

The KITE Application Server Architecture

Josef Templ

Software Templ OEG

Josef.Templ@aon.at,

<http://members.aon.at/software-templ>

Abstract. An application server represents a framework for server applications, which are programs that provide an API rather than a GUI and allow remote access to their functionality. The architecture of the Java-based application server KITE results from a generalization of the single-user desktop operating environment Oberon essentially by turning the global module list into a data structure associated with a particular client. We therefore introduce the notion of a *Service* as a named set of functions with state, the notion of a *ServiceContext* as an extensible set of service instances, the notion of a *ContextFactory* as the foundation for servers and clusters and the notion of an *Application* as a set of available services. We describe how these concepts are mapped onto the programming language Java and we discuss the fundamental implementation techniques being used. We also present a short overview of the available infrastructure, which is needed for using KITE in a production environment. Finally we compare our approach with Enterprise Java Beans.

1 Introduction

During the development of a large scale distributed e-business application called *ec2use* by the austrian corporation *Infonika* it became clear that the requirements could not be met with existing standard products such as Enterprise Java Beans (EJB [5] [1]) or other middleware products.

The application used a relational database as a persistent repository of business data. Initially DB2 was chosen but soon it became clear that database independence is a major concern too. The data model consisted of several hundreds of tables with some tables consisting of much more than 100.000 entries. Care must be taken to avoid code explosion for handling data models of this size and for the formulation of efficient database operations.

The application had to be fully internationalized, which also includes internationalization of database contents. This leads either to a large number of simple SQL-queries or to complex queries that use (left) JOINS in order to speed up execution. Hard-coding such queries is cumbersome and error prone. It also makes data model changes very risky and calls for some support.

The application had to be used in a variety of ways ranging from a multi-tier setup with a separate database server, a separate application server and

separated online web- and Java-clients down to an offline configuration on a laptop computer with no more than 256 MB main memory where all components were running locally and periodic synchronization steps were used to exchange information with a central server. The application must not be coded twice for the different configurations.

The application had to be platform independent and stable, which lead to choosing Java[2] as the programming language.

Clients had to be connected from within a fast local area network but also via a low bandwidth modem connection of no more than 56 Kbaud. On-the-fly data compression and encryption had to be used for business transactions over the internet. The number of network round trips had to be kept at a minimum.

In order to keep resource usage manageable, it has been decided to develop the application including the middleware (based on Java RMI [1]) from scratch, which resulted in a working application but at the expense of an ad-hoc design, which lacked clear concepts, the possibility of reuse for similar applications and a mechanism for custom specific extensions. Therefore it has been decided to redesign the application and to factor out the application independent part as an *application server*. This effort finally resulted in the architecture described in this paper.

2 Developing the Architecture

We consider an application server as a framework for server applications, which are applications that provide an application programming interface (API) for possibly remote clients rather than a user interface (UI) for local users as it is the case for traditional (desktop) applications. An application server helps to develop distributed applications in a number of ways:

- The application server defines the overall structure of a client/server application and thereby relieves the developer from many difficult design decisions.
- It introduces a server-side component architecture and allows for reuse of components.
- It provides the infrastructure for connecting a remote client to a server.
- It supports low bandwidth connections between client and server by means of compression techniques.
- It guarantees privacy and integrity of transmitted data even over untrusted networks by means of data encryption.
- It supports scalability of the server for a large number of clients.
- It provides high availability of the server and a certain amount of fault tolerance.
- It allows the crossing of fire wall boundaries between client and server.
- It supports the common case of database applications by means of connection pooling, object/relational mapping and managed persistency.
- It provides caches for both clients and servers in order to minimize network and database load.

- It provides the infrastructure for managing transactions.
- It allows the configuration of applications such that they can be used either remotely or locally without any change in the program code.

Despite the differences to traditional applications, we can derive the overall application server architecture by a careful look at a modern single user desktop operating environment, viz. the Oberon system[7], and some generalization steps.

The Oberon system contains two fundamental global data structures, the *module list* and the *task list*.

2.1 Generalizing the Module List

The module list keeps track of all modules that were either requested for execution of a command or that have been imported by another module. Every module is loaded only once and remains loaded afterwards. Oberon modules not only provide the notion of a command (as a parameterless exported procedure), but also provide an arbitrary API to be used by other (higher level) modules.

If we want to serve multiple clients, a natural approach would be to provide a module list for every client. This is in fact the base idea of the KITE architecture, but it is expressed in Java and tuned to scale well with a large number of clients.

