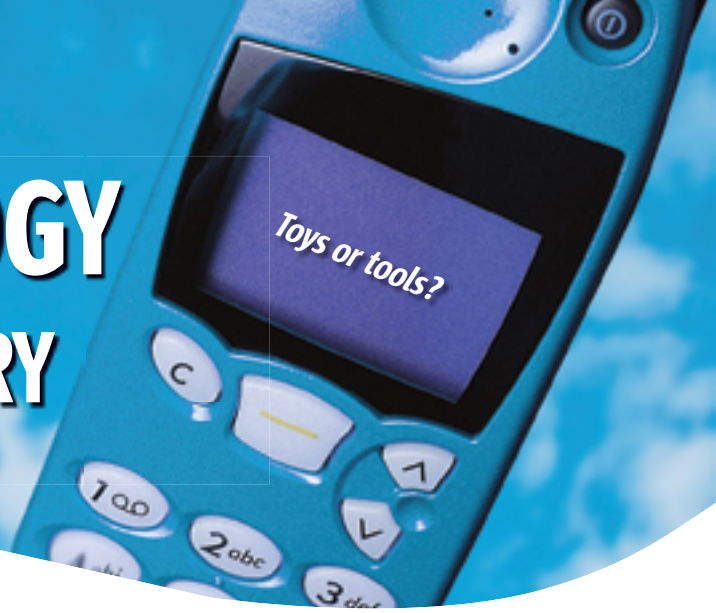
JAVA TECHNOLOGY FOR THE WIRELESS INDUSTRY



by David Parsons, Ilan Kirsh, and Mark Cranshaw

The Java Technology for the Wireless Industry specification (JTWI) encompasses a standard set of J2ME APIs for mobile device development that is being widely adopted by mobile telephone service providers, making it an important platform for Java developers.

Its core component, the Mobile Information Device Profile (MIDP), provides a number of specialized libraries for multimedia and games development; however, its underlying subset of general purpose Java classes is strictly limited. In addition, support for persistence via the Record Management System is relatively poor. This raises the question: Is JTWI a realistic application development tool or is it only good for games and other software trivia?

In this article we try to answer this question by exploring the viability of MIDP as a tool for nontrivial application development. An enterprise application that includes mobile components might reasonably expect to devolve some of its business processes and data management to mobile devices. Our chosen example, which considers both of these aspects, is a proposed implementation of the Java Data Objects (JDO) specification, which includes a number of interesting features that highlight the constraints of working with J2ME APIs for limited devices. We describe the issues around the development of such an implementation, the limitations that MIDP imposes, suggest some useful workarounds and architectural options, and finally draw some conclusions about the usefulness of JTWI as a set of APIs for serious application development.

Handheld computers such as the Pocket PC and the Palm can support reasonably complete Java Application Programming Interface (API) sets that can be used to develop serious enterprise applications, but smaller devices such as mobile telephones only support Java APIs that have been significantly reduced to work within the confines of limited hardware. There is no support for the types of persistence mechanisms that we have come to expect on larger Java platforms. The question we address in this article is whether the mobile telephone is a viable platform yet for serious business or scientific applications that need to store and process data locally and expect the services of a reasonably rich set of Java APIs. To make this assessment, we look at the JTWI specification, which provides APIs that can be supported by the limited hardware available on current mobile telephones. In particular, we examine the MIDP specification, which is a core component of JTWI.

To date, we have seen considerable development in areas such as mobile games development, but more serious business and scientific applications will need to be developed if JTWI is to be a useful component of enterprise software systems. Since such systems are likely to require considerable support for persistence, we focus on the JDO specification and examine some potential issues that arise when attempting to implement this specification using MIDP, extrapolating from this analysis to assess the general usefulness of MIDP as a general purpose application programming platform. We also look at how MIDP devices fit into larger distributed architectures that can mitigate the limitations of mobile telephones as Java application platforms.

The Java 2 Micro Edition (J2ME)

To cater to a wide range of small devices and application requirements, the J2ME architecture (see Figure 1) provides multiple configuration and profile layers that overlay the specialized Java Virtual Machine (JVM) and operating system. Configurations define the minimum set of available JVM features and class libraries for a specific category of device and are hardware focused, while profiles define the set of APIs available for a particular market category of devices and are software focused.

