AS/400 Java Application Models

Paul Remtema
July, 1998

Table of Contents

Introduction

This document describes four primary AS/400 Java Application Models. Each model represents a different runtime execution environment for server Java applications. The purpose of this document is to give you a mental framework for the structure of your application and to provide a point of reference for discussions on performance, security, and interoperability. Details on each model will be added over time.

This document assumes that you have some AS/400 experience and some knowledge of how Java has been integrated into the AS/400 architecture. The integration of Java below the AS/400 Technology Independent Machine Interface leverages historical AS/400 object technology. The AS/400 JVM provides industry leading server features such as asynchronous garbage collection, optimized direct execution code generation, legacy integration, and multiprocessor scalability. To read more about these concepts refer to

http://www.as400.ibm.com/as400mag/mar98/virtual.htm

The primary models described in this document recommend a thin client architecture with two physical tiers and three logical tiers. The three logical tiers are the presentation layer, the business logic layer, and the data access

layer. The business logic layer may be structured to allow additional physical tiers where necessary. All of the models are optimized for AS/400 server hardware. There are significant hardware price advantages to using these application models because they all run in optimized server hardware mode.

To help the transition of writing server Java applications, this document provides parallels to existing interactive RPG and COBOL application structures. In particular, the traditional interactive Job structure is used as a comparison to the Java application models. The models provide different levels of system services. The comparison to the traditional model helps you decide which model to select. In cases where a model does not provide all of the traditional system services, this document will help you build these services into your application.

Primary Models

There are four primary models:

HTTP servlets - This model exploits the idea of a Java servlet. Conceptually, this model is close to your RPG or COBOL interactive Job model. HTML is like DDS in the sense that it defines the communication protocol and screen mapping necessary for the user interface. The servlet is like your RPG or COBOL program. The HTTP server is like your subsystem. When you write a Web-enabled application, the Java servlet code is running on the AS/400.

Transaction Serving - This model exploits Java's thread and socket capabilities. In this model, each client connects to the AS/400 server directly through a TCP/IP socket or indirectly through AS/400 Data Queues. Threads may be assigned exclusively to clients or they may be pooled and shared between many clients. Some of the questions to be answered with this model are thread pooling, number of threads, number of Java Virtual Machines (JVM's), and the management of resources and transactions. This model provides the greatest potential for transaction throughput.

Domino Agents - Domino is a powerful front office application generator. Each Domino document may have Java Agents defined that manipulate the document. This model would typically be used for a class of applications not readily suited for any of the other models defined in this document.

Distributed Objects - Distributed objects may be the programming model of the future. In this model, objects communicate through an Object Request Broker (ORB) or Java Remote Method Invocation (RMI). You can build a distributed object solution today on the AS/400 using non-IBM products. In a future release, Enterprise JavaBeans will become an IBM solution. If you choose to start building distributed objects today, this document will give you some hints on how to structure your application in anticipation of distributed objects later.

Alternative Models

There are two alternative models. These models may be used to get started with Java. These models have been documented in other IBM publications and are listed here for completeness.

Java on the PC - The AS/400 ToolBox for Java provides legacy program and data access from a client PC. This model does not require Java on the AS/400 server. Direct data access from a Java applet through the Toolbox JDBC driver or Record I/O driver provides an initial Java environment and allows you to develop the presentation layer in Java on the client. For decision support applications, direct database access from the client may be required. An alternative to utilizing the ToolBox on the client is the use of dynamic DDS screen conversions to HTML and Java AWT. Various tools now exist to map the 5250 display I/O processing generated by an RPG or COBOL application into a Java GUI. The mapping code, (also written in Java) runs on the AS/400. Screen mapping does not require any modifications to the application.

Remote AWT - The AS/400 JVM supports a unique Remote AWT feature. This allows you to develop Java GUI and run the GUI on a PC with the logic and data access on the server. Performance improvements are planned for this model. In the future, a "distributed object broker" may be added that would significantly improve the performance of this model. The execution environment for RAWT fits with the traditional models since the JVM and surrounding Job are serving a single client. Threads in the JVM may be used to serve multiple windows for a single client. The one client JVM model is a very easy migration for existing RPG and COBOL applications. This model is intended for administrative applications which do not require high transaction rates.

The alternative models should be evaluated in terms of the long term primary models. They are excellent starting points and may be the final structure for some classes of applications.

Back to top

Distributed Computing Paradigm

There are many reasons to consider Java on the AS/400. Some of these advantages include

- Productivity through Objects
- Platform Independence
- Programmer availability
- Modern user interfaces
- Distributed computing

The approach in this document is to stress the productivity of Java from a *distributed computing* paradigm. The OO virtues of Java and the other business virtues listed above are well documented in other publications. For a good reference to the productivity gains achieved through Object Oriented programming in a business environment, check out the following URL:

http://www.midrangecomputing.com/mc/98/05/19980510.htm

Applications are moving to the Web and Distributed Computing. Java has significantly improved the ease of development of these paradigms. In fact, it is the Java language that is increasing the rate of deployment of distributed object models. Java manages sockets and URLs the way RPG manages files. It's a natural part of the language. For a good description of the "Object Web," look at the following *Byte* magazine article. It's far-reaching and provides a compelling reason for moving to Java.

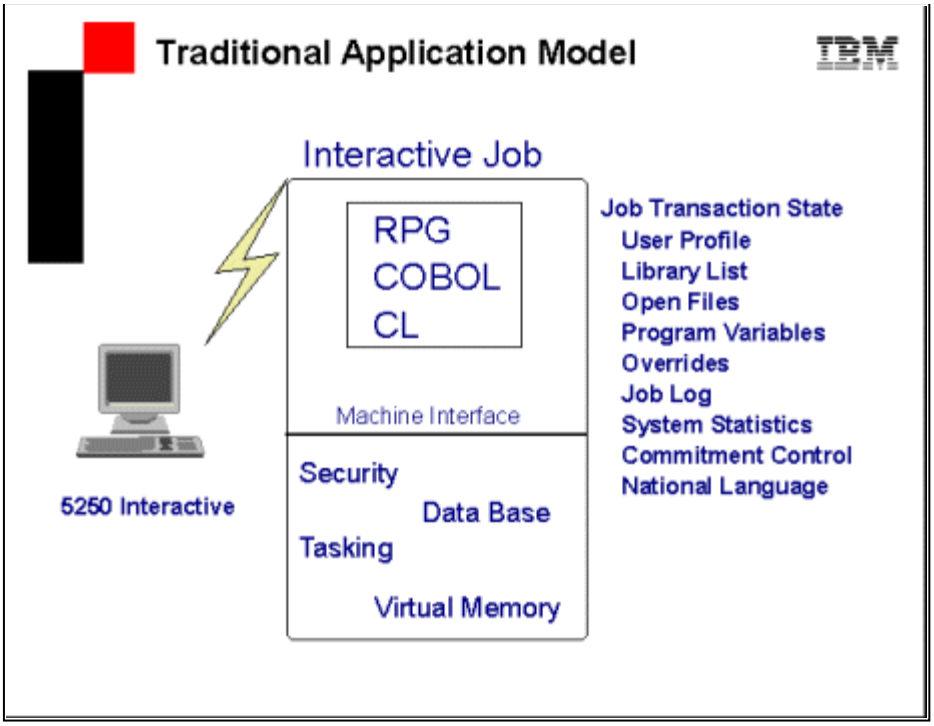
http://www.byte.com/art/9710/sec6/art3.htm

