for rapid client-side integration of information management systems. *South African Computer Journal*, (24):244–248, Nov. 1999.

[14] S. Schroeder. Design and implementation of a system management framework. Master's thesis, Wilhelm-Schickard Institute for Computer Science, University of Tübingen, 1999.

[15] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogenous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sept. 1990.