J2ME configurations specify the minimum requirements for memory, Java language features, JVM support, and runtime libraries. There are two standard J2ME configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). For the smallest portable devices, such as mobile telephones, the standard configuration is the CLDC. This configuration requires a very small virtual machine, such as Sun's KVM (Kilobyte Virtual Machine) or CLDC HotSpot Implementation, with footprints of only about 50–80K. These virtual machines don't have to comply with the full JVM specification, nor do they have to support the complete Java language specification. API support is limited to a selection of classes from a few packages from the Java 2 Standard Edition (J2SE), plus the Generic Connection Framework (GCF), comprising a hierarchy of connection interfaces (and the Connector factory class) that are intended to provide a generic way of expressing operations on connections regardless of the actual protocol.

David Parsons is a senior lecturer in information systems at Massey University, Auckland, and a knowledge engineer for Software Education Associates, Wellington. Until last year he was the director of emerging technologies at Valtech, London, and prior to that principal technologist at BEA Systems.

d.p.parsons@massey.ac.nz

Java Technology for the Wireless Industry

The key goal of the JTWI specification is "to minimize API fragmentation in the mobile phone device market, and to deliver a predictable, clear specification for device manufacturers, operators, and application developers" (http://jcp.org/aboutJava/communityprocess/final/jsr185/index.html).

Thus we can reasonably expect the next generation of Java-enabled telephones to support these technologies. The specifications included within JTWI are:

- ***Mandatory specifications:***
  –Mobile Information Device Protocol (MIDP) 2.0
  –Wireless Messaging API (WMA) 1.1
- ***Optional specification:***
  –Mobile Media API (MMAPI) 1.1
- ***Minimum configuration on which JTWI is built:***
  –Connected Limited Device Protocol (CLDC) 1.0

From an application development perspective, the most important API is MIDP (and by implication the CLDC upon which it builds), since these are the APIs that provide a subset of the standard Java packages found in the Java 2 Standard Edition, along with additional APIs specifically tailored for mobile development. One important issue with JTWI is that it mandates only CLDC 1.0, not CLDC 1.1, which, as we will see, introduced some important new features.

The Mobile Information Device Profile

The MIDP is one of two profiles (the other being the Information Module Profile) that works on top of the CLDC. It provides graphical interfaces for interactive applications and is the standard Java profile for mobile telephone development under the JTWI specification.

There are seven packages containing the additional classes and interfaces of the MIDP, providing user interface features at two levels of portability, sound support, certificate-based authentication, persistence, and the MIDlet framework for deploying classes into a MIDP environment. There are also extra classes in the javax.microedition.io and java.util packages.

The set of APIs available to a MIDP developer will be the set of classes in the CLDC and MIDP. Figure 1 summarizes the relationships between the relevant configuration and profile APIs and the underlying J2ME JVM.

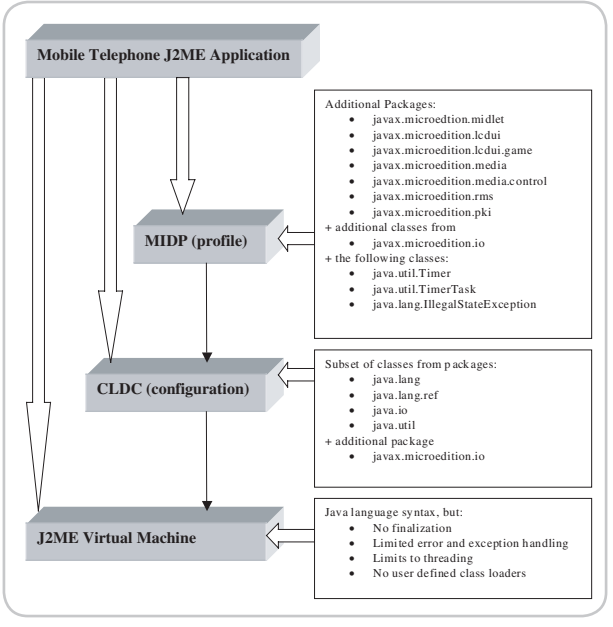


Figure 1 The APIs for MIDP development

What Is Missing from MIDP?