Back to top

Traditional Model Comparison

This section draws a parallel between traditional AS/400 interactive programming and the Java Application Models. The reason for drawing this parallel is to explain some of the default system services that have been added to the AS/400 over time. Not all of these services are available in the Java application models described in this document. This document will try to explain how you can fill the gap between these historic services and the Java application programming models. Hopefully, this section will help you map the Java world into a familiar framework.

Everything on the AS/400 runs in a Job. Jobs provide a large amount of default system management. For example, open files are connected to a Job. If a Job ends, the files are closed and commit processing occurs implicitly. Security is also assigned on a Job basis at user sign-on or during Batch Job initiation. One of the important Job types is an *INTERACTIVE Job. This type is assigned by the system whenever a 5250 device is connected to the Job. By definition, an *INTERACTIVE Job is a "non-server" Job. This figure describes some of the Job attributes provided today on the AS/400.

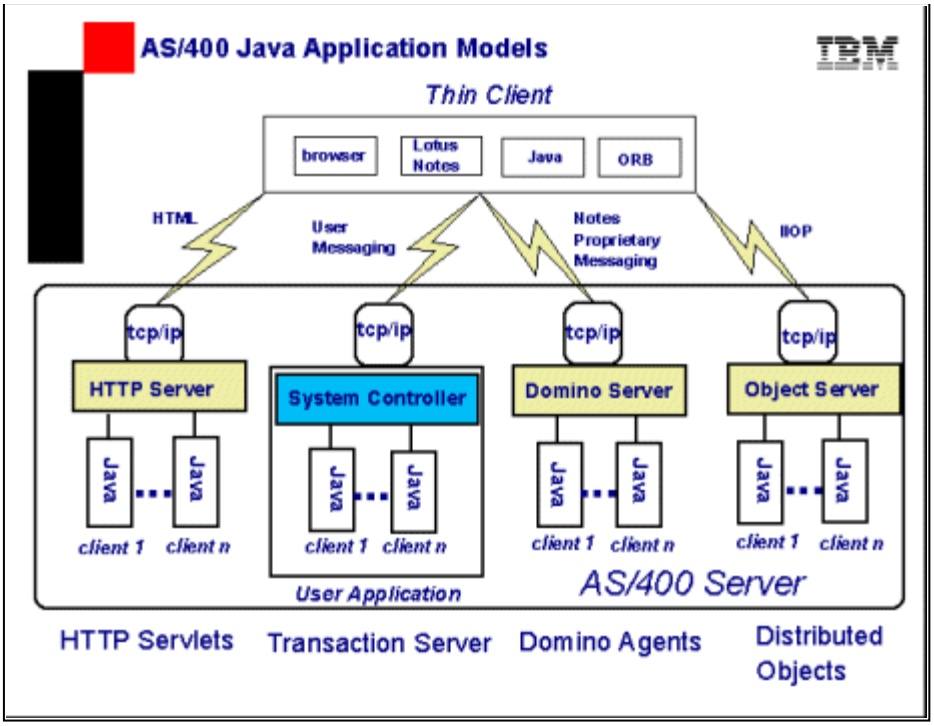


In the traditional model shown above, the majority of the processing is handled within the context of a single Job. Unlike other systems, AS/400 database access and security checks are made within the same Job (Machine Process) for traditional applications. The Java application models change this paradigm by distributing the work for a single application across multiple Jobs. For example, when you access DB2/400 from Java using JDBC, a separate server Job is used to handle the database access. The transaction state list shown in the diagram above should help you consider how each of these system services is being handled by the Java application model you are using. The Enterprise Java Beans server will handle all of the listed services above and should provide the best environment for server side Java applications. An Enterprise Java Beans (EJB) server is not available on the AS/400 at this time. Application level code will need to handle some of these traditional services until EJB is available.

Back to top

AS/400 Java Application Models

The following diagram shows the conceptual framework of the four application models. Each model provides a different level of system services. The distributed object model can be implemented in stages. Early stages of the distributed object model, as well as the other models, will not provide all of the system services provided by the traditional model. Later releases of the Enterprise Java Beans server will provide an environment that is equal to the traditional model.



Thin Client

Thin client, in the model diagram, refers to a logical first-tier concept. A thin client may be implemented on either a Network Computer or a PC. Thin clients simplify the tier one management by reducing the administration necessary to keep the client programs current. Thin client models use the client for presentation services. Business logic and data access remain on the server. In this case, thin client assumes that the presentation data is being moved from the client to the second tier through either a user message or parameters passed on a distributed call. One advantage to this model is that the client is not dependent on specific database semantics and architectures. The client is very platform-independent.

Traditional RPG and COBOL programming on the AS/400 makes use of direct database access. In the traditional model, the data is close to the logic and the logic is close to the display. It is possible to maintain this model by directly accessing the database from the client through the data access classes provided by the AS/400 Toolbox for Java product. The use of JDBC in an applet is an example of this type of processing. Executive information and decision support systems may require this structure. Thin client does not preclude the use of direct database access from the client. De-coupling the client from the database, however, is a paradigm shift that you should consider as a technique for increasing client platform-independence.