Using Java as the implementation language, things are very similar. Java provides the concept of a class loader, which essentially plays the role of Oberon's module list. The class loader keeps track of all classes that are loaded. As with Oberon's modules, a class is loaded only once per class loader and it may provide an API to be used by other classes. In addition to Oberon, Java allows to create multiple class loaders. This may lead to the idea that a class loader per client could be used as the basic application server architecture. However, using a separate class loader per client would result in loading a class multiple times, which in Java would mean that the code is allocated multiple times in main memory¹. In order to avoid this unnecessary code duplication, we introduce a data structure per client called a *service context*. This data structure acts like a class loader but it does not keep track of loaded classes but of class instances (objects) created per client. We refer to such objects as a *service*. A KITE application represents the set of available service classes much like the Oberon system provides access to all available modules.

2.2 Generalizing the Task List

Oberon's task list keeps track of all active tasks, which are procedures that are to be called by the system at some time between commands. They act as background activities. In the context of an application server it may also be required to carry out background tasks on the server. In order to keep applications separated, a task list per application is appropriate. Since Java supports multi-threading as opposed to Oberon's cooperative multitasking, we can express tasks

¹ We will show in Sec. 3.4 that we can make use of multiple class loaders but for a different purpose.

as threads and create a thread group for every application. There is no need to provide extra support for tasks per client, since those can be created under the control of services. Only global tasks that do not belong to a particular client must be supported separately.

2.3 Server and Cluster

In order to abstract from the details of creating a connection for a remote or local client, we introduce a mechanism, called a *context factory*, which returns a service context for a specified application. A KITE application server is nothing but a context factory, which maintains a set of applications as specified in an appropriate configuration file. The context factory (alias server) will be registered with a standard Java RMI registry to be accessible for remote clients.

For configurations, where client and server are located in a single Java VM, we provide a much simpler context factory, which is not based on Java RMI.

It follows naturally, that a simple form of clustering can also be achieved as yet another implementation of a context factory. A cluster context factory would be configured with a number of servers and may distribute requests for new contexts in a round robin style or some other strategy between the available servers.

2.4 The Static View

Figure 1 shows the static structure of the application server KITE according to a concrete example. There are five services involved, where *Service1* to *Service3* belong to *Application1* and *Service3* to *Service5* belong to *Application2*. Note that *Service3* belongs to both applications. There are two tasks *T1* and *T2* involved, where *T1* belongs to *Application1* and *T2* belongs to both applications. Above that there are two application servers configured, which contain both applications. The two servers are identical, therefore it is meaningful to arrange a simple 2-node failover cluster on top of them, which will connect clients with *Server1* in the normal case, but switches to *Server2* if *Server1* fails.

The static structure closely corresponds with a hierarchy of weight as shown in Table 1. The bottom level (service) is the lightest one. A service is simply an instance of a class with the only overhead of an entry in the service context's map of services. Since there may be a large number of clients and a number of services per client, this is an important property. The next level (application) corresponds with a Java name space. It uses its own class loader, which maintains a list of loaded classes and (in current Java VMs) both the code and the static variables are replicated for every name space. An application also maintains its own profile and a thread group. The application level is considerably heavier than the service level. Tasks are somewhere between services and applications and correspond to a Java thread. The top levels (server, cluster) are the heaviest since every node corresponds to a Java virtual machine.

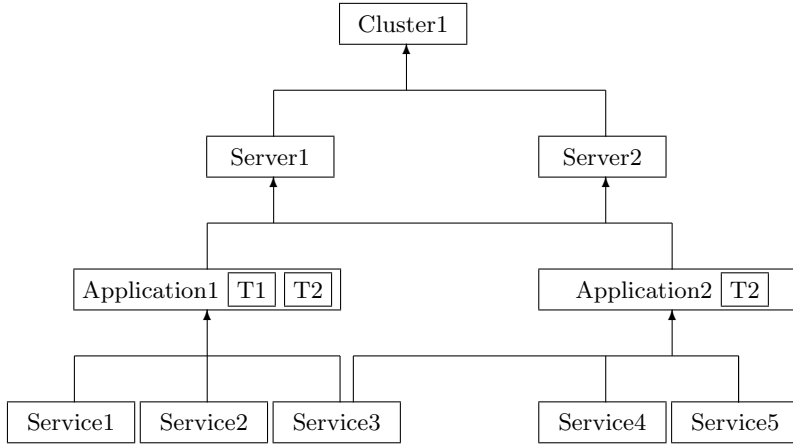


Fig. 1. The static structure of the application server KITE

Table 1. Hierarchy of weight

Level	Weight	Representation
Server, Cluster	heavy	Java VM
Application	medium	Class Loader, Thread Group
Task	light	Thread
Service	flyweight	Instance

