Extending J2EE for Dynamic Integration

Shuping Ran
Software Architectures and Component Technologies
CSIRO Mathematical and Information Sciences Division, Australia
Shuping.Ran@csiro.au

Abstract

Internet access is growing quickly every day. Internet use is being extended from that of static content browsing to business application exposure that permits the integration of enterprise application systems. As one step further, business-to-business integration via the Internet is gaining momentum, where enterprise applications from one business interact with enterprise application systems from their business partners. This enables business process automation, which leads a greater efficiency.

This paper analyses Java™ 2 Platform Enterprise Edition (J2EE) in terms of its ability to satisfy this new dynamic application integration requirement. The paper identifies some shortcomings of J2EE in satisfying this new dynamic integration demand and proposes a content-based lookup as extension to J2EE specification to enable the dynamic integration. It also proposes an implementation for achieving the desired outcome.

1. Introduction

The Internet has brought many opportunities and changed our life forever. Internet was initially used for browsing static information on computer-based systems. The initial usage was quickly extended far beyond this capability because organisations have had the need to integrate the many heterogeneous legacy systems that exist in order to streamline their business processes [1] and to do business on-line. As one step further in the automation of business processes, business-to-business integration via the Internet has recently gained huge momentum, where enterprise applications from one business interact with enterprise application systems from their business partners.

Middleware technology has been evolving in a fast pace in response to the demands of the increased scope of enterprise systems across the Internet. Application servers (AS) are one type of these middleware technology products. These products provide a ready-made distributed system infrastructure for building high performance, enterprise-scale information systems. They provide an abstract model to hide the low-level complexity of systems programming interfaces, so that effort can be concentrated on implementing business functionality. AS middleware products provide infrastructure for the enterprise systems by providing system services such as transaction management, object life cycle management, resource management, security services and directory services. AS products are typically built on top of some existing middleware technologies, such as CORBA [2], Java based technologies, or COM+[3]. These middleware technologies have traditionally been used for enterprise systems with well-defined requirements, resources and application systems to integrate. This presents a static application environment. Nowadays, there is an increasing need for business automation, which involves an enterprise system integrating with other enterprise systems of their business partners. This

integration can be permanent or short-term. A company can require a permanent integration, while a travel agency may require a short-term integration, as it involves changing customer-supplier relationship (see examples in Section 2). This emerging type of application environment requires the enterprise system to integrate with a dynamic application environment, more so in the latter case. Section 2 describes these examples in detail.

Sun's Java™ 2 Platform Enterprise Edition (J2EE) [4] based application server technology continues to mature, consolidate and is gaining increasing acceptance. According to META Group – a leading research and consulting firm, 40 percent of enterprise development will use J2EE based technology by 2004 [5].

The aim of this study is to analyse J2EE technology in terms of how it currently integrates application components. A lack of support in integrating with the dynamic application environment is identified. However, an extension to J2EE specification to enable dynamic integration is possible with a relatively simple implementation.

Section 2 presents two examples illustrating the need for dynamic integration support. Section 3 gives an overview of J2EE, Section 4 analyses J2EE naming service in detail and identifies its shortcomings, and Section 5 proposes an extension to J2EE specification and an implementation.

2. Examples of dynamic application environment

2.1 Travel agency example

A travel agency *A2Z* needs a system for selecting products for their customers. The products are such things as air tickets, accommodation, car hire and tours. The product range can change frequently, as can the suppliers for the products. *A2Z* decided to use a J2EE based application server as their system platform. Figure 1 illustrates this example.

The system integrates with different suppliers' systems using different interfaces provided by the J2EE application server, such as JCA, Web Service, and JMS etc., as required. For a uniform access, a session bean wrapper is used for each supplier at the next level. The system uses a session bean (*SBselect*) with a method called *selectSupplier* to interact with these wrapper session beans.

When a travel agent uses the system to find the *supplier* for a *product* (eg. Air ticket to London) with the best price, the web component takes the query and invokes the method *selectSupplier* on the session bean *SBselect*. *selectSupplier* looks for all the *suppliers* (session bean wrappers) that provide the *product*, gets references to them and gets prices from them by invoking the method *getPrice* on each returned object. Based on these prices, *selectSupplier* which selects the *supplier* that offers the best price.