HTML presentation is sufficient for many business applications. With this form of presentation, an HTML browser is sufficient. If the HTML forms are not sufficient, the next logical step is to use Java applets to enhance the user interface. Currently, Java AWT takes more CPU on the client that traditional Windows GUI. If you are planning to use extensive Java AWT on a Network Station, a Model 1000 (200 MHz Power PC) is recommended.

An Object Request Broker (ORB) is part of the CORBA distributed-object architecture. PC-resident ORBs are now available from companies such as InPrise (Visigenics) and Iona. These ORBs allow platform-independent, remote object calls to the server. Call-backs from the server to the tier-one client are also allowed. This can provide some interesting dynamic client modifications handled transparently by the server. PC based ORBs are now being packaged with Web browsers. They can also be downloaded transparently from the server (Called and Orblet) by an Applet running on the client. An example of a client ORB is VisiBroker from INPRISE Corporation.

http://www.inprise.com/visibroker/papers/its/

At the time of this writing, negotiations are underway to provide AS/400 server-side ORBs. These should be available soon and will complete the "client side" options shown above.

Back to top

Threads

V4R2 OS/400 introduced kernel threads. Java makes use of kernel threads by assigning each Java thread that you create to a kernel thread. OS/400 kernel threads are managed by assigning each thread to a low-level system task. Threads execute in parallel and make full use of multi-processors.

In the AS/400 Java application models, threads are used by the server controller to improve system response time and to maximize the use of system resources. In the **HTTP servlet**, **Domino Agent**, and **Distributed Object** models, the Java code that you write does not require the use of threads. In these models, threads are being used by the server products to balance the system load and to invoke your Java code. Assigning a user to a thread and managing the security associated with each user is the responsibility of the server. It is not recommended that you use multiple threads in these models.

In the **Transaction Serving** model, your application handles the thread control. In this model, you need to decide how to assign users to threads, how to pool resources like JDBC connections, and how to manage security. The Transaction Serving model provides the most flexibility for performance tuning and load-balancing, but also requires you to write additional system-level functions.

Back to top

HTTP servlets

A servlet is Java code that is invoked by an HTTP server as part of a special HTML servlet tag request. Servlets run in an environment created by the HTTP server. The servlet classes define this environment and become the mechanism for your Java code to communicate with the client. A good reference for servlet architecture is the Sun Web Site at:

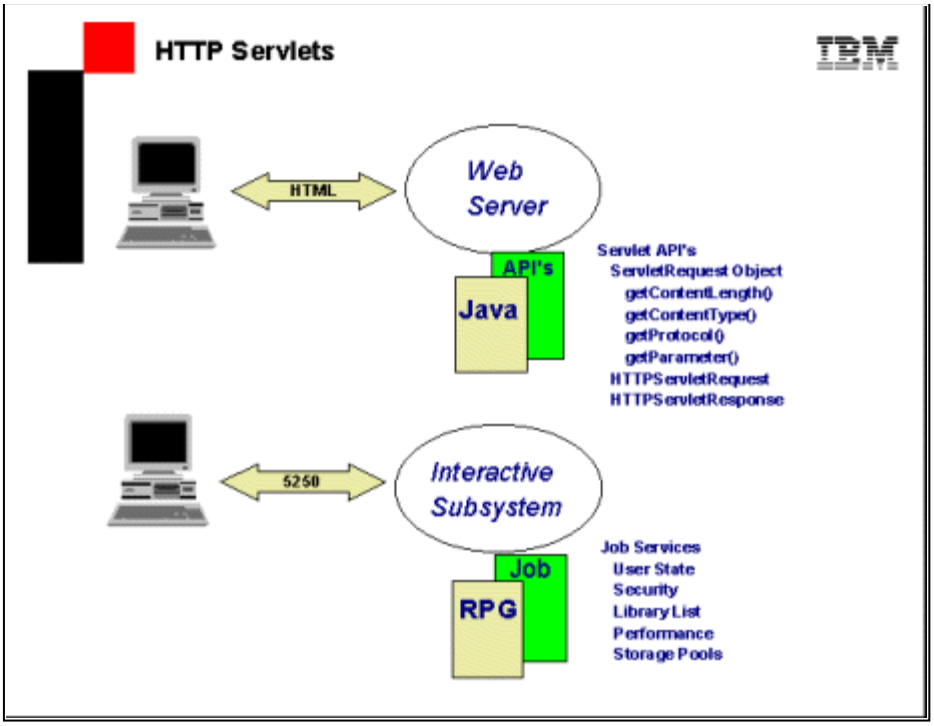
http://jserv.javasoft.com/products/java-server/documentation/webserver1.0.2/servlets/architecture.html

The servlet APIs are part of the Java Standard Extensions implemented by any Java servlet-enabled HTTP server. On the AS/400, a variation of the Domino Go Webserver with Servlet Express will be provided later this year. The API documentation for this product may be found at

http://www.ics.raleigh.ibm.com/dominogowebserver/

Other products, such as Sun's **Java Web Server,** have been running internally on AS/400 systems with OS/400 V4R2. The Java Web Server is located at

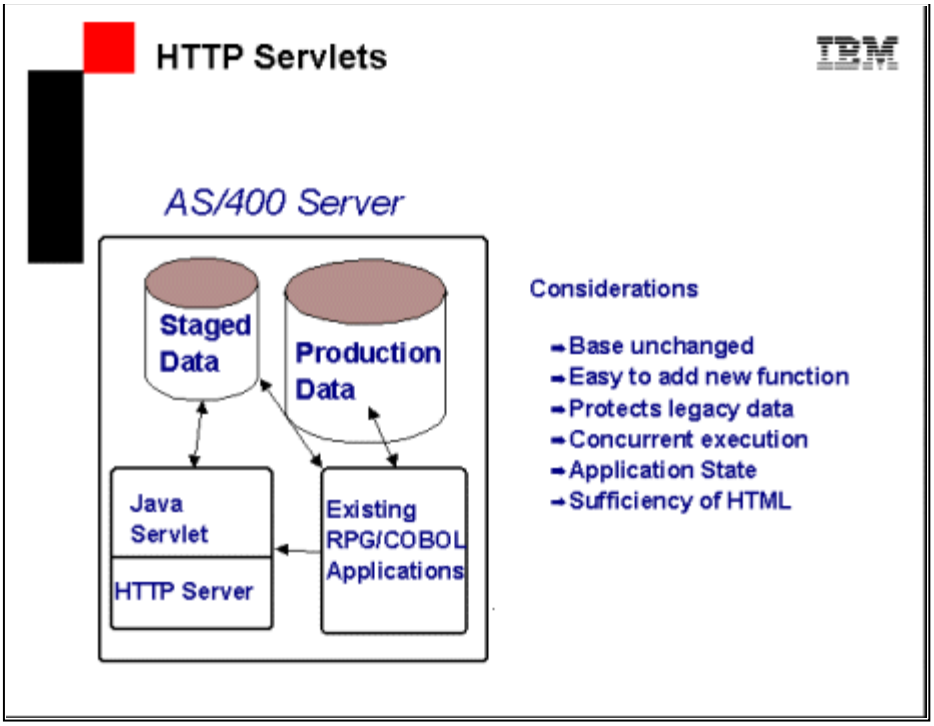
http://jserv.javasoft.com/products/webserver/index.html



