WebSphere connector architecture evolution

by S. M. Fontes
C. J. Nordstrom
K. W. Sutter

The ability of applications to communicate with resources that are outside of the application server process and to use those resources efficiently has always been an important requirement for application developers. Equally important is the ability for vendors to plug in their own solutions for connecting to and using their resources. These capabilities have evolved over time in the IBM WebSphere Application Server, from the JDBC™ application programming interface to the Common Connector Framework and later to the J2EE™ Connector Architecture, the latter providing functions such as application server inbound communications, life cycle management, and work management. In this paper, the evolution of the WebSphere Application Server implementation of these architectures, their benefits, and their trade-offs are discussed. A preview of an important new architecture, the WebSphere Channel Framework Architecture (a logical extension to the J2EE Connector Architecture) is also presented.

Applications need access to many types of resources. A resource may be located on the same machine as the application or on a remote machine and may have no restrictions on its access or a multitude of restrictions. Access to a resource starts with a *connection,* which is simply a path from an application to a resource that it wants to use (see Figure 1). The process of establishing a connection may be as simple as opening a file descriptor for a file that is located on the same machine as the application, or it can be as complicated as accessing databases that are located on remote machines. The path to the resource may involve going through a number of protocols before finally reaching the resource.

The design and implementation of code that accesses resources can be a complicated and time-consuming task. It makes sense to move as much of the management of connections as possible outside of the application so that application developers can focus on business logic instead of connection logic. The technology for making and using connections has evolved to make getting a connection easier and to improve the performance, security, and robustness of the connection itself.

The following considerations are common to all connections.

1. *Creating a connection can be expensive.*
   Setting up a connection can take much time when compared to the amount of time the connection is actually used.
2. *Connections often need to be secure.*
   Securing the connection is often a joint effort between the application and the server working with the resource.
3. *Connections need to perform well.*
   Connection performance can be critical to the success of an application and is a function of appli-

0018-8670/04/$5.00 © 2004 IBM

cation server performance and resource performance.

4. *Connections need to be monitorable and have good diagnostics*.

   The quality of the diagnostics for a connection depends on information regarding server status and resource status.

5. *Methods for connecting to and working with a resource generally vary for each type of resource*.

   For example, relational databases such as DB2* are accessed via SQL (structured query language), whereas hierarchical databases such as IMS* (Information Management System) are accessed via transactional procedures.

6. *Access to a resource might require a certain quality of service*.

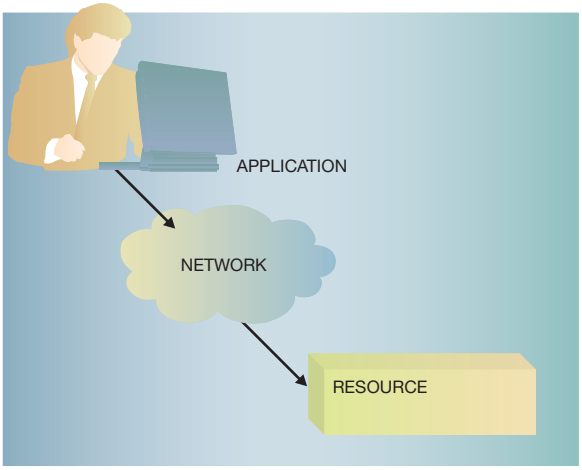
   For example, the application might want the ACID (atomicity, consistency, isolation, and durability) properties that can be obtained when using data in the management of a transaction.

Connection technology has evolved based on two types of access that applications have historically utilized: client application access to Web-server- or application-server-managed resources, and server-managed component access to enterprise legacy resources.

Client access to server-managed resources. Assume a client application wants to connect with a resource that is being managed by a server process that is most likely running on a different machine. The resource might be a Web page that is managed by a Web server process or a component, such as a servlet or an Enterprise JavaBean** (EJB),[1] that is managed by an application server process. These are relatively new types of resources that have emerged with the development of the Internet and are expected to be readily available, a reflection of the Internet environment they were developed for. They are often available to the general public, such as the home page of a business, which is usually available and easily located by anyone.

Some resources may contain all the data and logic that the application needs to access. Other resources may contain business logic that the client wants to execute, and as part of that execution, the resource may need to access data in an Enterprise Information System (EIS).[2] An EIS is a system running in its own set of processes that exists outside of the application server process.