This example system cannot be implemented under the current J2EE specification, because of the static nature of the existing J2EE naming service described in Section 4. In a J2EE based application server environment, each wrapper session bean needs to be deployed with a name, the method *selectSpplier* needs to look up for the wrapper beans by their names. If there is a product change or supplier change, the session bean *SBselect* needs to be modified and re-deployed.

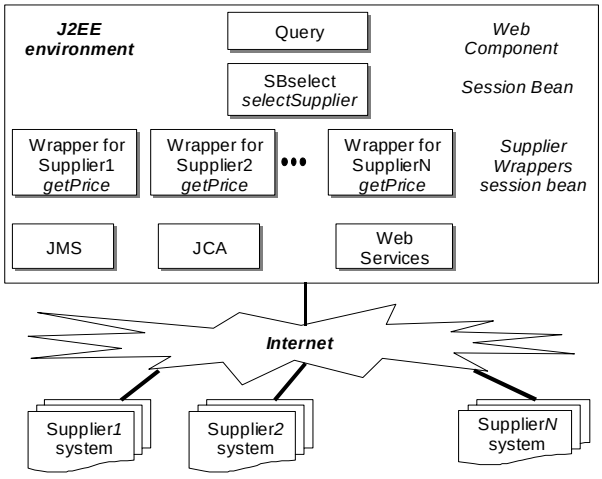

Figure 1. Travel agency A2Z example

2.2. Companies merger example

Two companies merge together, which happens more and more often these days. The resulting company ends up with two sets of systems for most functions; HR systems, for example. Integrating

these systems requires significant effort. If there were support for dynamic integration from J2EE, integrating two companies systems would become much easier.

3. J2EE overview

3.1 J2EE architecture

J2EE is a middleware specification that offers a component-based and platform independent (but not language independent) architecture. The J2EE architecture consists of containers and services. Containers provide the interface between an application component and the low-level platform specific functionality that supports the components, such as transaction management. Containers also manage the object life cycles for business objects and resource management such as database connection pooling. Containers provide the run-time environment for the application components. This hides the complexity of traditional programming from the application developers.

A J2EE complaint application server needs to support the following: HTTP/HTTPS; Java Transaction API (JTA); RMI-IIOP; Java IDL; JDBC™ Data Access API; Java Messaging Service (JMS); Java Naming and Directory Interface (JNDI); JavaMail™; JavaBean Activation Framework (JAF); J2EE Connector Architecture (JCA); Java Authentication and Authorisation Service (JAAS).

3.2. Application components

There are four types of application components that a J2EE compliant application server must support:

- Application Client – Java programming language programs that are typically GUI programs that execute on a desktop computer. They have access to all the facilities of the J2EE middle tier.
- Applets – GUI components that typically execute in a web browser, but can execute in other applications or devices that support applet. They provide powerful interface to J2EE applications.
- Web components – include Servlets, JSP pages, filters, and web event listeners typically execute in a web server. They may be used to generate HTML pages as application's user interface, or to generate XML or other format data consumed by other application components.
- Enterprise JavaBean™ (EJB) components – execute in a managed environment that supports transactions. They typically contain the middle-tier business logic for a J2EE application.

4. J2EE naming service

4.1. J2EE naming

As indicated in Section 3.1, J2EE v1.3 based application server products are required to provide naming and directory service, and Java Naming and Directory Interface™ API (JNDI). Naming provides a mechanism that allows separation of the development and deployment process, so that:

- Application component's business logic can be customised during deployment without the need to access or change the application component's source code.
- Application components can access resources and external information in their operational environment without knowledge of how the external information is named and organised in that environment.

These goals are achieved by supporting the following reference mechanisms:

- JNDI API support – specify and access the application component's naming environment;
- Enterprise JavaBean (EJB) Reference – mechanism of obtaining the home interface of an enterprise bean using an EJB reference;
- Resource Manager Connection Factory Reference – interface for obtaining a resource manager connection manager connection factory reference;

- Resource Environment Reference – interface for obtaining an administrated object that is associated with a resource (e.g. a JMS destination) using a resource environment reference;
- UserTransaction Reference – reference to *UserTransaction* object in the component's environment to start, commit, and abort transactions.