2.5 The Dynamic View

Figure 2 shows the dynamic structure of the application server KITE by applying the previous example to a particular runtime scenario. There are two clients *Client1* and *Client2*, where *Client1* uses services of *Application1* and *Client2* uses services of both *Application1* and *Application2*. *Application1* is used by both clients. The applications are represented by application objects on both the server and the client side. Accessing a service needs services within a service context, which are not shared between the clients. On the server side, there are various services instantiated per service context, but not all available services are instantiated immediately. For reasons of simplicity the client side service context does not show its service objects, which in most cases simply are proxy objects that forward all calls to the server. An alternative client side implementation of a service, which is possible in KITE, would be instantiated in the client side service context.

After a client has requested and received a service context, it does not see the cluster or network structure behind it any more. The servers and the cluster are therefore not shown in Fig. 2.

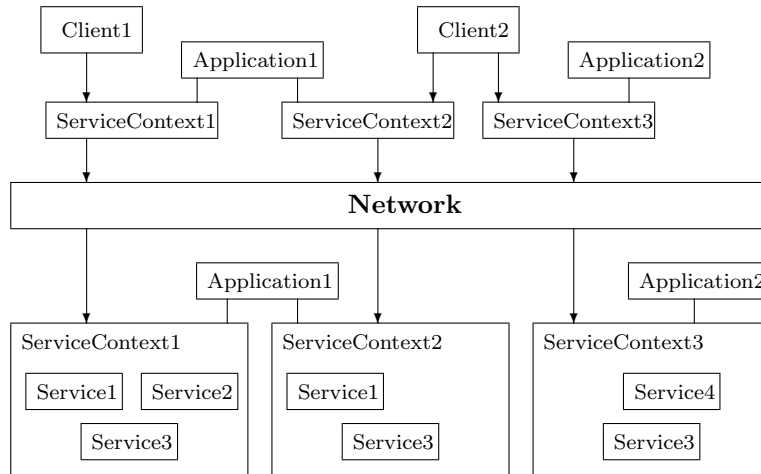


Fig. 2. The dynamic structure of the application server KITE

3 KITE Architecture

The architecture of KITE is represented by a hierarchy of concepts expressed as Java interfaces and classes that operate on these interfaces. In addition there is a recommended naming convention for classes and packages. The basic set of interfaces and classes defined by the KITE architecture is contained in package *kite.service*.

3.1 Service

The smallest unit of the KITE architecture is the so called *service*. A service provides an interface and an implementation for a particular function or group of functions (sometimes also called *business logic*). A client may access a method on a server by means of a service. Services may refer to each other if they are within the same service context. A service is potentially stateful, i.e. it may store information (session state) across individual requests for a particular client. Technically speaking, a service is nothing but an instance of a class that implements its interface and the interface *Service*. KITE provides various standard services, which may be used by all applications. Other services will be provided by the application developer.

All services follow a basic life cycle which is expressed by interface *Service*. This interface must be implemented by every service class ².

```
public interface Service {
    void init(ServiceContext ctx) throws Exception;
```

² In order to minimize coding effort an auxiliary class *kite.service.ServiceAdapter* with empty method implementations has also been prepared.

```

void reset() throws Exception;
void close() throws Exception;
void setLocale(java.util.Locale locale) throws Exception;
}

```

After a service has been instantiated, it will be initialized by a call to *init(ctx)*, which passes the service context to the service. The service may perform arbitrary initialization steps including requesting other services from *ctx*. *reset()* sets the service instance into a well defined state after an exception has been caught. From the reset state, the service may accept further requests. For example, a database service could close open transactions upon *reset()*. *close()* marks the end of the lifecycle of a service and is the last call a service receives. *setLocale()* sets the locale to be used by the service. This allows the service to behave in a language or country dependent way.

For an example service outline please refer to Sec. 8.