To consider how viable MIDP is as a general-purpose programming framework, it's useful to explore which packages and classes are excluded from it when compared with the standard edition. Of course, MIDP provides its own classes for graphics and sound, so there are no AWT, Swing, or sound-related packages from the standard edition. Similarly, MIDP has its own (limited) security classes, so there are no javax.security packages either. Since the Generic Connection Framework covers connectivity, packages relating to CORBA and network connections are also excluded. RMI likewise is not part of MIDP; although there is a separate J2ME RMI profile, it can only be used with the CDC configuration, not with CLDC. Other missing packages are those that relate to JavaBeans, reflection, XML, printing, and JNDI.

Although MIDP excludes many packages that are present in the standard edition, many of these have equivalents in the MIDP packages or, like printing, are not particularly important for mobile devices. However, it is in those packages that are included in MIDP that we find the most constraining factors, since these packages have far fewer classes in MIDP than in the standard edition. For example, MIDP includes only one interface and nine classes from java.util, as opposed to 14 interfaces and 41 classes in the standard edition (version 1.4), principally due to the absence of the Java 2 Collections Framework.

MIDP Persistence with RMS

In the context of enterprise Java development, there are a number of standard APIs that can be used to support data and/or object persistence: serialization, JDBC, Java Data Objects, and entity Enterprise JavaBeans (Enterprise Edition only). In contrast, MIDP does not automatically support any of these persistence mechanisms. The CLDC java.io package contains only the lower-level streams, readers, and writers and doesn't contain any file or object streams, or, indeed, the Serializable interface.

This means that persistence-related code in MIDP differs considerably from other Java programming contexts and uses the Record Management System (RMS). The RMS comprises variable length record stores, each of which is a collection of variable size binary data records. Each record store has a unique name, and each record within a store has a non-reusable integer index that acts as a primary key. Although individual operations on a record store are atomic, there is no transactional support apart from a version number that can be used to support manually implemented locking strategies.

The Java Data Objects Specification

The Java Data Objects specification is an output of the Java Community Process (JSR 12), which had its first final release in April 2003. JDO defines an interface-based standard for the persistence of domain objects. There are currently around 20 vendors offering JDO implementations with varying levels of specification compliance. JDO implementations can be used with a range of data stores and across all three editions of the Java 2 platform and is a recommended persistence mechanism for data-centric applications on mobile devices.

JDO can be used in the context of a nonmanaged or a managed scenario. The former case refers to a typical two-

NEED HEAD SHOT

Ilan Kirsh is a ??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????
??????????????????

???@???.com

tier or embedded application, the latter to a server-based architecture with the JDO implementation resident within a J2EE container. In both cases the JDO implementation hides Enterprise Information System (EIS) specific issues such as data type mapping, relationship mapping, and data retrieval and storage from the application components. In addition, the managed scenario allows the application to make use of J2EE container mechanisms for transactions, security, and connection management. Clearly any JDO implementation within J2ME would be nonmanaged, though a distributed architecture that used JDO on mobile devices and also on a remote server would be possible.

The rationale behind JDO is that an application needs be extended only once to encompass the JDO architecture and will then be able to access multiple and different EIS: object database systems, relational database systems, mainframe transaction processing systems, or ERP systems. The EIS vendor will be able to create a single JDO implementation and in doing so will allow pluggable access to any JDO-compliant application. The ability to run JDO in a MIDP context depends on the ability of the JDO implementation to provide the services required by the specification using only the resources available in a small device.

From the application perspective the primary interface is PersistenceManager, supported by the Query and Transaction interfaces. An object implementing this interface will be responsible for the management of the JDO instance life cycle including reading and writing from the data source, and working with Query and Transaction objects to create the illusion that the entire network of objects reachable from the application, including persistent objects, are resident in memory at the same time. Within a nonmanaged environment a JDO application will retrieve a PersistenceManager directly from a PersistenceManagerFactory instance provided by the JDO implementation.