4.2. Access to J2EE naming

Application components use JNDI to access the required resources, such as an application component's environment, referencing EJBs, etc. An application component instance needs to locate the environment naming context using the JNDI interfaces first, by creating a *javax.naming.InitialContext* object, and looks up the naming environment via the *InitialContext* under the appropriate name, such as *java:comp/env* for component's environment. A few examples:

Access to Application Component's environment

```
// Obtain the application component's environment
// context.
Context initCtx = new InitialContext();
Context myEnv =(Context)
    initCtx.lookup("java:comp/env");
// Obtain the maximum number of tax
// exemptions configured by the Deployer.
Integer max =
    (Integer) myEnv.lookup("maxExemptions");
```

Access EJBs

```
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();
// Look up the home interface of the
// EmployeeRecord enterprise bean in the
// environment.
Object result =
    initCtx.lookup("java:comp/env/ejb/EmplRecord");
// Convert the result to the proper type.
EmployeeRecordHome emplRecordHome =
    (EmployeeRecordHome)
    javax.rmi.PortableRemoteObject.narrow
    (result, EmployeeRecordHome.class);
```

Access Resource Manager Connection Factory References

```
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// perform JNDI lookup to obtain resource manager
// connection factory
javax.sql.DataSource ds =
    (javax.sql.DataSource) initCtx.lookup
    ("java:comp/env/jdbc/EmployeeAppDB");
// Invoke factory to obtain a resource. The security
// principal for the resource is not given, and
// therefore it will be configured by the Deployer.
java.sql.Connection con = ds.getConnection();
```

Access to Resource Environment

```
// Obtain the default initial JNDI context.
Context initCtx= new InitialContext();
// Look up the JMS StockQueue in the environment.
Object result = initCtx.lookup
    ("java:comp/env/jms/StockQueue");
// Convert the result to the proper type.
javax.jms.Queue queue =
    (javax.jms.Queue) result;
```

Access to UserTransaction References

```
// Context initCtx = new InitialContext();
// Look up the UserTransaction object.
UserTransaction tx = (UserTransaction)
    initCtx.lookup("java:comp/UserTransaction");
// Start a transaction.
tx.begin();
...
// Perform transactional operations on data.
...
// Commit the transaction.
tx.commit();
```

4.3. J2EE naming limitations

As shown in the Section 4.2, access to EJBs and resources is accomplished by invoking the *lookup* method on a *javax.naming.InitialContext* object instance. The API for the *lookup* method is:

```
public Object lookup(Name name)
        throws NamingException
```

Where *name* is a parameter consists of the name to lookup for. *Lookup* method returns the object bound to *name*. It has the following limitations:

- The invoking application needs to know the required component's exact name; therefore, this requires a static relationship between the invoking application and the components to be invoked. Although this can be overcome by obtaining the *name* by

referencing the invoking application component's environment (refer to the access to application component environment example in Section 4.2), it is still based on a *name*. This becomes impossible when the number of components involved varies during run-time.

- The system administrator needs to keep track of what components have been deployed with what names. When deploying a new component, a new name needs to be given; this can be an administrative hassle for a dynamic or growing system.
- Only one object can be returned by the *lookup* method. This does not work when access to all the objects and resources of the same type is required. Refer to the travel agency *A2Z* example. The session bean (*SBselect*) needs to access the wrapper session beans of all suppliers offering product *x*. Under the current specification, the relationship between the travel agency and all suppliers need to be implemented as static. Each wrapper session bean needs to be deployed each with a name. The method *selectSupplier* needs to lookup them using their names one by one. This is very tedious. When there is a change of suppliers, one needs to modify the *SBselect*'s code, re-compile, and re-deploy the whole system. This will not work for a dynamic application system, such as the travel agency system, because of constant change of customer-supplier relationships and product range.
- There is a method called *list* under *javax.naming.InitialContext* class:

```
public NamingEnumeration list(String name)
                    throws NamingException
```