3.2 ServiceContext

A *service context* is a dynamically extensible set of instantiated and initialized services, where every service is instantiated only once. Instantiation of a service is done upon the first request for a particular service. A service context is similar to a Java class loader, where every class is loaded upon the first usage of the class and it is loaded only once. The difference is that classes are *loaded* but services are *instantiated*. The typical usage of a service context is that for a client, which connects to a server, a new service context is provided. This context holds the services the clients requested together with the state of these services. In principle, a client may open an arbitrary number of service contexts to an arbitrary number of servers, for example to copy data from one server to another. KITE provides implementations for service contexts for local and remote clients. In any case, service contexts are provided by KITE and need not be extended in an application dependent way.

```

public interface ServiceContext {
    Service getService(Class service) throws Exception;
    Service getService(String serviceClass) throws Exception;
    Service getService(Class serviceInterface,
                      String serviceClass) throws Exception;
    void reset() throws Exception;
    void close() throws Exception;
    void setLocale(java.util.Locale locale) throws Exception;
    java.util.Locale getLocale() throws Exception;
    Application getApplication() throws Exception;
}

```

getService() returns a service object of the requested type, which may be specified in three variants. The simplest and most important form is where the

requested interface is specified as a class object, which represents the Java interface. If the naming conventions of Sec. 4 are followed, the implementing class will be derived automatically. A new instance of this class will only be created once per context.

The methods *reset()*, *close()*, and *setLocale()* will simply be forwarded to all services. *getLocale()* and *getApplication()* provide information about the currently set locale and the application, the context is associated with.

```
public interface ServerContext
    extends ServiceContext, java.rmi.Remote {
    Object[] invokeService(String serviceClass, String methodName,
        String signature, Object[] arguments) throws Exception;
    Object invokeService(String serviceClass, int methodId,
        Object[] arguments) throws Exception;
}
```

A *ServerContext* represents a specialized service context for remote access of services. It provides method *invokeService()*, which may be used to invoke a method of a service from a remote client from within its proxy (stub) object. The first variant allows to specify a method by class name, method name and signature and returns two objects, the result and a method id, which may be used in the second variant to access the method more efficiently.

3.3 ContextFactory

A *context factory* is a class which creates service contexts. KITE provides implementations for context factories for both local and remote clients and for a simple form of clustering, but in special cases (for example for a special form of clustering) an additional context factory could be implemented.

```
public interface ContextFactory {
    ServiceContext newContext(kite.util.Profile profile)
        throws Exception;
}
```

newContext() creates a new service context. The specified profile contains the configuration parameters, which will be interpreted by the context factory implementation.

A context factory for creating remote connections is itself a remote object. It provides a variant of *newContext*, which returns a specific kind of service context with the required parameters specified explicitly.

```
public interface RemoteContextFactory
    extends ContextFactory, java.rmi.Remote {
    ServerContext newContext(String appName,
        String majorVersion, java.util.Locale locale,
        int protocol, Object data) throws Exception;
}
```

The requested application is specified by a *name* and *major version* number, the clients preferred *locale* is specified and the parameter *protocol* signals if encryption and/or compression is required, parameter *data* is reserved for future use.

3.4 Application

An *application* in the KITE framework is the set of services, which are visible via the application's class path, and a set of tasks, which are associated with the application. An application is defined by a set of configuration parameters (called *profile*) which includes name, version number, class path and profiles for the services and tasks. An application is loaded during server startup by reading the application's profile and creating a class loader for the application. The class loader is used for dynamically loading all classes and resources of the application. This results in a separation of applications in different name spaces, which minimizes interferences between applications, since all the application classes have their own code and static variables. It also allows applications to exist in multiple versions at the same time in the server. This feature is commonly known as *hot deployment*, because it allows to update an application without shutting down and restarting the server. The system classes (`java.lang`, `java.io`, `java.net`, etc.), however, are shared between the applications, which avoids the waste of resources in the server. In addition to services KITE supports the concept of Tasks, which are server side threads associated with an application. KITE provides the class *Application*, which handles the management of class loaders, tasks, etc. This class is not extended for particular applications.

3.5 Task

A *task* is a program which is associated with an application and is automatically started by the server upon loading of the application. A task always runs within the name space (class loader) of its application. Technically speaking, a task is a class that implements the interface *Task*, which extends the Java interface *Runnable*. Therefore, tasks are specialized Java threads. All tasks of an application are collected within a Java *thread group*. Tasks may use all services of the application they are associated with.

```
public interface Task extends Runnable {
    void initTask(ServiceContext ctx, kite.util.Profile taskProf);
    void stopTask();
}
```

A task is associated with a service context and a profile by means of the method *initTask()*. In order to avoid dead lock problems and the usage of deprecated Java API, tasks employ the recommended pattern for stopping threads by introducing the method *stopTask()*, which every task must implement.