Figure 1 Path from application to resource



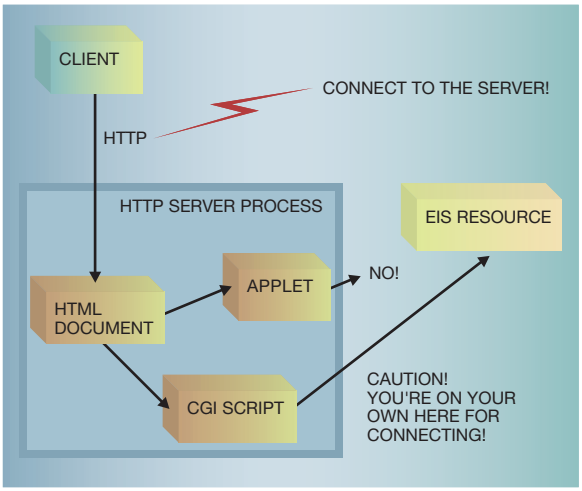
Some examples of server-managed resources are HTML (HyperText Markup Language) documents, CGI (Common Gateway Interface) scripts, applets, servlets, JavaServer Pages** (JSP**), and EJBs (such as session beans, Container Managed Persistence [CMP] beans, and message-driven beans).

Component access to enterprise resources. Enterprise resources are often legacy resources that were developed over time by a business and are external to the application server process. Each type of resource has its own connection protocol and a proprietary set of interfaces to the resource. This means that the resource has to be adapted to be accessible from a JVM**-based (Java Virtual Machine-based) process such as an application server. The resource is usually important to the business and is protected from unauthorized access by strict rules governing how it is used.

In the following subsection, we present a brief review of connector architectures and their evolution.

A brief history of connecting. In early connector architectures, a client machine, using a Web browser, could connect to anything it wanted on the Internet as long as it was a Web page. While this was very useful, there was a need to access resources that existed outside of the Web page. CGI scripts became a popular way to provide some of this access but had limited capabilities and were not an object-based solution. A designer of CGI scripts had to be careful not to introduce means for the user to intentionally or unintentionally initiate actions beyond that which

Figure 2 Simple client access to resources



the script intended. Some of these "holes" were not always immediately apparent until a clever (or unfortunate) user discovered them.

Applets constituted an object-based solution that provided a better way to execute code in a Web page, eliminating security exposures by not allowing access to resources outside of the machine on which the application was running (see Figure 2). Despite these advantages, a solution that would allow access to enterprise assets that was secure, easy to develop, and easy to deploy was needed.

With the advent of the servlet, an object-oriented way to execute code that could access external resources became available. The servlet could be used to access resources managed by a server, such as EJBs, or it could be used to access enterprise resources directly. Java Database Connectivity (JDBC[3]) provided a standardized way to access relational databases (see Figure 3). JDBC** functions were supplemented and enhanced by the Common Connector Framework (CCF) and later by the J2EE** Connector Architecture (J2CA). The remainder of this paper describes the implementations of these technologies in the context of WebSphere, their benefits, and their tradeoffs, and presents the WebSphere Channel Framework Architecture.

Access to relational databases

The introduction of JDBC offered new capabilities in accessing relational databases. In the WebSphere Application Server, JDBC calls are intercepted by the WebSphere Connection Manager (CM), as shown in Figure 4. CM provides a layer between the JDBC calls and the JDBC drivers. This layer provides a common dynamic connection pool scheme regardless of which database is being used, connection cleanup at the end of a transaction, an easy way to configure drivers and connection information, configured connection testing, common prepared statement caching, and common exception handling for "stale" connections.

The common dynamic connection pool ensures that connection pooling is available regardless of whether the JDBC driver provides it. Configuration options for pooling provide a great deal of control to administrators. Administrators can control the minimum and maximum number of connections in the pool, how long connections can be inactive in the pool before they are discarded, how long an application can keep open a connection that is inactive before it is discarded, and how long to wait to get a connection before the attempt is abandoned. Stale connection handling provides an easy way for applications to find out that a connection is no longer usable and implement a strategy for handling this type of situation without having to interpret each error code from the specific JDBC vendor for this condition. Customers can add more error codes to the list of stale connection exceptions to improve stale connection detection.