The diagram above shows the basic concept of an HTTP servlet. Servlet APIs define the context and services under which the servlet is running. You can compare these APIs to CGI-BIN services. The main difference is that Servlets run in the HTTP server Java Virtual Machine. This improves performance over CGI-BIN and allows servlets to communicate between multiple invocations. A comparison is shown with an interactive RPG program. In this diagram, the Job structure provides the containing services for the RPG program. The analogy (though somewhat stretched) is that the HTTP server and APIs provide the containing services for Java servlets. Servlets allow you to conceptually maintain the interactive programming paradigm while Web-enabling your applications.

HTTP Servlet Example

The following example is an excellent use of servlets. Web ordering is now a common paradigm. Just look at http://www.1800flowers.com/flowers/welcome.asp. This is a good example because it shows how you can add functions to existing legacy ordering systems without moving your entire application to Java. In this example, the order is taken and passed to a staging area for processing by the legacy system. This reduces the requirement for immediate order confirmation and transaction load on the system. Usually within a few hours, the order is processed by the legacy system and an e-mail is sent back to the user through a second Java application.



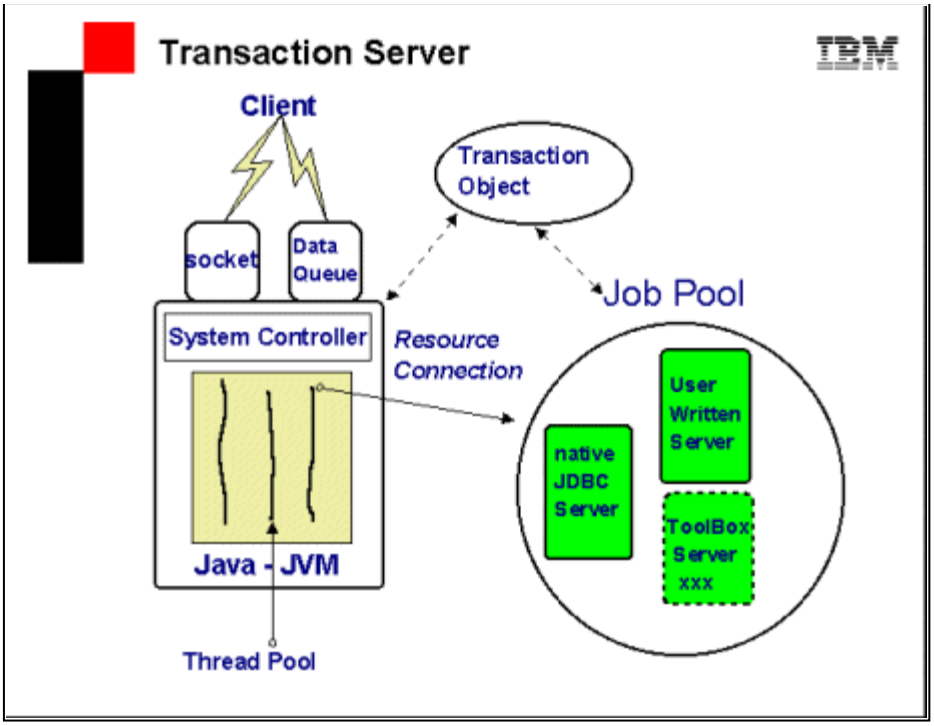
Back to top

Transaction Server

A transaction server is a Java application that manages application-defined transaction messages. Either AS/400 Data Queues or Java sockets are used to communicate messages between the tier one clients and the server. As transaction messages arrive, the application either creates a new thread to handle the message or assigns an existing thread from a thread pool. A fundamental concept of client/server implementations is the idea of resource pooling and funneling. If a server maintains a unique connection and set of resources for each active client, the total number of resources active on the server can become prohibitive. For example, 10,000 active clients each requiring 10 open files would mean there were 100,000 open files on the system!

Funneling means that the 10,000 active users share the server resources. A ratio of 10-20 to one can usually be achieved, depending on the think time associated with each client. Using resource pooling with a 10 to one ratio would reduce the number of open files to 1,000.

The following diagram shows the basic structure of a Java transaction server.



The primary resources that need to be pooled are threads and ToolBox/JDBC connections. One programming paradigm that has changed with Java is the idea of a "connection." Java is a client/server language. Instead of "opening" a file, you "connect" to a driver. The AS/400 ToolBox for Java provides an AS/400 object that represents a connection. As in "sharing" a file in RPG or COBOL, you may want share these connections to save time and resources.

In the traditional programming model described above, your AS/400 Job kept track of the resources that you were using. With Java, these resources may be distributed between many jobs. Furthermore, as you move to a distributed-object architecture, most of the objects that comprise your application will not be running in one Job. A transaction object is a concept that you can use to keep track of the current state of each transaction as it moves through the system. With the future products provided by Enterprise JavaBeans, transactions will be managed for you. With the **Transaction Server** model, your application will have to manage these resources. You may want to refer to the traditional model description above to remind yourself of the transaction state usually associated with an AS/400 Job.

Transaction Server Example

As an example, the TPC/C order entry workload used to rate many commercial systems has been implemented in Java using a physical two-tier/logical three-tier architecture. On the AS/400, this workload is currently achieving its maximum throughput using a client-to-server thread ratio of 20 to one. Each server thread is generic in the sense that it can handle any of the TPC/C transaction types. Each thread uses one generic JDBC database connection. Twenty clients are therefore getting their work done using one Java thread and one database connection.