4 Naming Conventions and Class Lookup

KITE organizes its classes and interfaces in a hierarchy of packages. This hierarchy reflects the fact that in a client/server application some components will be needed on the server side only, others will be needed on the client side only and some may be needed on both sides. By separating classes and resources this way, it is very easy to create archives that contain only the minimum number of files for either side. The resulting package structure used and recommended is called the KITE package triplet.

4.1 The KITE Package Triplet

All classes, interfaces and resources of a component c , which are needed only on the server side, form a package named $c.server$. All classes, interfaces and resources of a component c , which are needed only on the client side, form a package named $c.client$. All classes, interfaces and resources of a component c , which are needed on both the client and server side, form a package named $c.service$. Applying these rules to KITE itself leads to the package triplet shown in Table 2.

Table 2. The KITE package triplet

Package	Meaning
<i>kite.service</i>	Contains the interfaces and classes of the service architecture.
<i>kite.server</i>	Contains the server side implementation of the service architecture.
<i>kite.client</i>	Contains the client side implementation of the service architecture.

The standard services provided by KITE do not belong to the server, but to the applications loaded and managed by KITE. They must therefore be located in the application class path and not in the system class path if the advantages of hot deployment are to be used. For this reason, the KITE standard services represent a separate component available in the packages *kite.std.service*, *kite.std.server* and *kite.std.client*.

Application specific services, for example services of the e-business application *ec2use*, consequently lead to a package triplet consisting of *ec2use.service*, *ec2use.server*, and *ec2use.client*.

In addition, KITE contains some packages that exist on their own, for example *kite.util*, which is a collection of general purpose utility classes. In this case, a separation into client and server side classes is not possible.

4.2 The KITE Service Triplet

For each service S of a component c , which may be called remotely, there must be a Java interface. This interface defines all methods which the service provides

for remote clients. In the regular case, this interface is implemented by a class, which will be loaded and executed on the server. The interface, however, will be needed on both the server and the client. For various technical reasons (e.g. caching) it may also be necessary to provide another implementation of the interface S which will be loaded and executed on a remote client. In this general case there are three class files involved. Each of these will be located in the corresponding package of component c . To simplify the handling of such files, KITE recommends the naming convention shown in Table 3

Table 3. The KITE Service triplet

Class	Meaning
$c.service.SService$	The Java interface of service S .
$c.server.SServer$	The server side implementation of service S .
$c.client.SClient$	The optional client side implementation of service S .

If the proposed naming conventions are followed, a KITE client receives the appropriate implementation class automatically when a service is requested by specifying the interface only. Otherwise, the client would have to specify the name of the implementation class of a service explicitly.

As an example, the *MoDDL* service, which is part of the KITE standard services, would result in the triplet consisting of *kite.std.service.MoDDLService*, *kite.std.server.MoDDLServer* and *kite.std.client.MoDDLClient*.

4.3 Class Lookup

Whenever a service is requested and only the interface is specified, a mapping to the corresponding implementation class must be performed. This mapping is implemented by the service context being used. In any case the servlet triplet naming convention will be used.

For remote clients, the lookup is performed on the client side by checking the availability of a client side implementation first. If found, this class is used, if not, a stub is created. There is no network round trip involved so far. When a method of a stub is called, this call is forwarded to the server, which checks the availability of a corresponding server side implementation. If found a service will be instantiated and registered. This occurs as a side effect of the first remote method call of a service and minimizes the number of network round trips.

The service context for local clients only checks for the availability of a server side implementation class, but otherwise behaves exactly in the same way.

5 Implementation Aspects

The KITE communications infrastructure is based on Java RMI (Java Remote Method Invocation [1]), however, it minimizes the usage of RMI to an absolute

minimum. In particular, there is an RMI interface for `RemoteContextFactory`, an implementation of a context factory for remote clients. This enables the establishment of a connection between two Java VMs over the Internet (or Intranet). The RMI lookup returns a context factory stub object to the client. The stub allows the creation of a service context for a particular application. In the case of a remote client, the service context is also an RMI remote object. If the client shares the Java VM with the server, there is no RMI involved at all.

