

# An Application Model and Environment for Personal Information Appliances

Olivier Gruber and Ravi Konuru

IBM T.J. Watson Research Center  
30 Saw Mill River Road, Hawthorne, NY 10532  
{ogrub, rkonuru}@us.ibm.com

**Abstract.** Recent years have witnessed a rapid increase in the use of personal information devices such as Palms or smart phones, with some people carrying more than one. Moreover, these devices need to work in disconnected mode and vary widely in their features. How does one provide an information-centric experience, across devices, for the end user? The goal of the Bali project is to provide a run-time platform that improves application portability and adaptability across devices. Bali addresses these issues through a minimal, easily deployable Java Runtime Environment, and a JavaBean-based application model. Beans transparently persist and are the units of replication. Bali provides semi-automated replication where applications only deal with the resolution of conflicting updates across devices. Code deployment is fully automated and coordinated with replication. Bali supports a powerful linking framework between beans allowing hyper-linking applications to be easily developed, even in the presence of cross-device replica. Bali fosters programmers' productivity through transparent object management and leverages a model-view-controller architecture enabling applications to adapt to device features as well as increasing code reuse. This paper describes the Bali application model, the minimal JRE, our partially implemented prototype as well as preliminary experience.

## 1 Introduction

Recent years have witnessed a rapid increase in the use of personal digital assistant (PDA) devices such as Palms or smart phones, with some people carrying more than one. Moreover, these devices typically need to work in disconnected mode and vary widely in their features and the extent of those features. For instance, two PDAs may both have a screen to display but the resolution can be different. In another instance, a PDA may only be capable of speech input and output. Despite this diversity, we believe that end users want an *information-centric* experience across their devices. First, they want to access the same information across devices. Second, they want the ability to update that information from any of their devices, while being connected or not. Concurrent updates on different devices may naturally occur and need to be dealt

with appropriately. Third, they want to be able to relate information as they see fit, adhering to the hyper-linking principles.

The number of devices, their diversity, and the end-user requirements represent a challenge for current software technology. First, it is a challenge for programmers who need to write portable and adaptive applications. Ideally, one would like the same code to manipulate the same information across devices; if not possible, one would like to leverage a maximal reuse of code. Moreover, small devices require a fine control over application footprints, which suggests a modular approach to software building. Sharing libraries and leveraging common services are also essential in that regard. Second, it is also a challenge for administrators to centrally manage all these devices, including both automated deployment and update of applications.

The Bali project addresses these challenges through a minimal Java runtime environment (MJRE), and deployable software components based on JavaBeans™, extended with a model-view paradigm. Each device wishing to participate in the Bali universe needs a MJRE. Bali applications are designed as a set of cooperative software components that discover common services at runtime. Some components are models and represent the information that is portable and replicated across devices. Models can be freely hyper-linked even across application boundaries. Other components are views and provide a multi-modal rendering. Multiple views enable application adaptability. For each device, an application has one or more corresponding views, specific to the hardware characteristics of that device. Leveraging orthogonal persistence [2], models are replicated across devices while preserving inter-model hyperlinks and thereby enabling Bali's information-centric experience for end users. During replication, the runtime up calls into the application in the presence of conflicting updates. Support for application deployment and evolution is provided via a service-oriented architecture and a versioning framework.

Although Bali's focus is on the client space, these clients have to integrate with existing web application servers, allowing their information to originate from corporate and legacy information systems. Furthermore, it makes sense to leverage mission-critical scalable storage systems to make the information of millions of devices persist. Consequently, Bali has been designed to inter-operate with the Enterprise JavaBean™[5,6] framework for advanced Java servers.

This paper presents a broad overview of Bali and in particular, describes the core concepts underlying its programming model, and how these are exploited by the runtime to automate replication. The rest of the paper is structured as follows. In Section 2, we provide an overview of the Bali components and frameworks. In Section 3, we discuss how replication is built on top of transparent persistence, describing databases, beans, and bean references. In Section 4, we discuss related work. In Section 5, we present the status of our prototype and in Section 6 we conclude.

## 2 Bali Overview

To participate in the Bali universe, a client needs the Bali runtime. Some client devices would download it from a server, some other may have it factory-installed; it may even be their only runtime environment. The Bali runtime is essentially a minimal Java runtime environment. Java is the programming language of choice for it has widespread acceptance and useful characteristics. Java is portable and supports incremental loading, both essential features when targeting heterogeneous platforms and automated application deployment from servers. Java is safe (no direct access to memory, no pointer can be forged) and it is garbage collected, providing efficient memory management. Java safety enables security, which is important for better robustness of devices, enabling information privacy and some failure isolation between applications from different providers.