Candidate persistent classes must implement the PersistenceCapable interface, typically through the use of an enhancement tool provided as part of the JDO implementation. Enhancement is carried out after the domain model is complete and is usually applied to the compiled byte code, making it a completely transparent process. The binary compatibility of JDO instances means that they are portable between EISs without recompilation, providing a JDO implementation is available. It would not be realistic to expect a mobile JDO implementation to include the development tools, such as the byte code enhancer, for JDO. Mobile devices are unlikely to be used as software development platforms. Rather, the device would be required to act as a JDO client, downloading pre-enhanced byte code and using the JDO client APIs. However, it's not enough for a JDO client application to have only byte code, since any class that is to be persisted must have a corresponding entry in an XML file, known as the persistence descriptor file. The developer may supply a separate descriptor file for each class or a single file for a package. This file, containing key mapping information, must be available to the JDO implementation at runtime. Therefore a MIDP JDO implementation would need to include both the JDO client APIs and the means to process the persistence descriptor.

NEED HEAD SHOT

Mark Cranshaw is a
??????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
??????????????????????????????
?????????
???@???.com

Implementation Issues for JDO on MIDP

Current JDO implementations cannot run on highly constrained devices using the J2ME CLDC configuration. The smallest realistic JDO platform is currently a CDC configuration using a larger nonstandard library, such as the IBM JCL Max available with the J9 VM. This kind of configuration is appropriate for Pocket PC type devices but not for mobile telephones.

There are a number of reasons why implementing JDO in a CLDC/MIDP environment is problematical. The core issue is the required footprint size, because a JDO implementation would require space for the implementation itself, plus the larger byte code of classes enhanced to become persistent. Object data would have to be stored using RMS or a proprietary JDBC implementation. Depending on the way that it mapped the objects, overall storage space requirements for JDO persistence would probably be larger than the most optimized equivalent using direct RMS. Another problem would be the speed of RMS, since it's just a set of indexed flat files that require either direct indexed access or sequential access to locate data. This could be very inefficient when attempting to implement the JDO query language, JDOQL.

As we've discussed, JDO also requires an XML persistence descriptor that is accessible at runtime. A full validating parser to process this file would be much too large, but there are highly optimized nonvalidating XML parsers designed for use with J2ME, such as kXML or others that implement the XML APIs of the J2ME Web Services Specification. An alternative strategy might be to use some other way of providing metadata; for example, using the optional JAD (application descriptor) or manifest files.

Another limitation of the MIDP APIs is that they don't include the JDK 1.2 collections framework, so the only container classes that are available are the Vector and Hashtable from 1.x versions of Java, which is inconsistent with the JDO requirements of at least supporting the HashSet. We could develop some kind of wrapper class around the Vector to produce a different type of container that can be used in a J2ME context; but this doesn't address the further issue that Vector is synchronized, which could impact performance. Another option is to write a custom HashSet class (using an array of linked lists) and/or other custom library classes.

To mitigate the potentially large footprint resulting from the proliferation of additional classes necessary to make JDO function in a MIDP context, a byte code shrinkage and/or obfuscation tool could be used. Since the main shrinkage benefit is gained where only small parts of libraries are used, the already

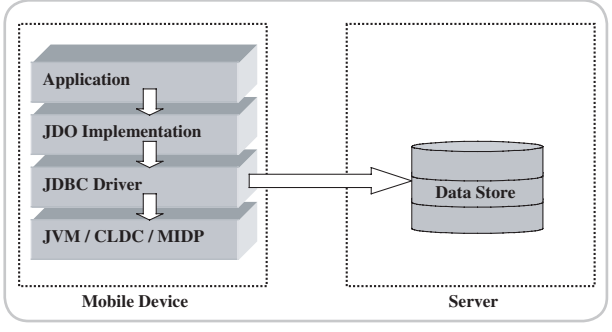


Figure 2 Client JDO architecture for MIDP

small J2ME libraries would not necessarily yield a major change in footprint. For example, test results show only a 16% reduction in size for a sample J2ME application, as opposed to a best result of 91%.