5.1 Dynamic Stubs

The individual services of a KITE application are (in the regular case) not RMI objects, however, they can be used locally as well as remotely. In case of remote invocations, the semantics of a method call is the same as with RMI. In particular, this means that all parameters and return values must be serializable. The use of dynamic stubs results in a number of advantages:

1. No need for using the Java RMI compiler (`rmic`) if the interface of a service changes and less files to deploy to the client.
2. No RMI objects for services on the server. Those objects would have a significant memory overhead, which is about 600 Byte per object.
3. No additional network traffic for lookup of services, which allows exceptionally short startup times even in the case of low bandwidth connections.
4. No additional server load or network traffic through RMI's distributed garbage collector (DGC).

Dynamic stubs in KITE follow the idea described in [3] for Oberon but expressed in Java by means of so called *dynamic proxies* of package `java.lang.reflect`.

A proxy class is created for every service and registered in the service context of the remote client. The proxy class implements a set of interfaces and any method call results in activating a generic invocation handler, which in our case simply forwards the call via method `invokeService()` of the remote service context to the server.

5.2 Socket Factory

Java RMI uses a so called *socket factory* for creating sockets. A socket represents a bidirectional communications link between client and server. The standard RMI socket factory may be replaced by KITE's own socket factory, which results in creating sockets with extended functionality. In particular, KITE sockets allow online compression and encryption of data and the routing of requests through a servlet on an HTTP server in order to cross firewall boundaries.

KITE uses compression technology based on ZLIB (via package `java.util.zip`), which is also the foundation for well-known compression tools such as `pkzip`, `winzip` and `gzip`. Encryption is based on the usual hybrid (or two-level) approach. In the first step upon establishing a connection, there will be a signature exchange based on asymmetric (private/public key) encryption between client

and server and this key is used later on to encrypt data with a secure block cipher (Blowfish[4]).

Java RMI normally uses its own TCP/IP ports. Unfortunately, these ports are usually not available if a firewall is used to protect a company's Intranet from the Internet. The routing of requests across a firewall therefore requires a special servlet, which is installed on an HTTP server and forwards requests across the firewall. Of course, the HTTP port of this server must be enabled by the firewall. The routing is transparent to KITE based applications.

6 A Note an Exception Handling

Since services in KITE are stateful objects, the question arises how to deal with the occurrence of exceptions during a method call, in particular, how and where to reset a service. KITE once more relies on a simple and proven concept found in the Oberon system. It requests (by convention) all exceptions to be propagated to the very end of the call chain. At this place, most probably a command invocation initiated by the user of an application, there should be an exception handler, which notifies the user and resets the service context by calling *reset()*. This call will be distributed to all services belonging to the context and resets the context so that it is ready for new service requests.

We intentionally declare the base type *Exception* in the *throws*-clause of Java methods rather than listing all possible exception kinds as it is recommended by Java (we are in line here with Microsoft's C# language). Thereby we avoid the need for explicitly declaring the checked exception *RemoteException* for all methods and, more importantly, we discourage the proliferation of exception handlers throughout the code. We observed that in almost every large Java program there are cases where exceptions are silently ignored and the program is allowed to continue, only to show follow up errors at some unrelated place. The resulting program behavior is almost as bad as C programs with dangling pointers.

The Java library contains some methods, which throw exceptions rather than returning appropriate values. In such cases, of course, it would be necessary to catch the *expected* exception (and nothing else) and treat it as a return value.

7 KITE Infrastructure

This section presents an overview of the infrastructure bundled with KITE. A detailed description of these topics is beyond the scope of this paper. However, it should demonstrate that KITE is not a theoretical concept but a production quality tool and that the architecture is a powerful base for solving real world problems.

7.1 Standard Services

In order to support the common task of database access in a client/server environment, KITE provides standard services, which allow the access of databases

and to map database records to Java objects and vice versa. Furthermore, KITE provides a simple *EchoService*, which may be used for connection and performance tests for remote clients.

DatabaseService The *DatabaseService* allows to perform prepared and dynamic SQL-commands on a database according to various configuration parameters (jdbc driver, connection URL, database, login, connection pool size). It provides a nested transaction model on top of JDBC's flat transaction model. An efficient connection pool is used internally in order to multiplex an arbitrary number of parallel requests on a fixed number of connections. A *DatabaseService* starts without any open transactions in auto-commit mode. All update operations will be immediately committed to the database. By calling *beginTransaction()*, an explicit transaction may be opened (manual-commit mode) and by calling *commitTransaction()* it may be closed. Calls to *beginTransaction()* and *commitTransaction()* may be nested at any depth. Upon reaching the outermost transaction level the changes will be committed to the database. This allows the nesting of method calls, which themselves contain transactions. Rolling back open transactions is done by *reset()*.