The Bali runtime is a minimal JRE in the sense that it is the minimal set of Java types, classes and interfaces, needed to support the Java language specification. This is actually extremely small, less than a hundred Java types. It also contains a small-footprint support for a service-oriented platform, allowing service registration and discovery. This approach is similar to the OSGI proposal [1]. For a device, the Bali runtime also contains a set of native services corresponding to the hardware capabilities of that device. Examples of such native services are pointing device (mouse or pen), keyboard or speech recognition engine, low-level graphics, etc. These native services will obviously be a mix of Java and C code. The Bali runtime then provides a pervasive runtime foundation to deploy other services, libraries, and applications under a pure Java assumption.

The next step is to determine the device *profiles*. A profile represents a set of libraries and services that a developer can safely program against given that they are certified to be present on any device belonging to that profile. Based on which native services are available on a given device, certain libraries and services are deployable. Consequently, the device will be considered capable of downloading certain profiles. Once potential profiles for a device have been determined, the end user is offered a list of *cabinets* to subscribe to. A cabinet contains applications and *databases* which are collection of persistent objects, called *beans*. Beans are used to implement the models and views of applications. Models are non-visual beans that are pure Java data structures and therefore portable across devices. The views are more traditional visual beans and they render models. Notice the use of the term “render” in order to put the emphasis on the multi-modal nature of views, including viewing, printing, or speaking. Views are device-specific leveraging device-specific characteristics and have limited portability. More about software components, models, and views will be said later.

A cabinet is the unit of subscription, which controls replication and code deployment. On subscription, the cabinet databases are replicated and the Java types (classes and interfaces) in the cabinet are deployed. Cabinets can be subscribed to across multiple devices, thereby enabling the same information to be replicated across multiple devices. Database replication is totally coordinated with code deployment. Replication is fully transparent to programmers. By implementing models as beans,

beans are transparently replicated and their updates are automatically tracked by the Bali runtime. Concurrent updates are possible while disconnected. Potential conflicts are automatically detected and are resolved by up calling an application plug-in at a database level. Plug-ins may resolve conflicts automatically or delay it. The database frameworks will keep the multiple conflicting versions until they are merged. Conflicting versions are replicated to allow application programmers to involve end users in a more manual resolution of conflicts.

Cabinets support type evolution through versioning of immutable types. This allows us to guaranty a loss-free environment across application upgrades. Information is promoted from an older version of an application to a new one by programmer-provided conversion code. Any promotion can be automatically roll-backed if it fails or if, for any reason, the end user is not satisfied with the results. Administrators decide how long end users can wait before rolling back a particular promotion.

We believe that it is important that models can be hyper-linked as end users see fit, therefore, Bali supports references between beans, even across database boundaries. For instance, consider a list of attendee names in the meeting-entry bean in a calendar application. It is desirable to provide seamless navigation from the attendee name to the attendee address bean in an address book application, even if they originate from different data sources and preserve links in the presence of replication. The framework to achieve such a goal is the subject of the next section.

To handle the persistence of beans at servers, especially when considering that most of these beans are probably representing information extracted from legacy and corporate information systems, we have a designed a *conduit* based framework based on ideas adapted from the Palm device. A conduit is a plug-in at a database level. If no plug-in is provided, Bali takes care of bean and database persistence on the server side. If a plug-in is provided for a database, it will be up-called, during device synchronization, with any new bean state that needs to persist. Notice that a plug-in has to expect multiple versions for a bean in case conflicts cannot be resolved automatically. Conduits allow database contents to be stored and managed in external industrial-strength storage systems such as DBMS, CICS, Notes, etc.

The interface between the Bali runtime and the conduit is bean based. When a bean needs to be saved, the conduit is up called with that bean and the bean identity within the Bali world. The bean is passed as a Java object, not as a serialized state.

Ensuring the long-term persistence of beans is actually not enough; bean references may also have a state that requires to persist as well. The principles of the interactions with the conduit are the same. Bean references need to have an identity in the Bali world as beans do so the graph of beans and bean references can be reconstructed.

### 3 Replication

Bali advocates transparent persistence as the basis for replication. Orthogonal persistence [2] is a simple extension to garbage collection: objects persist as long as

they are reachable from persistence roots, some roots being long-term roots and therefore surviving JRE shutdowns.

The Bali registry provides a natural long-term root for persistence. The registry is basically a naming service for supporting a service-oriented architecture. It is simply a well-known place to find what is available in the current JRE: deployed services, available libraries (Java packages), and of course deployed cabinets and their databases.