It enumerates the names bound the context (specified by the parameter *name*) along with the class names of objects bound to them. If this method was used for the purposes of dynamic integration described in this paper, the following problems could arise:

- It requires an application component to be deployed using a JNDI name with a context name relating to where and how the component might be accessed. In the travel agency example, the naming context hierarchy needs to represent the type of products, i.e. all the supplier components that offer *air-tickets* need to be deployed under *air-ticket* context name. This is not always possible, because a supplier can offer more than one product (e.g. *accommodation*, *conference facility*), and the component representing this supplier cannot to be deployed under different naming context.
- It does not work for lookups requiring more complex conditions, such as: lookup for all the suppliers that have the product *x* and can deliver in time *z*.

5. Extending J2EE naming

Section 4.3 revealed that the current J2EE naming presents limitations for integrating application components dynamically. A dynamic application environment needs plug-able application components where components can be deployed and re-deployed in a running system, without having to take the whole system down, modify the application code, re-compile and re-deploy.

To satisfy this requirement for dynamic integration, we propose an extended J2EE naming service, where components can be looked up not only by their names, but also by the values of their associated attributes – that is *content-based lookup*. There is some similarity to the CORBA's Trader concept [6] and the most recent Universal Description, Discovery and Integration (UDDI) [7] for web service technology, although these are designed to find services.

Under this extended naming service, application components can be deployed with a name, also with a set of attributes and their associated values. The invoking application components can search for components or resources by their names or

attribute values, and have a collection of all the objects that satisfy the lookup conditions returned. The invoking component can then invoke the necessary methods on each of the objects. This would enable application systems to integrate with their dynamic application environment. In our travel agent example, when the travel agent tries to find a supplier offering the best price for an air ticket, the method *selectSupplier* can lookup for all the wrapper session beans that have attribute "*product"* with the value "*air ticket"* and get all the air ticket suppliers objects back. *selectSupplier* then invoke *getPrice* method on each of the object returned and find the best price.

Since JNDI required by J2EE specification is an API that provides directory and naming functionality to J2EE applications and the *lookup* method is part of JNDI, this section analyses JNDI first.

5.1. JNDI overview

JNDI [8] is defined to be independent of any specific directory service implementation. Thus a variety of directories can be accessed in a common way.

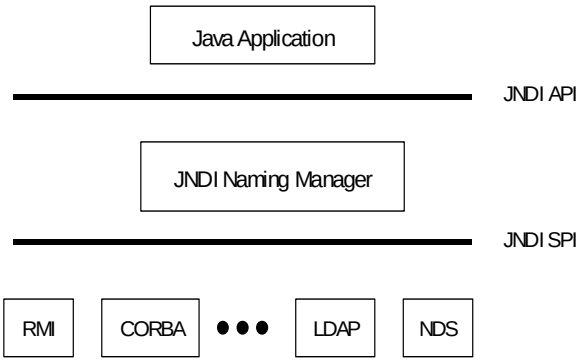

Figure 2. JNDI architecture

As shown in Figure 2, JNDI consists of a set of APIs: JNDI API and JNDI SPI. The JNDI API allows Java applications to access a variety of naming and directory services. The JNDI SPI is designed to be used by arbitrary service providers. This enables a variety of directory and naming services to be plugged in transparently to the Java applications, which uses only the JNDI API.

A directory service provides access to many kinds of information about users and resources in a network environment. It uses a *naming system* for identifying and organising *directory objects* to represent this information. A *directory object* provides an association between attributes and values. Thus directory services enable information to be organised in a hierarchical manner, to provide a mapping between names and directory objects.

Humans identify things with names. Therefore, objects can be bound to a name in a directory service. Under JNDI, a name can be an atomic name, which is indivisible, or compound name, which are composed using atomic names following some naming convention. An object can only bind to an atomic name, which represents a node in the directory hierarchy. A context is an object whose state is a set of bindings with distinct atomic names. A context provides a lookup (resolution) operation, which returns an object. Every name is interpreted relative to some context, and every naming operation is performed on a context object. A client can obtain an *initial context* object that provides a starting point for resolution of names.