MoDDLService The *MoDDL service* provides dynamically typed access to data bases, whose model is defined with MoDDL (*Modular Data Definition Language*). A detailed description of MoDDL[6] is beyond the scope of this paper, however, the basic idea is simple. MoDDL allows to describe a relational data model together with auxiliary information in a platform independent and modular way. A compiler transforms the description(s) into an internal data structure, which can be used by *MoDDLService* to create SQL-commands (using *DatabaseService*).

The service provides support for database internationalization and for caching of database contents on both the client and server side. It also allows to perform complex database operations using a single network round trip.

The concept of a *DatabaseMapper* is introduced to abstract from the differences of the various SQL dialects.

In many cases, however, it is more productive, if a Java program does not work at the level of the dynamically typed *MoDDLService*, but uses a thin, statically typed API layer on top of that. For this reason a compiler is provided, which creates a statically typed service for any given MoDDL module (see Sec. 7.2).

7.2 Tools

KITE provides the following tools as separate programs.

KITE Management Console The *KITE Management Console* (KiteMMC) represents a GUI framework for arbitrary management tools. Adding new managers is possible with little effort. The following managers are provided by KITE:

DatabaseManager KITE provides an MoDDL based database manager. The management tasks include creating a physical database model from an MoDDL module, the removal of a physical data model, clearing database tables, and exporting and importing database contents. The database manager uses only functions which are available via a database mapper on every database. It allows easy migration of data models and contents between different database server products. The database manager also allows to create, view and maintain documentation of a data model.

ApplicationManager KITE provides managers for configuration of application servers, clusters, clients and applications.

Jhf2xml The tool *jhf2xml* (javaHelpFile-to-XML) may be used to create help texts in JavaHelp format (package *java.x.help*) from L^AT_EX files. As a converter to HTML the freely available tool *latex2html* must be used. Such help texts may be integrated into Java programs and opened for example under the program's *Help* menu or as context sensitive help. The creation of the JavaHelp table-of-contents (.TOC file) and the URL mapping (.jhm file) is done automatically by *jhf2xml*, if the L^AT_EX source has been structured using the style *javahelp.sty*, which is also provided. Using this approach, it is possible to create Java help texts without the need for any other tools.

Mapic A tool called *mapic* (*MoDDL-API-Compiler*) generates a statically typed KITE service for a given MoDDL module. A Java beans like interface is provided for accessing the attributes of MoDDL objects. Using a statically typed API allows one to find incompatibilities between a program and the underlying data model at compile time rather than at run time, because the Java compiler will complain about any incompatibilities it detects. The generated API is based on the underlying MoDDL service level. *mapic* also supports the inclusion of javadoc comments in the generated service if an appropriate property resource bundle with the documentation is available.

7.3 Utility Classes

KITE offers via the package *kite.util* a set of classes, which are supposed to be of general help for KITE itself and for KITE based applications on both the client and the server side. Among the supported concepts are:

Logging KITE provides a simple and flexible logging framework via the class *Log* and the associated sub package *kite.util.log*, which allows log output on an arbitrary and extensible set of logging media with respect to a log filter. The format of the log messages may be freely configured. Logging to the system console and logging to files are readily supported by KITE, application specific logging media, such as logging to a database, may be added. The logging properties are configured per application, i.e. every application may use its own logging media, filter and formats.

Profiles KITE provides via class *Profile* an extension to the concept of Java properties, which consists of the introduction of a hierarchy of properties similar to XML files. The property types may be defined in so called *Property Type Definitions* (PTD), which not only allow the definition of elementary properties but also structured properties, property inheritance and polymorphism. PTDs are used to validate profiles and to control a generic profile editor. Profiles and PTDs are used, for example, in the *KITE Management Console* (see Sec. 7.2).

Commands For programs that contain graphical user interfaces (GUIs) KITE offers the class *Command*. A command in this sense is a pair of methods (*action*, *guard*), which define what a command does if it is executed (*action*) and whether a command is enabled or disabled (*guard*). Checking the guards and synchronising with the command representation on screen is done automatically and thereby simplifies programming of a Java GUI significantly. It also saves a lot of memory resources since the implementation of a command would normally require a separate class per command in Java. KITE's command mechanism employs the Java reflection API to do the same job with only a fraction of the resources. Commands also allow the centralization of exception handling, which is not supported by standard Java GUI programming mechanisms.