Replication builds on transparent persistence through the concepts of databases, beans, and bean references. A database is a collection of persistent beans and can be replicated on any devices. The first technical challenge with replication is our optimistic approach to concurrent updates. The rationale for an optimistic approach is the intermittent connectivity between devices and servers. In an optimistic approach, it is impossible to merge unconstrained object graphs that have been concurrently updated.

Databases and beans provide the first necessary structuring concepts, carving out sub-graphs from the total graph of persistent objects. Beans can then be time-stamped and a straightforward protocol for maintaining replicas up to date can be applied. But this is not enough. First, concurrent updates may still conflict and such conflicts have to be resolved. Secondly, replicating beans may very well break references between these beans. For instance, suppose that a meeting entry in a calendar database refers to the meeting attendees: entries in the address book database. Each entry is a bean, that is, a persistent object. Bean references are therefore Java references between objects. Persistence and replication have to collaborate to allow new versions of beans to be installed during synchronization without breaking referential integrity. Our approach to solve these two problems is presented in the following sections.

### 3.1 Bean Model

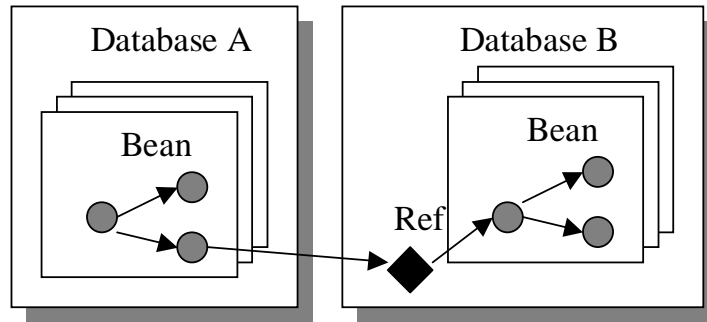
A bean is an object graph and can have cycles. It can be shared and therefore referenced. To implement beans, Bali provides two classes: a `Bean` and a `BeanRef` class.

A bean is the graph of reachable objects from an instance of the class `Bean`, stopping the recursion (non-inclusively) on instances of the class `BeanRef`. In other words, a bean is a complex object graph whose boundary is defined by `BeanRef` objects. Note that the bean framework is for carving out sub-graphs for replication purposes, not for persistence. Locally on a device, Bali adheres to orthogonal persistence. Fig. 1 shows two databases with beans and cross-database bean reference.

Designing a bean is a three-step process. First, one has to implement a data structure by sub-classing the class `Bean`. Second, one has to define the interfaces that will be used to interact with the bean, including the event model if the bean allows listeners to be notified of changes. Third, subclasses of the `BeanRef` class have to be provided, each implementing one or more interface. These `BeanRef` subclasses will play the role of “*remote*” reference to the bean, acting as proxies for the interface they implement.

Let’s illustrate beans and bean references with two examples: Swing trees and HTML. In Swing, from JavaSoft, several collection APIs are defined such as list,

table, or tree. One may wish to implement a bean that exports a tree API so that a Swing viewer can render it. One will have to choose an implementation and also to design the various bean references for carrying the Tree and TreeNode APIs. Another example is HTML along with its W<sup>3</sup>C Java API. A browser designer could choose to implement a bean supporting an HTML document and exporting the corresponding document object model as specified by W<sup>3</sup>C. Additionally, it is interesting to remark that an HTML document is a tree, so the HTML bean may want to export a Tree API as well, allowing an authoring tool to display the HTML tree.



**Fig. 1.** Beans and Bean references

This example introduces the interesting issue of multiple references carrying multiple interfaces. We need to provide a way of navigating between the different APIs supported by a bean. Java provides a type casting operator allowing interface navigation but on a single object. Unfortunately, this does not work here because a different bean reference object may carry each interface. Having multiple reference objects instead of multiple Java references shatters the Java notion of two objects being equal with respect to their identity.

We therefore introduce a new navigational capability that enables to test for identity equality on beans. We introduce the IBean interface, root of all interfaces on beans. IBean provides interface navigation through the method "QueryInterface" which takes a type as an argument. If the bean is capable of returning a bean reference implementing that requested type, it returns a Java object that can be downcast to the specified interface. If it cannot, it returns null. The returned object must be an instance of the BeanRef class, which implements IBean, allowing interface navigation from any bean reference. Interface navigation actually provides a solution to test if two beans are identity equals. QueryInterface must always return the same bean reference object when requested for an interface of type IBean, thereby allowing Java identity to be used to compare the bean reference objects.