In addition to limitations of the MIDP APIs, the JTWI specification imposes further restrictions by mandating only version 1.0 of CLDC. This means that JDO implementations based on CLDC 1.1 could not be guaranteed to work on a device that was JTWI compliant. One significant issue here is the lack of floating point support, which is required by JDO but only supported in CLDC from version 1.1.

Similarly, options for providing the required support for the cache are restricted. Implementing a JDO-compatible cache requires weak or soft references, because the JDO specification requires that persistent objects remain in the cache as long as they are in use by the application, but will be removed from the cache automatically when the application stops using them. Because the application does not report when objects are disposed of, the only way for a JDO application to know that an object can be removed from the cache is to use weak or soft references.

CLDC does not include support for soft references and weak references have only been supported since version 1.1. An alternative approach for managing the cache could be to hold objects in the cache only when a transaction is active. When the application reports the transaction closed, all the objects that were retrieved during that transaction could be removed from the cache. This approach was common in object database implementations for older Java versions in which weak or soft references were not provided.

Given these constraints, a JDO implementation for MIDP cannot easily meet the full JDO specification and pass the Technology Compatibility Kit (TDK) tests. It would be realistic to follow the model adopted by Oracle TopLink of supporting the JDO APIs as closely as possible within the constraints imposed. Braig and Gemkow in their "The BonSai Principle" article demonstrate a similarly cut-down implementation, supporting only parts of the API (e.g., no query language) based on AspectJ. However, their implementation does at least run within the confines of a MIDP environment.

It may be that the best that can be achieved given current constraints is an architecture where remote proxies are used in combination with the JDO implementation running on the server. This should not be seen as a purely negative situation, since any enterprise-level application is likely to require distributed access to applications and data that would not benefit from very heavyweight mobile clients. Rather, the bulk of the system's services and data would be server-centric, with only small subprocesses and data caches being devolved down to individual devices. In this type of architecture, a JDO implementation could encapsulate the distributed cache management required and assist in the transparency of developing distributed applications.

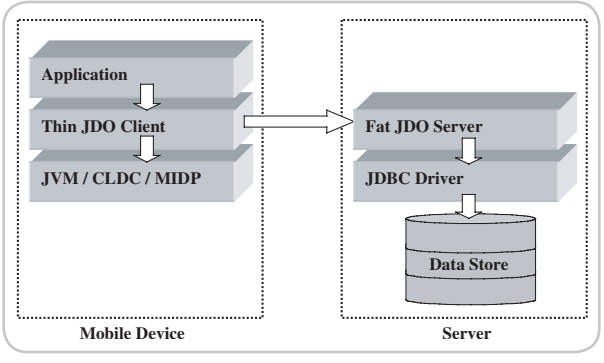


Figure 3 Client/server JDO architecture for MIDP

JDO Architectures for MIDP Devices

A MIDP application can benefit from using JDO in two different scenarios. In the first scenario, the data store is located on a central server and a MIDP application running on a client mobile device uses JDO to access the remote data. The JDO layer in this case functions as a high-level wrapper for communication with the remote server's data store. In the second scenario, JDO is used to manage local data on the mobile device. The JDO layer in this case functions as a high-level wrapper around the MIDP RMS mechanism. In both scenarios, JDO provides a similar, easy-to-use API for managing the data, whether local or remote, using the application domain model. Each scenario, however, requires a different design solution.

Client JDO Architecture

A common technique in the design of JDO implementations is to build a JDO-JDBC bridge. An implementation that functions as a wrapper layer around JDBC can easily support a wide range of data sources. Figure 2 shows a naive architecture for using a client JDO implementation of MIDP based on this approach.

In this architecture, the data store is managed by a remote process that runs on a central server. A JDBC driver on the mobile device communicates with the remote server process, while the JDO implementation functions as a layer between the MIDP application and the JDBC driver.