The use of JDBC directly by the servlet required the application developer to learn JDBC. An alternative to direct JDBC use, EJBs provided a better access model with the "bean" providing a way to access data that takes care of data caching, security, transactions, and persistence scenarios. BMPs (bean-managed persistence beans) could be written by bean developers to encapsulate the JDBC calls. Application developers could then use the BMP to access the data without having to learn how to write JDBC code.

Using beans for data access was an advantage over using JDBC directly because this model could later be used to access other types of resources as well as relational databases, when the appropriate beans became available. For bean providers, CMPs provided an even greater advantage, as the tools embedded the JDBC calls into the bean; therefore, the bean provider no longer had to learn a specific resource's API (application programming interface). Additionally, the EJB container took over the handling of security

Figure 3 Client access to resources via servlets

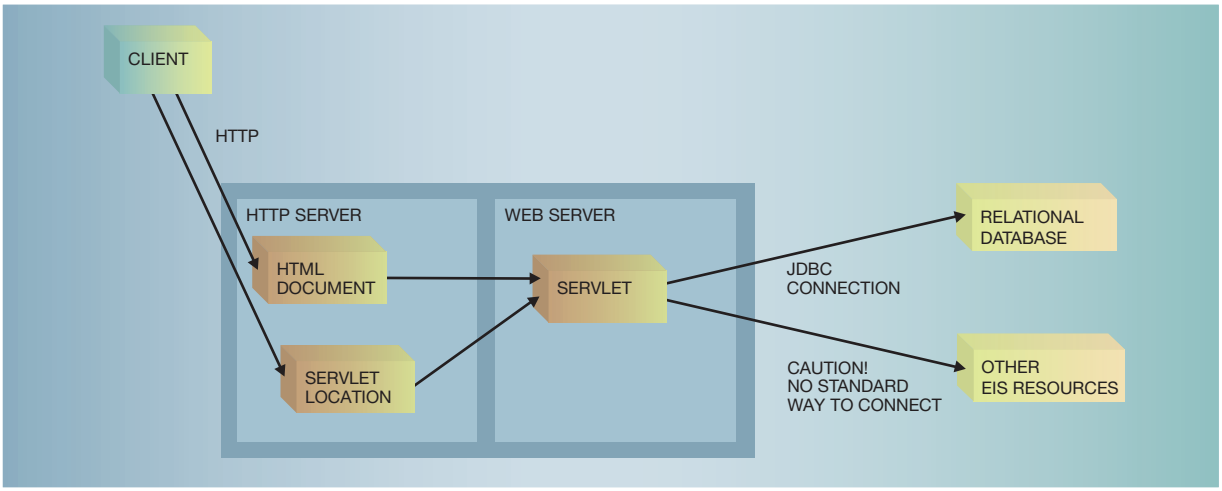
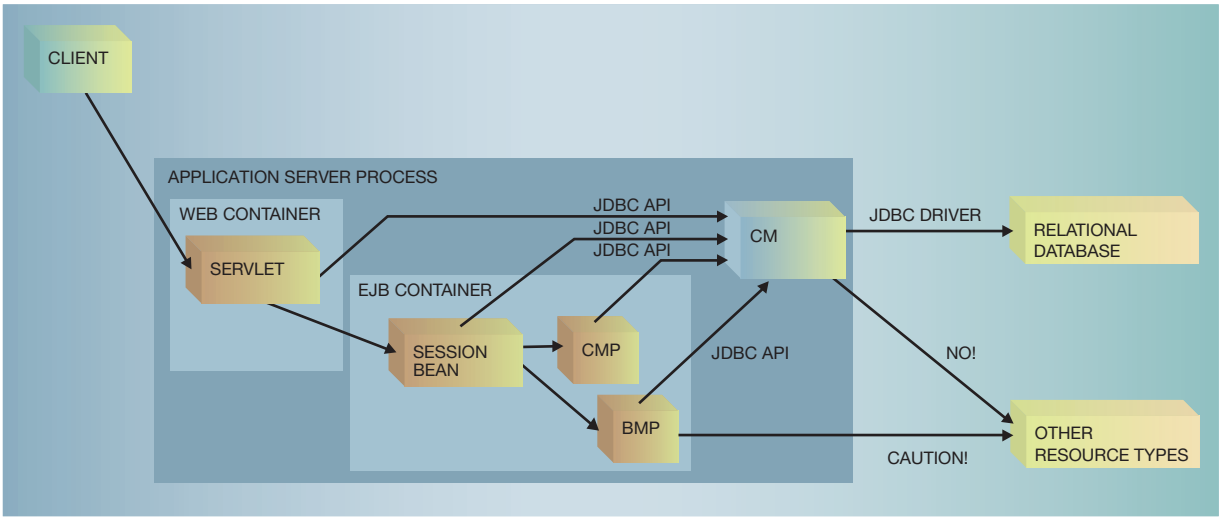