Because bean references are delimiting the boundaries of beans, they entail some marshalling to avoid leaking Java reference on internal objects. Fortunately, this marshalling does not imply any network communication or thread-context switch. Indeed, direct Java references across bean boundaries are forbidden from a replication point of view. In other words, such a cross-bean reference has to be on an instance of the class BeanRef to isolate the states of the two beans. We are conscious of the

resulting overhead, both in time and space, as well as the lack of transparency that it entails. While at first, we were concerned about this overhead, it proved quite necessary for most applications and even proved to be quite aligned with Java philosophy. In fact, Java tends to promote a programming style where data structures are accessed through intermediate objects. For instance, hash tables or vectors are accessed through enumeration object. So if we are thinking of a hash table or a vector as a bean, the enumeration object is nothing else than a bean reference.

More importantly, most data structures, especially when considering application interoperability, will want to control reference on their internal structure as well as the kind of access rights. For instance, a name service may want to export a read-only or a read-write interface to different users. It is important that a simple Java cast does not allow upgrading a read-only interface to a read-write interface. By using two bean references, on two different Java objects, we prevent the use of Java casting, forcing to use our interface navigation where extra checks can be included. Having bean references also allows revocation of references. For example, the name service may allow deletion of a directory in which case the name service would revoke the bean references on that directory. Having an indirection would prevent other applications from hanging on to a directory object that does not belong to the naming service anymore.

Additionally, bean references support logical references, offering a late-binding mechanism through a primary key or a name. A typical example is a URL that is a reference but binds to the document through a naming service. Another typical example is a spreadsheet exporting its cells to external tools. Most likely, the goal is a logical naming of relevant cells, that is, through column and row numbers. Having the spreadsheet export a selection API based on row and column numbers can do this. Most likely, an enumeration object, which can be an object reference, will then represent a selection. That object reference would have a strong reference on the spreadsheet and the row and column ranges, allowing logical access the cells.

To summarize, it came to us as a surprise that bean references would prove actually very useful for programmers as opposed to being only a system-driven overhead for supporting replication. Also, it seems that the corresponding overhead is light. Indeed, most method invocations are internal to beans and are consequently using direct Java references, causing no further overhead. Only method invocations for cross-bean interoperability go through bean references, and the ratio has proved to be quite favorable so far.

### **3.2 Database Concept**

A database is simply a collection of beans as far as replication is concerned, the consistency granule being the bean. The Bali JRE keeps track of which beans are modified across replication cycles, allowing to drive and optimize the replication consistency protocol. This obviously requires support at the Java virtual machine (JVM). Without virtual machine support, programmers have to inform the JRE about bean updates, which would be error-prone but possible.

In the absence of concurrent updates on different devices, the consistency protocol is quite simple. Beans carry a replication sequence number (RSN). The server

increases the RSN of a bean whenever a client provides a new state for that bean. This happens when the client has modified a bean since the last synchronization. Conversely, a client will acquire the new state of a bean if its locally known RSN is smaller than the one at the server.

Because databases may be replicated on different devices, it may happen that the same beans be updated concurrently on different devices. Such a conflict is not resolved automatically. However, it is detected automatically. In the presence of a conflict, the server keeps the different conflicting states for a bean, as versions, this ensures a loss-free approach. Reconciliation is an application-specific task, and as such, is carried by programmer-provided plug-in at a database level. Upon synchronization, the plug-in is presented with the conflicting beans. Some plug-ins will be able to offer automated reconciliation during synchronization. This is the most effective approach for only one version per bean is kept and therefore replicated on each device. If a fully automated approach is not possible, the reconciliation can be delayed. The server keeps the conflicting versions and replicates them on devices. The applications may then request the end user to re-conciliate the different versions of a bean.

## **4 Related Work**

The Bali work is related to many other works such as other JRE, Java profiles, persistent languages, software component models, model-view paradigms, replication algorithms and protocols, and finally schema evolution support. Quite a vast body of literature exists in these domains and our design is happy to rely on it, not claiming contributions. Indeed, the main contribution of Bali is to define a complete and practical end-to-end solution within the context of orthogonal persistence and model-view paradigm allowing for transparent replication of hyper-linked information.

Bali relates to PalmOS or Lotus Notes on a replication front. PalmOS has a very interesting approach for small devices although it does not promote transparent persistence. PalmOS advocates a database concept similar to ours. A database is a set of records and in-memory construct. Applications are mapping their data structures directly onto database records, mapping C structures on the memory chunks of records, leveraging records references for building linked data structures. Through databases, PalmOS avoids format translation and byte-copy overheads but it suffers from promoting a dialect of C language. The reference framework for records is weak, the approach suffering from a lack of garbage collection. Additionally, mapping C structures does not work across hardware architecture. Records are untyped sequence of bytes from a runtime perspective, preventing the replication engine to apply the necessary format translation (byte order and alignment).

