9 The best language for server-side applications

Even if application servers make sense, what's uniquely compelling about a Java application server? Does it really make such a difference in bringing a product to market. Java developers are convinced that it does, for these reasons:

- *It is a better "3GL."* Java is a simpler, higher-level alternative to C++, and still scales to larger problems.
- *It is a better "4GL."* Java comes with many useful classes and it's easy to extend it with higher-level, reusable business abstractions.
- *It is ubiquitous.* Java is everywhere the I*Net is.
- *It is mature.* While the AFC/JFC battle wages, server-side Java is solid today.
- *It is robust.* On a single machine, that robustness means these immense reductions in time-to-market for Java developers:
  - It is interpreted.
  - No memory leaks.
  - No memory reference errors (scribble bugs).
  - When multiple components are integrated, failures can be diagnosed more easily -- no segmentation violations.
- Neither is Java fragile on a network. Java provides these development features that translate into money and time saved in bringing a product to production and maintaining it over its lifetime:
  - Its higher-level representations -- *business objects* -- like customer, account, money, etc., can be communicated as objects, that is, *passed by value.*
  - *Changing underlying representations* of objects need not break remote applications.

    - It makes possible the dynamic *loading of new functionality via standard bytecodes*, which means that clients and servers can be specialized or updated at runtime.
    - *Garbage collection* is distributed.
- *It provides a platform for component-oriented computing.* Any Java developer can assemble best-of-breed solutions out of reusable JavaBeans, rather than reengineering new applications monolithically from GUI-to-data formats.
- *It is plenty fast.* Much is made of the performance gap between Java and C/C++ native code, but for a typical distributed business application, 60-80% of the CPU typically goes to the DBMS and another 5-15% goes to the network. So even if your Java application logic is two or three times as slow (and it's not any worse than that), Java will -- at worst -- slow your application by 10-20% today. And Java performance is improving rapidly relative to the native-compiled 3GLs.

9.1 Characteristics of a Java application server

A Java application server brings the full power and ease-of-use of Java to an I*Net application server. This paper identifies the key characteristics of a Java application server and discusses why a Java application server makes developing and deploying production-quality faster and easier.

First, to define what a Java application server is, we examine what a Java application server must do:

- Make it easy to develop and deploy distributed Java applications.
- Scale to permit hundreds and even thousands of cooperative servers to be accessed from tens of thousands of clients. That requires that it must:
    - Be fully ***multithreaded***,

    - Be parsimonious in its consumption of network connections and other scarce resources, and
    - Have no architectural bottlenecks that prevent linear scaling.
- Provide an integrated management environment for comprehensive view of application resources (for example, Java Beans, objects, events, etc.), network resources (DBMSes), system resources (ACLs, threads, sockets, etc.), and diagnostic information (unchecked exceptions, logs).
- Provide transaction semantics to protect the integrity of corporate data even as it is accessed by distributed business components.
- Provide secure communications, including SSL support, access control lists (ACLs), as well as HTTP and IIOP tunneling for transfirewall communications.

These requirements are a minimum for any Java application server. More controversially, BEA claims that a Java application server must also do the following:

- Provide rich, very flexible application architectures without complexity. That in turn requires these features:
    - Secure, transactional DBMS access from the client and/or server.
    - Event publishing and subscription from the client and/or server.
    - Remote component (Java Bean) and object invocation from client-to-server, server-to-client, server-to-server, and client-to-client.
- Support Java industry-standard ways of programming. Wrapping C/C++ APIs with Java veneers is insufficient for a Java developer who wants both the richness of Java and to avoid reliance on proprietary programming models. To satisfy Java developers, the Java application server must also offer these features:
    - Database access via JDBC.
    - Name system and directory access via JNDI.

    - Remote object invocation via RMI.
    - Dynamic application partitioning via JavaBeans.
    - Event management and messaging via JMS.
    - Runtime management via JMAPI.
- Support a variety of clients and allow server-side business components to be re-used as they are accessed via HTTP, RMI, event invocation, etc.
- Be compatible with leading industry IDEs (for example, Borland JBuilder, WebGain VisualCafé, MS Visual J++). The application server's ease-of-use should be exposed via a standard IDE rather than requiring user to adopt an IDE proprietary to the application server.
- Be compatible with industry leading DBMSes (for example, Oracle, MS SQL Server, Sybase, Informix, DB2). An application server should not limit its database support to a single DBMS, or otherwise be DBMS-specific in such a way that limits its scope of use.
- Be compatible with standard Java platforms (for example, Sun, Microsoft, IBM, Digital, HP, Netscape, Novell, etc.). Application servers should not require the user to embrace JVM/JIT technology that is proprietary to the application server.
- Be packaged as embeddable infrastructure so that the application server itself can be easily bundled with third-party value-added solutions.
- Be written entirely (100%) in Java (both the client- and server-side), to insure the following:
    - Complete portability.
    - Complete support for all Java native features like passing objects by value, dynamic update of system and business programs, and.
    - That Java features and capabilities can be exposed naturally and rapidly.

EJB uses RMI-IIOP for communication. The main difference is in that EJB formalizes how to define your remote server classes and interfaces. EJB also defines how to access Application Server functionality. But from a communications point of view the differences between EJB and RIM-IIOP is small. NB. I RMI-IIOP demands a little more formalism than plain RMI, thus the differences between EJB and RMI is larger but still of the same kind.