The number of threads per JVM will vary with application type. The AS/400 JVM can handle thousands of threads per JVM. From a scalability perspective, however, it is better to design your application with multiple threads and multiple JVMs. Each JVM contains its own heap space. Segmenting application types into unique JVMs may help paging and heap re-use. Thread management and termination may also be easier to manage if the number of threads per JVM is kept to a reasonable number.

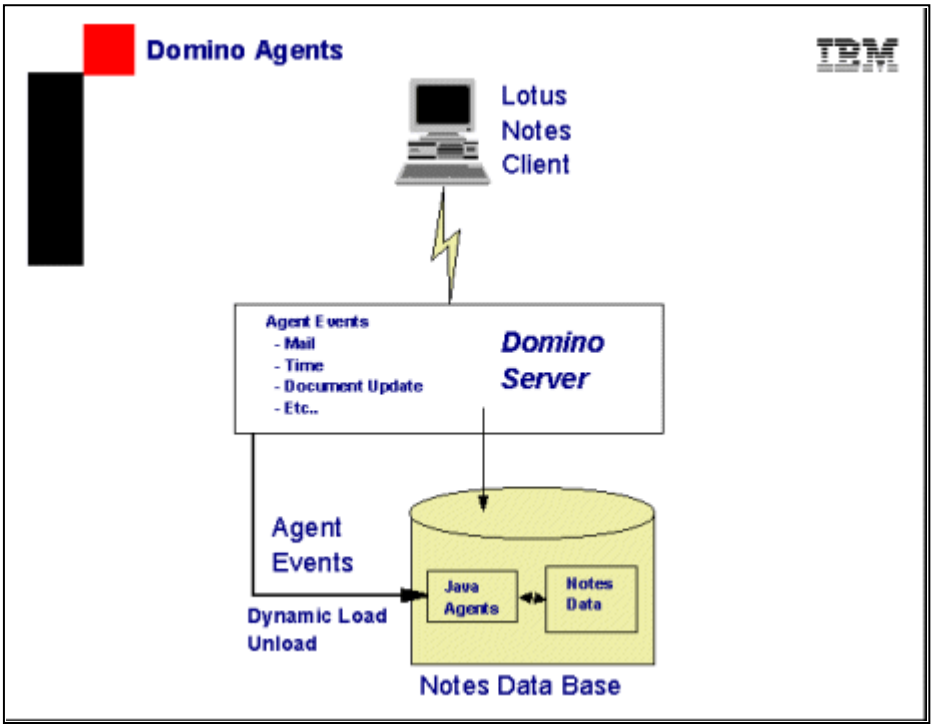
Back to top

Domino Agents

The Lotus Domino Server defines a feature know as agents. An agent is an event-driven program that operates on Lotus Notes data. You can write Domino agents in Java. The diagram below shows that the Java agents you write

are stored with the Notes database. They are called when an event such as mail arrival occurs. Since they are stored with a Notes database, they can be replicated with other Notes data.

Domino Java Agents have the same set of APIs as Lotus Script Agents. The main advantage of using Java over Lotus Script is that you may already be using Java as an application language. When you write Java agents, you should consider them single-threaded. The Domino server is using threads to manage the clients connected to the server. The Domino server will create a thread to run your Domino agent.



Back to top

Distributed Objects

Distribute Objects is an application model that represents local and remote objects in a transparent manner. The "objects" represent business logic and data. The data can be either temporary or persistent. In its most general form, distributed objects can be represented by any language, including non-OO languages such as RPG. The local/remote transparency can be accomplished through directories and routing services outside of the application source code.

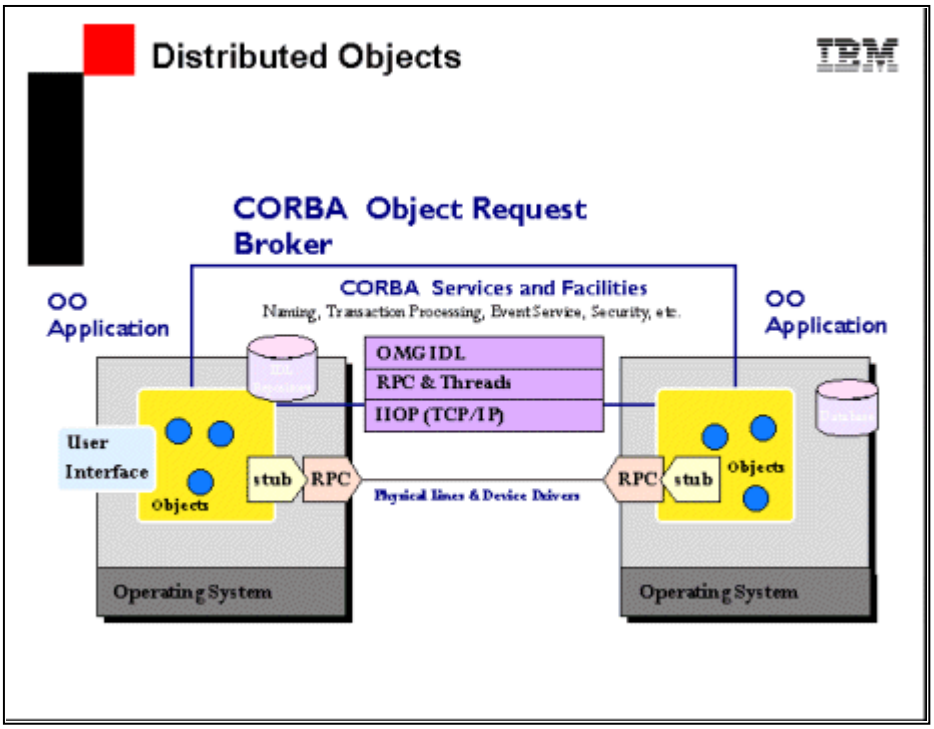
Distributed Objects support is not fully implemented on the AS/400 at this time. You can anticipate this application model, however, by using either Java Remote Method Invocations (RMI) or AS/400 Data Queues accessed through the AS/400 ToolBox for Java **BaseDataQueue** class. By structuring you application around either of these concepts, you can move to an industry standard Distributed Object architecture at a later time.

CORBA

The Common Object Request Broker Architecture (CORBA) is an important distributed-object architecture. It has been designed by a consortium called the Object Management Group (OMG) that includes over 700 companies, including IBM. A fundamental part of the CORBA architecture is the ORB middleware. At the time of this writing, ORBs are being ported to the AS/400. You should look for these solutions later this year.