Lotus Notes has been very successful, based on very similar assumptions: databases, replication, forms and views, automated deployment, etc. The two main differences are that Lotus Notes does not support a real programming language such as Java, but supports scripting through LotusScript. The second difference is that databases contain traditional tuples instead of persistent objects. Forms and views are conceptually equivalent to models and views, and Lotus Notes support hyperlinking.

Not assuming Java just makes runtime support easier as long as scripting and forms are enough for the targeted applications.

Of course, Bali relates to JavaBean™ and Enterprise JavaBean™. The Bali extensions are mostly around multiple interfaces and references, along with replication based on transparent persistence. In fact, our extensions have been quite inspired by Microsoft COM. The same influence can be seen in the early Sun's proposals for the Glasgow specification, tentatively the new coming JavaBean™ specification. A BeanRef is very similar to Microsoft Monikers. Beans are similar to COM objects, both supporting multiple interfaces with navigation. Beans are also better integrating a model-view paradigm in a very ActiveX way. Although sharing concepts with COM, Bali enjoys accrued robustness and security through Java.

Bali also relates to OSGI in its goal of automated deployment of software components. OSGI is proposing a framework to bundle code and data in JAR files that are the unit of deployment. A JAR file contains services and packages which can be exported. It may also import packages. We differ from OSGI by looking at immutable but versioned types. OSGI has basically no support for evolution, blindly relying on name equivalence. The deployment of new versions of JAR files may lead to failing services and unusable devices.

## **5 Status**

Bali is still in its early prototyping phase. We have completed the design and implementation of the minimal JRE and the framework for databases, beans, and bean references. We have exercised that framework to develop some applications as well as some part of the system itself. For instance, the naming service of our registry is in fact implemented as beans and bean references. The code repository on the Bali server is also coded using databases, beans, and bean references. Bean references have proven to be quite powerful and easy to use, although a bit fragile from a consistent marshalling point of view without tools to generate them.

Transparent persistence is supported by an underlying object store using a single-level store technology. It is plugged under the J9 JVM from IBM OTI, specialized in Java for embedded systems. Although J9 runs on many hardware and operating systems, our prototype runs only on Windows so far. We are currently porting it to Palm and are looking at embedded Linux. Replication and automated code deployment are both fully designed and coding is underway.

## **6 Conclusion**

The Bali project is exploring how the qualities and success of the Java language can be leveraged in order to provide a pervasive runtime environment suited to pervasive devices. We are interested in allowing an information-centric experience for end users on replicated data across devices.

We have chosen the cornerstone of orthogonal persistence associated with a software component model that adheres to a model-view paradigm. We believe

orthogonal persistence is best suited for small devices for it best benefits from their persistent main memories. This model-view paradigm, along with a transparent persistence assumption, enables automated replication with optimistic consistency, across different hardware architectures. We plan to demonstrate it is a more viable long-term approach than approaches based on markup languages such as WAP.

Bali's main contribution is in exploring a Java-based platform as a practical solution by assembling a complete, end-to-end solution for server-based businesses to deploy information, applications, and services onto pervasive devices. Our approach nicely integrates with Enterprise JavaBeans™ servers, opening our approach to legacy and corporate information systems.

Bali is still in its infancy and experience is needed to evaluate the usability and practicability of the approach. This is especially true regarding type evolution. The design is complete and the prototyping well underway. This summer, programmers outside our group will be programming against the model, and we plan to present their experience at the workshop.

## Acknowledgments

The authors are indebted to Robert D. Johnson, at IBM Watson Research Center, for making this research possible, refining its scope and providing valuable contributions to the definition of its corresponding business case.

## References

1. Open Services Gateway Initiative, Specification V1.0, see <http://www.osgi.org>
2. M.P. Atkinson, L. Daynès, M. Jordan, T. Printezis, and S. Spence, “*An Orthogonally Persistent Java*”, ACM SIGMOD Record, Volume 25, Number 4, December 1996.
3. M. Dmitriev: “*The First Experience of Class Evolution Support in PJama*”, The Third International Workshop on Persistence and Java, Tiburon, California, September 1998.
4. M. Dmitriev, M. Atkinson, “*Evolutionary Data Conversion in the PJama Persistent Language*”, In the Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases”. In Association with 13<sup>th</sup> European Conference on Object-Oriented Programming, Lisbon, Portugal, June 1999.
5. *JavaBeans™ Specification*, see <http://www.javasoft.com>
6. *Enterprise JavaBeans™ Specification*, see <http://www.javasoft.com>