This architecture has several disadvantages. First, because the entire JDO implementation, including the JDBC driver, is running on the mobile device, substantial memory resources are required. Second, standard JDO operations such as converting queries from JDOQL to SQL (i.e., from the query language of JDO to a format that a JDBC driver can handle) might be too slow on the standard CPU of a mobile device. A further issue is the limited number of JDBC drivers that are available today for MIDP.

Client/Server JDO Architecture

Because of the problems associated with deploying the entire JDO implementation on the mobile client, a more realistic architecture would be to develop a client/server JDO framework (see Figure 3). In this architecture the JDO implementation is split between the mobile device and the server. A "fat" JDO process runs on the central server, communicating with the data store using a JDBC driver.

On the mobile device, the MIDP application uses a thin JDO client library to access the data store. Only operations that must be implemented on the client side are included in the thin JDO client library, and all other operations are implemented by the JDO server. For instance, converting queries from JDOQL to SQL should be done on the server side due to the limited resources, in terms of memory size and CPU speed, of the mobile device.

To keep the footprint of the client as small as possible, some features of JDO, such as supporting local queries on memory collections (which, if supported, would have to be implemented on the client), may be omitted. This would mean the loss of full JDO compatibility.

The client/server JDO architecture is very flexible and can be tailored according to requirements. For example, instead of using a two-tier architecture, in which the data store and the JDO server are located on the same machine, a three-tier architecture can be used, deploying them on two different machines. Another design change is required when a JDBC driver is not available, for example, if the data store is an object database. In that case the JDO server is expected to access the data store directly.

Local Storage JDO Architecture

There are many benefits in storing the application data on a central server, but in some situations local storage may be preferred. For example, a MIDP application that manages a contact list or a personal organizer should keep the data locally on the mobile device (possibly in addition to a backup of that data on a remote server). This can provide a faster response time and also ensures the availability of data when the server is unreachable because of network problems or maintenance, but the JDO implementation, which acts as a wrapper around the RMS APIs, is complex (see Figure 4).

To understand what is expected from local storage JDO for MIDP, it might be helpful to distinguish between two types of JDO implementations: those that provide JDO support for relational databases by implementing a JDO-JDBC bridge and those that use object databases to support the JDO APIs. Because object databases do not rely on another database system as back-end storage, they must provide all the services that a standard database system provides, including (among others) storage management, lock management, query processing, and transaction support. For example, object database JDO implementations cannot convert JDOQL into an SQL query that is executed by a relational database, but rather have to include their own mechanism for processing and executing queries.

A local storage JDO implementation for MIDP would be very similar to a JDO object database implementation. Such an implementation has to support all the standard services that a database provides and everything has to be implemented on the mobile device. RMS as a low-level storage system is at least as good as binary files, but does not provide the database services that JDBC provides.

A storage solution can be based on allocating the first record in the RMS record set for general database information and an additional record for every object and every class schema. Other common internal database data structures, such as BTree+ for indexes, can be implemented by multiple RMS records (for instance, every node in the tree could be stored in a record). We have to consider whether the local management of such data structures is realistic because of the memory resources that they consume on the device, both for storing the data structures and in the implementation byte code that they add. A more appropriate solution might be to avoid supporting indexes (which are not required by the JDO specification) and to process queries by iteration over all the objects one by one (using RMS filters).