The following diagram shows the basic CORBA concepts. An Interface Definition Language (IDL), defines the parameter and object structure associated with each distributed object. An IDL compiler maps this language into stubs on the client and server systems. The stubs marshal the parameters and communicate with the remote object. Local optimizations make it possible to use this architecture for calls within one system. For details on CORBA, refer to the OMG Web site at

http://www.omg.org

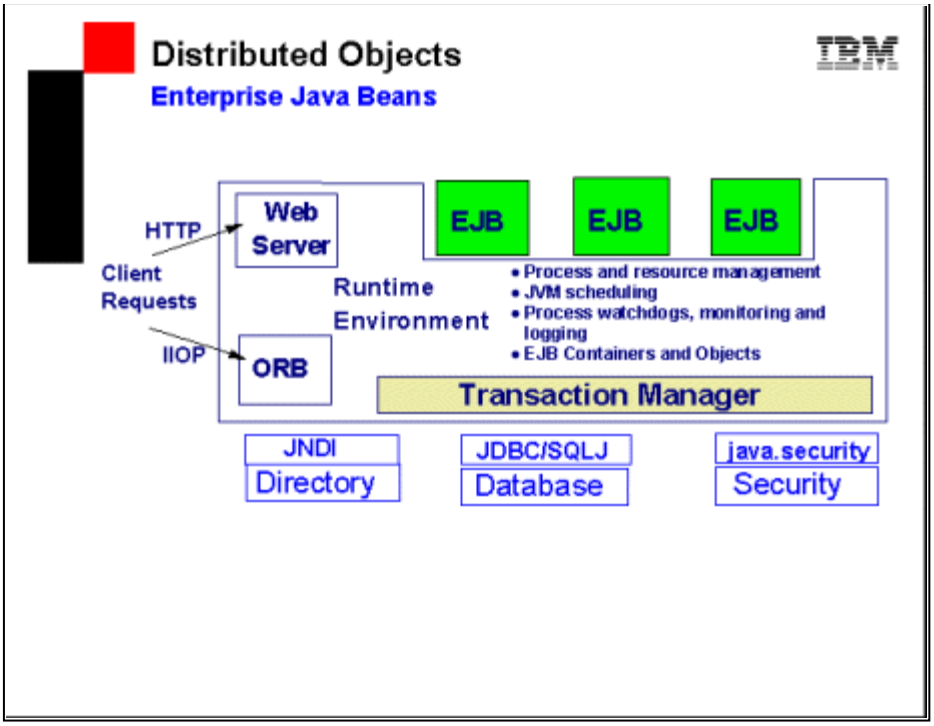


Enterprise Java Beans

Enterprise Java Beans is an industry Java server architecture. EJB includes middleware products that will enable you to write reusable components using Java's bean architecture. Distributed objects will be supported through an ORB and distributed directory services. When you write EJB Java code, you will be communicating to a transaction manager through a set of APIs. These APIs will define the context in which your Java code is running.

When you write Java programs using the EJB model, it will be similar to your Job services provided in the traditional model described above. The EJB model describes the idea of a *container.* You can think of a container as somewhat similar to an AS/400 Job. Since the EJB server is using multiple threads to improve server response, you should not use multiple threads inside of an EJB. For more information on EJB refer to the following site:

http://java.sun.com/products/ejb/docs.html



Back to top

Summary

Java programming on an AS/400 represents a major paradigm shift from the traditional RPG interactive paradigm. The **HTTP servlet** model provides the most natural transition from interactive AS/400 programming. The**Transaction Server** model requires you to understand threaded programming, but it will provide the highest transaction rates for high volume transaction applications. The **Domino Agents** model addresses a new set of front office applications. In this model, the Java programs are background agents that contribute business logic and document review capabilities. The **Distributed Object** model is not complete. By leveraging some of the concepts and early middleware products, you can simplify your application structure and be ready for additional distributed-object solutions next year.

Back to top

Interoperability

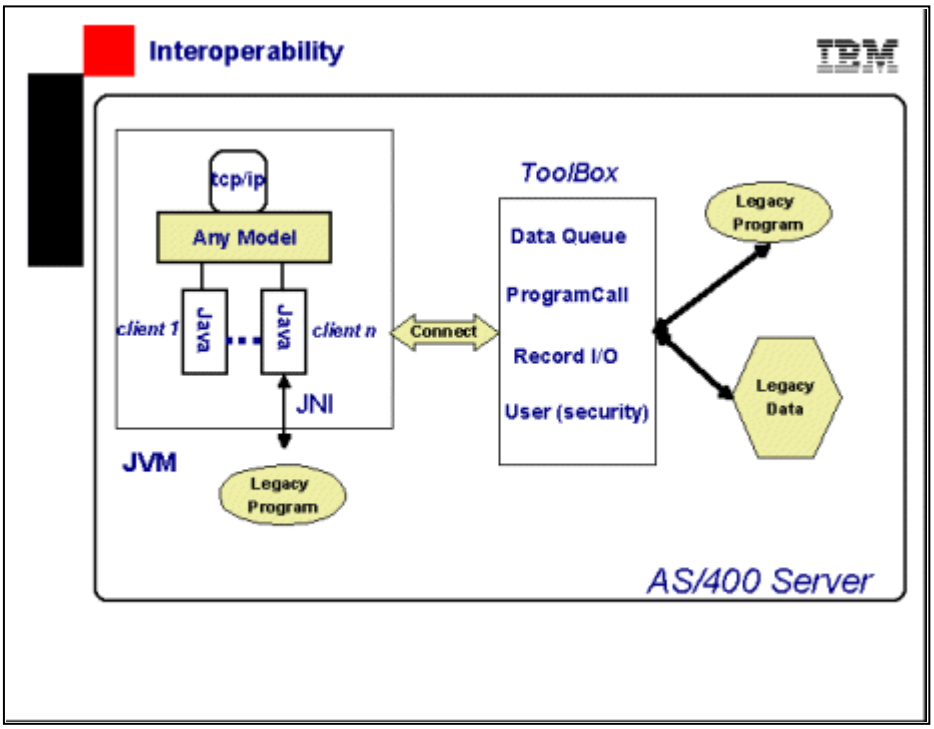
Java applications and legacy applications will co-exist for a long time. This chapter describes techniques that you can use to call between legacy programs (RPG,COBOL, and CL) and Java applications. In this chapter it is important to think long-term. If you are planning to move to a distributed-object application model, inter-operability between legacy programs and Java should be handled through an architected distributed program call. CORBA and Enterprise JavaBeans are based on architected distributed program calls. One of the recommendations in this section is to simulate this distributed calling model through the use of Data Queues. Calling within one system through an architected distributed object call to RPG or COBOL programs is not available at this time. You can save yourself some application restructuring by simulating this model through Data Queues and then replacing your implementation with an industry-standard implementation later.