8 Putting It All Together

This section outlines the resulting structure of a KITE service and compares it with a corresponding module structure in Oberon. We use the *MoDDLService* described in Sec. 7.1, which builds upon *DatabaseService*.

```
(* Oberon version *)
MODULE MoDDLService;
  IMPORT dbs := DatabaseService;

  PROCEDURE getRecord*(...): DbRecord;
  BEGIN
    ... (* use 'global' dbs *)
  END getRecord;
  ...
END MoDDLService.

// KITE version
public class MoDDLServer implements MoDDLService {
  private DatabaseService dbs;

  public void init(ServiceContext ctx) { //replaces IMPORT
    dbs = (DatabaseService)ctx.getService(DatabaseService.class);
  }
}
```

```

    public DbRecord getRecord(...) {
        ... // use 'per client' dbs
    }
    ...
}

// KITE client; uses MoDDLService; profile name in args[0]
import ...;
public class MyClient {
    public static void main(String[] args) throws Exception {
        Profile p = new Profile(args[0]); //read configuration
        //use convenience class 'Client' for simple clients
        Client.init(p); //initialize logging, locale, etc.
        ServiceContext ctx = Client.getContext(p);
        MoDDLService ms =
            (MoDDLService)ctx.getService(MoDDLService.class);
        DbRecord rec = ms.getRecord(...); //use service
        ...
    }
}

```

9 Related Work

The J2EE [5] standard includes a specification of a container for so called *Enterprise Java Beans* (EJB), which addresses a similar domain as KITE does. The main differences between EJBs and KITE are summarized below.

- EJBs come in multiple forms as stateless and stateful session beans, entity beans and message driven beans. KITE provides a service context, which corresponds to a stateful session bean.
- EJBs requires the use of dual interfaces, one for remote and one for local clients. KITE uses the same interface for both purposes.
- EJBs require the explicit specification and implementation of a so called *home interface* for every bean. KITE provides a generic context factory instead.
- EJBs entity beans with remote interfaces proved to be too inefficient for practical use. Therefore local interfaces may be used together with an auxiliary class that holds a copy of an entity which can be transmitted to a client as a whole. This approach is sometimes called a *session facade* and requires to program the facade manually. KITE's MoDDL service provides this functionality without the need for any programming.
- EJBs neither support the concept of services nor tasks.
- EJBs support distributed transactions based on the XA specification. KITE currently does not support distributed transactions. We can imagine, however, to introduce an additional service (*TransactionService*), which provides

distributed transactions. The problem is, that there are only very few resource managers available that can take part in such a transaction. Most JDBC drivers, for example, are not XA enabled.

- EJBs support declarative security constraints based on individual methods. KITE does not support such a feature.
- Resource usage (memory, CPU, programming effort) is significantly higher for EJBs.

10 Acknowledgements

The idea and the requirements for the application server KITE resulted from a multi year experience with the Java based E-business tool *ec2use* of the company Infoniqa (previously known as Infocom). Without this background and the foundations layed by Christian Senn, Thomas Achleitner, Stefan Übleis, Jürgen Höller, Stefan Schiffer and many others KITE would not exist. David Kusché proposed the decoupling of the application server functionality of *ec2use* into a separate product. The *ec2use* project managers Hermann Trappmaier and Vincent Stüger supported this idea. Walter Hinterhölzl and Jürgen Zauner implemented most of the communications infrastructure. Jürgen Zauner implemented the PTD-compiler and validator and Otto Weichselbaum provided the generic profile editor as well as *mapic*. Jürgen Zauner, Walter Hinterhölzl und Otto Weichselbaum also implemented first *ec2use* services based on KITE. Hristo Iliev and Markus Armbruckner implemented the KITE management console and the MoDDL based database manager. We thank all employees of Infoniqa for their support and contributions through informal discussions and last but not least we thank the Infoniqa board of directors for agreeing to publish the KITE architecture at JMLC 2003.

References

1. Flanagan, D., Farley, J. Crawford, W.: Java Enterprise in a Nutshell. O'Reilly, 1999.
2. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison Wesley, 1996, The Java Series.
3. Hof, M.: Just-in-Time Stub Generation, Proc. JMLC, Linz, 1997.
4. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 1996.
5. Sun Microsystems: Java (tm) 2 Platform, Enterprise Edition. Addison Wesley, 2000, The Java Series.
6. Templ, J.: MoDDL 1.0 language report, Infoniqa technical report, Jul 2001.
7. Wirth, N.: Project Oberon – The Design of an Operating System and Compiler. Addison Wesley, 1992.