The primary function of a naming system is to map names to objects. The objects can be of any type. A directory object is a particular type of object that is used to represent the variety of information in a computing environment.

The JNDI specification indicates: "If we make a directory object also be a naming context, we can represent trees of directory information where the interior nodes not only behave like naming contexts but also contain attributes."

The JNDI API is contained in four packages: *Javax.naming*, *javax.naming.directory*, *javax.naming.event* and *javax.naming.ldap*. JNDI SPI contains one package: *javax.naming.spi*.
The packages relevant to this paper are:
Javax.naming – provides access to naming service;
javax.naming.directory – provides access to directories.

The lookup operations in these two packages are:

- By invoking the *lookup* method on an *InitialContext* object. Where *InitialContext* represents first context from which an

application can bootstrap from. The following is an example:

```
ctx = new InitialContext();
Printer myPrinter =
            (Printer) ctx.lookup("colorPrinter");
myPrinter.print(photo);
```

- By invoking the *search* method on a *DirContext* object. Where *DirContext* interface behaves as a naming context by extending the Context interface. This means that any directory object can also provide a naming context. The *DirContext* interface supports content-based searching of directories. The search method returns the matching directory objects along with the requested attributes. The API is:

```
public interface DirContext extends Context {
   …
   public NamingEnumeration search
      (Name name, Attributes matchingAttributes)
       throws NamingException;

   public NamingEnumeration search
      (Name name, Attributes matchingAttributes,
       String[] attributesToReturn)
      throws NamingException;
    ….
}
```

The results of the search are returned as a *NamingEnumeration* containing an enumeration of objects of class *SearchResult*. There are more sophisticated methods that use the *SearchControls* argument, which are not covered here.

5.2. Extending J2EE naming

As discussed in Section 5.1, JNDI API offers the capability to search for objects based on specified conditions, which consists of a set of attributes and values. This capability is offered through the *DirContext* interface.

Although JNDI APIs provides this capability, it is not been used by J2EE to look up application components and resources as discussed in Section 4.2. The J2EE specification limits only to the use of invoking the *lookup* method on an *InitialContext* object, which requires the name of the object and returns a single object bound to the specified name.

We propose J2EE specification to be extended to include the *DirContext* interface to be used for content-based lookup, which is based on attribute and values. This extension would involve other extensions to the J2EE specification to support the proposed extension, such as support for *NamingEnumeration* class required by *DirContext* interface; support with tools for associating name with attributes and values; methods required for processing *NumingEnumeration* class, etc.

This extension will provide support for dynamic integration within the J2EE environment. Systems described in Section 2 will be able to integrate with their business partners by plugging (or un-plugging) system components in a running system without having to take the whole system down, and modify the system's code and re-deploy.

6. Conclusion

This paper discusses the need for dynamic integration of plug-able application components and identifies J2EE's lack of support for dynamic application integration. It proposes a content-based lookup as an extension to J2EE specification to enable this dynamic integration. It also proposes a simple implementation for achieving the desired outcome.

References

[1] R. Zahavi and D. S. Linthicum, *Enterprise Application Integration with CORBA Component and Web-Based Solutions*, John Wiley and Sons, November 1999.

[2] T. J. Mowbray and R. Zahavi, *Essential CORBA – System Integration using Distributed Objects*, John Wiley and Sons, 1995, ISBN 0471106119.

[3] D. Box, *Essential COM*, Addison-Wesley, 1998, ISBN: 0201634465.

[4] *Java™ 2 Platform Enterprise Edition Specification* v1.3, Sun Microsystems Inc.

[5] D. Sholler, *Net seen gaining steam in dev projects*,

http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2860227,00.html

[6] *Trading Object Service Specification*, OMG , v1.0 May, 2000.

[7] *Universal Description, Discovery and Integration*, http://www.uddi.org

[8] *Java Naming and Directory Interface Application Programming Interface (JNDI API)*, v1.2, Sun Microsystems Inc.