AS/400 ToolBox for Java

The AS/400 ToolBox for Java product provides Java classes that may be used to inter-operate with legacy programs and data. For this chapter, you may want to refer to the ToolBox documentation, which can be downloaded from

http://www.as400.ibm.com/toolbox

The following diagram shows a model-independent view of the AS/400 Toolbox for Java classes accessing legacy data and programs. In V4R2, the AS/400 Toolbox classes use the Client Access drivers and connect to these drivers through the same TCP/IP connection used from a client PC. Server-side optimizations are being added in a later release. In particular, the Record I/O, Data Queue, and User Space classes will stay directly on the Java thread and avoid the Client Access drivers entirely. The Program Call and Print Server Classes will be optimized to use an internal local socket connection to the Client Access drivers. These optimizations will provide significant performance gains over V4R2 and will be enabled without requiring modifications to your application. JDBC and Integrated File System (IFS) access on the server should use the native JDBC driver or java.io. Toolbox classes for JDBC and IFS access should be used from a client PC or from server to server when the data is remote.



Back to top

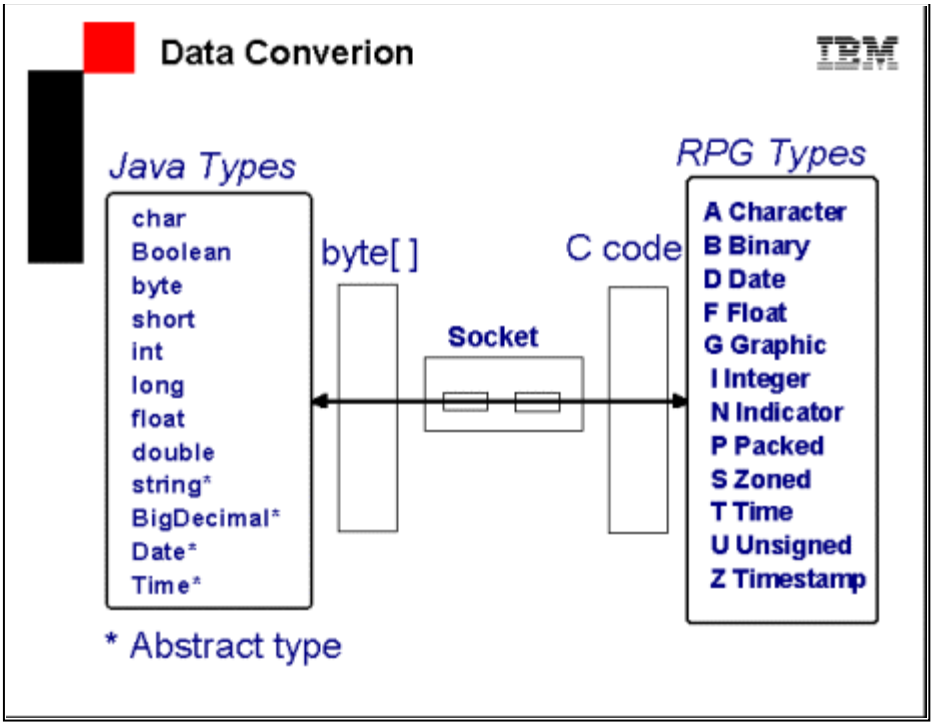
AS/400 ToolBox for Java Concepts

This section describes a few basic concepts that pertain to any form of inter-operability with Java through the ToolBox classes.

Toolbox uses the idea of a Connection. An **AS400** object is used to represent that connection. When you are writing Java code on the AS/400 and accessing the ToolBox Client Access drivers, you first connect to one of the Client Access servers and then access the programs or data. Connection time can be improved by either pooling the connections (saving them for later use), or using pre-started Jobs for each Client Access server Job. In a sense, managing connections is like managing open files in legacy programs.

Some Java data types do not match AS/400 data types. The AS/400 JVM stores string data in the JVM as two-byte Unicode data. This is a universal data type that allows Java programs to exchange data in a more global manner. Most character data on the AS/400 is stored in one-byte EBCDIC format. As a general rule, you must convert string data in Java to EBCDIC data when using the ToolBox access classes. ToolBox provides many classes to help in that conversion.

The following diagram shows all of the Java data types and all of the RPG data types. If you called a legacy program such as an RPG program, you would need to use the ToolBox conversion classes for every call and return. In the diagram below, the Java char[] array is used to hold AS/400-specific data types. The conversion occurs in the Java program before and after the call is made.



Back to top

Java calling Legacy Programs

If you have followed the development of the application models described in this document, you will notice that all of the models start with Java in control of the application. From this perspective, it is important to access legacy data and programs by calling *from* Java to the legacy systems.

Here are the three most important techniques:

- ToolBox **ProgramCall** Class
- ToolBox **BaseDataQueue** Class
- Java Native Interface

ToolBox ProgramCall

The ToolBox **ProgamCall** Class uses the Client Access Program Call driver to call any AS/400 program object. This is an easy way to access legacy programs. The ProgramCall class can be used for relatively long running programs, such as print routines or batch processing. Because this call is relatively expensive compared to an RPG to RPG program call, you should not use the **ProgramCall** Class for frequently called small programs.

ToolBox BaseDataQueue

The ToolBox BaseDataQueue Class can be used to communicate between Java and legacy programs. An AS/400 Data Queue object needs to be created, and you need to establish some message-passing conventions to use a Data Queue. Although this is not as easy as the **ProgramCall** class, the performance is better.

Data Queue communication potentially allows you to structure your application with distributed objects in mind. Since a Data Queue can be accessed from Java as well as legacy programs, you can stage the replacement of legacy programs with Java programs over a period of time. Since your legacy programs may not be set up to talk to a Data Queue, you may have to write some wrapper code that handles the queue communications and then invokes your program.

One added benefit is that the access to the Data Queue from your Java program will be optimized by the ToolBox product to avoid the Client Access driver when the Java program and Data Queue are on the same server. This

local optimization will significantly improve inter-operability.