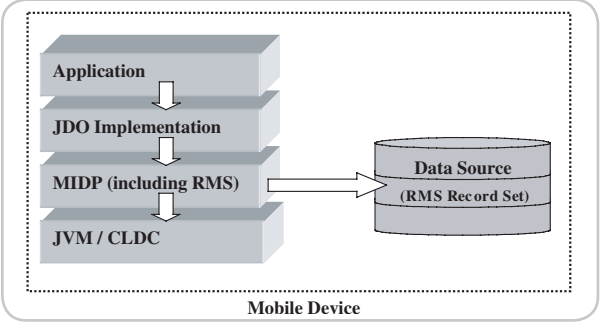


Figure 4 Local storage JDO architecture for MIDP

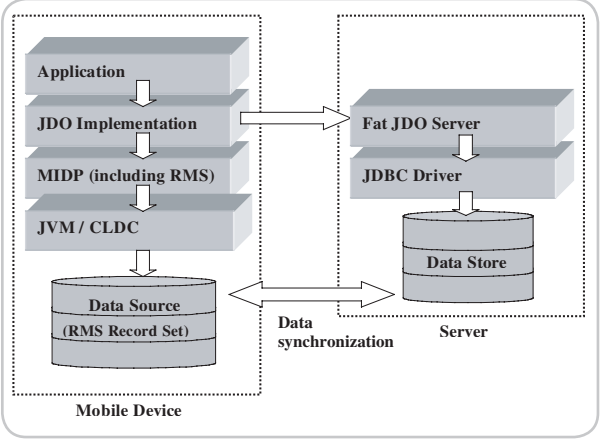


Figure 5 Hybrid architecture

Hybrid Architecture

The most flexible solution for JDO on MIDP would be a hybrid architecture that included elements of both the client/server and local storage approaches. This would provide the benefits of a fully featured JDO implementation running in the server, plus the ability to maintain disconnected local data to maintain quality of service (see Figure 5). Of course, this type of solution is much more complex, since it requires the JDO implementation to manage the distributed data that is being cached on mobile devices. We can expect the development of systems using this kind of architecture to be supported by implementations of the Java synchronization APIs.

Conclusion

The current version of the MIDP specification is an interim set of APIs that reflects a particular point in the development of mobile telephone technology. At present, mobile phone developers must work within the constraints of current devices and work around the constraints of the platform as best they can. Although the limited CLDC/MIDP libraries constrain a number of aspects of Java application development, there are a number of initiatives in place to support applications migrating down to smaller devices, including small footprint XML parsers and databases.

Regarding JDO and whether or not it could be implemented to run on a MIDP device, our conclusion is that

while MIDP alone cannot realistically host a full JDO implementation, a distributed implementation that combines local processing with server support can indeed meet our application needs. Not only that, but such an architecture actually opens up a more challenging set of options for truly distributed systems that provide for widely distributed data and processes.

The real challenge for MIDP developers is to build applications that not only work locally on a single device but can interact and synchronize with multiple nodes of different types in a disparate architecture. In practice, running JDO on a single device provides few advantages over alternative APIs for data access. However, a distributed JDO implementation that integrated and synchronized data across multiple nodes, encapsulated behind a single distributed object model, could be a very valuable tool.

From our discussion of JDO as an example of serious application development, we can see that developing software for mobilized architectures requires us to consider a range of aspects of design and implementation to identify the optimum configuration. MIDP alone cannot provide a fully featured Java deployment platform, but by playing to its strengths, such as the ability to maintain a persistent local data cache and supporting it with server-side resources, it opens up a range of new opportunities in software development.

References

- Kochnev, D., and Terekhov, A. "Surviving Java for mobiles." *IEEE Pervasive Computing*, Vol. 2, no.2, June 2003.
- Sun Microsystems. (2003). "JSR-000185 Java Technology for the Wireless Industry 1.0 (Final Release)." Java Community Process: http://jcp.org/aboutJava/communityprocess/final/jsr185/index.html
- *JDOCentral, Developer's Community for Java Data Objects:* www.JDOCentral.com
- Reese, G. (2003). *Java Database Best Practices*. O'Reilly.
- *ProGuard Java Class File Shrinker and Obfuscator. Test results:* http://proguard.sourceforge.net
- *Java Data Objects Expert Group, "JSR-000012 Java Data Objects 1.0.1 (Maintenance Release), 2003, section 5.5.4." Java Community Process:* http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html
- Dubé, J.; Sapir, R.; Purich, P.; and Siegal, E. "Oracle Application Server TopLink - Application Developer's Guide 10g (9.0.4)." Oracle Technology Network, pp. 470–486, 2003: http://download-west.oracle.com/docs/cd/B10464_01/web.904/b10313.pdf
- Braig A., and Gemkow, S. "The BonSai Principle – Persistenz in der Java 2 Micro Edition." *Java Spektrum*. September 2002: www.sigs.de/publications/js/2002/09/Braig_JS_09_02.pdf