Java Native Interface

Currently, the Java Native Interface (JNI) works for ILE/C and ILE/C++. It is used by the Java Virtual Machine to define native methods that complete the implementation of the Java Virtual Machine. Application writers are not expected to make heavy use of JNI. It is a low-level interface that should be used in performance-critical situations.

The Java Native Interface is an architected Java interface that is platform-independent. You can access native programs by defining a native method in Java and then using the JNI functions. When you call a native method in Java, your program is running on a Java thread. You must be aware of the thread-safe features of the programming language and system APIs that you use. In particular, RPG, COBOL, and CL are not thread-safe languages and should not be called from a native method unless you can guarantee that thread safety is not required.

You can read about AS/400 threads at

http://www.softmall.ibm.com/as400/threads

You can read about the JNI architecture at

http://java.sun.com/products/jdk/1.1/docs/guide/jni/index.html

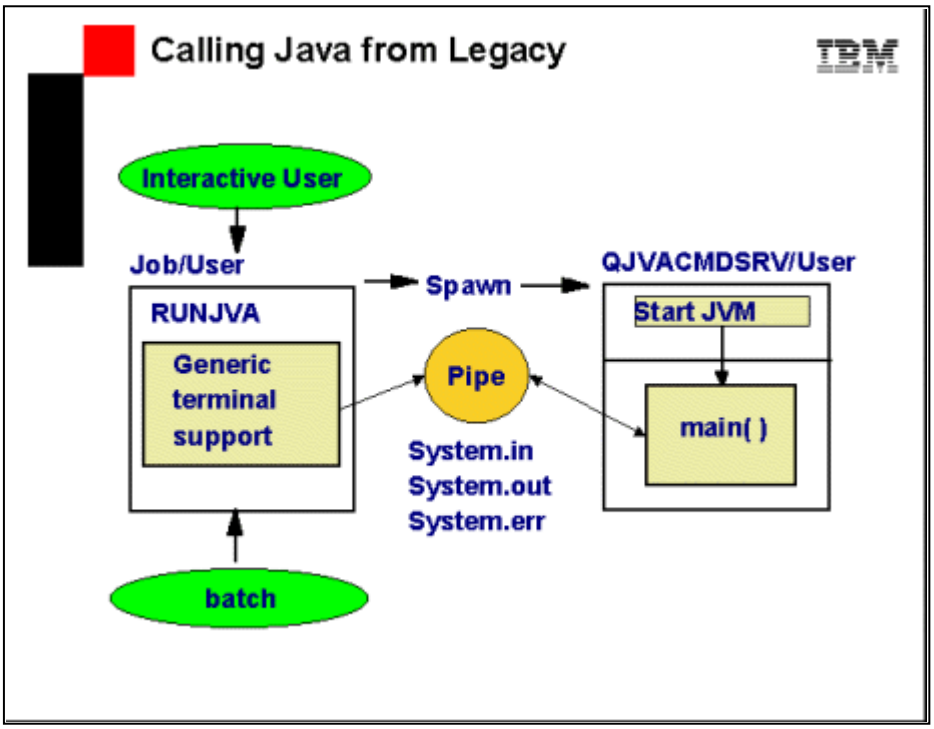
Back to top

Legacy Programs calling Java

There are two techniques for calling Java from legacy programs:

- RUNJVA or JAVA CL command
- JNI Invocation Interface

Java is multi-threaded and requires a multi-thread capable Job. Not all AS/400 Job types are multi-thread-capable. The RUNJVA or JAVA CL command will create a multi-thread-capable Batch Immediate (BCI) Job to start the Java Virtual machine. The use of RUNJVA or JAVA should be limited to long-running Java programs. The command creates a second Batch Immediate Job and starts the JVM in that Job. The basic structure is shown below:



When RUNJVA/JAVA is used, the calling Job is stopped, waiting for the termination of the main() method in the Java class that you started. Starting the JVM requires loading several system classes in addition to your application classes. This is an expensive operation and should not be repeated on a per transaction basis. If your application requires fast, frequent entry to Java, you should consider using the ToolBox **BaseDataQueue** Class between your legacy application and the JVM. To implement this, you will need to use the RUNJVA/JAVA command from an independent Job (since that Job will now be held), and then communicate through a Data Queue between your legacy programs and Java.

In the diagram above, java.io, which is routed to System.in, will not work if the starting Job was a batch Job. For Batch Job initiation, System.out and System.err will be written to a spool file for that Job. For Interactive Jobs, a virtual, terminal support function has been added to the AS/400. This function routes the Java input and output back to the Interactive Job User.

The user profile and security attributes of the JVM will be run with the original authority of the Job that issued the RUNJVA/JAVA command. If you are using adopted authority, this authority will not be propagated to the JVM through the JAVA/RUNJVA command.

The JNI architecture defines the Java invocation APIs. In V4R2 these APIs were used internally to start the JVM. The servers that control the Java Application Models use the JNI invocation interface internally to call your Java programs from the server.

In v4r3 the invocation APIs are public. You can access these APIs from ILE/C or C++. There is a significant performance improvement in this approach in that the JVM can be re-used in the Job without the overhead of starting a second Job.

Interoperability Summary

Here is a summary of the inter-operability options described in this document.

From Java	
Program Call(Toolbox)	Good for long running legacy access such as print or batch.
System Command (Toolbox)	for "occasional" adhoc CL commands.
Data Queues (Toolbox)	Bi-directional. Use to "emulate" distributed objects.
JNI: Java to ILE C/C++	Be aware of thread-safe issues.
JNI: Java to ILE RPG/COBOL	Future release. Will not work from Interactive Job.
To Java	
Data Queues (Toolbox)	Bi-directional. Use to "emulate" distributed objects.
JNI: C/C++ to Java	Future release. Will not work from Interactive Job
JNI: RPG/COBOL to Java	Future release. Will not work from Interactive Job.
runTime.exec()	Use to invoke QSHELL functions. Best option for starting JVM from JVM
QCMDEXEC()	Use for Dynamic construction of RUNJVA command
RUNJVA	Starts another Job. Calling Job is held until JVM ends.

Back to top

Comments?

This document was written to provide a framework for recommendations on AS/400 Java Application Structures and run-time environments. Java programming on the AS/400 represents an unprecedented paradigm shift from traditional AS/400 programming. Comments on future topics are welcome.

Back to top