Figure 4 Using EJBs for database access via WebSphere Connection Manager in WebSphere Application Server Version 3.5

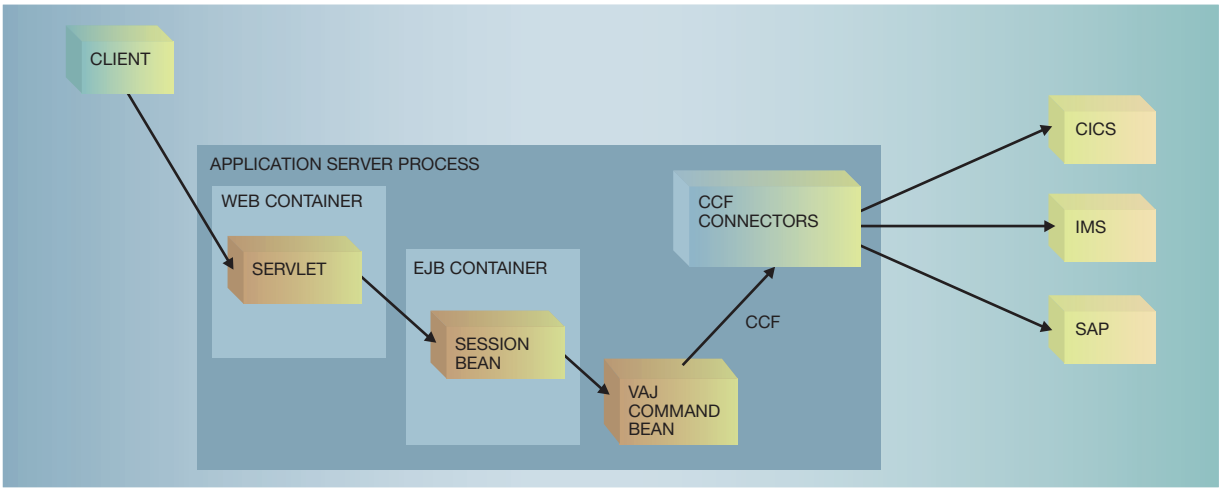


and transactions for BMPs and CMPs and data isolation for CMPs. The Web container also provided some transaction and security capabilities.

Access to procedural and hierarchical resources

JDBC provided a standardized way to access relational databases, but there was still a need for a way to connect to other types of resources, such as transaction-based systems and hierarchical database systems. These resources contained data that had evolved over time with businesses, and many "legacy" applications depended on these data structures. It was not always possible to move functions out of the legacy systems to places that might be more readily available to object-oriented systems. The logical approach was to provide a solution similar to that which JDBC

Figure 5 CCF standardized interface for resource access in WebSphere Application Server Version 3.5



provided; that is, a standardized interface including a common API and a way for vendors to develop drivers that could be used to access resources.

CCF[4] was introduced by IBM to provide this standardized interface. CCF provides a non-managed interface to systems such as CICS* (Customer Information Control System), IMS, and SAP, as shown in Figure 5. It provides a common client programming model for connectors but can only be used in a non-managed environment, and so does not allow the provider to interact with the server for services such as security and transactions. VisualAge* for Java (VAJ) provided support for CCF by providing a "command" bean that could be used with CCF. The concepts introduced by CCF later evolved into the J2EE Connector Architecture 1.0 specification.

J2EE Connector Architecture

The following section describes the evolution of the J2EE Connector Architecture, from its technology preview in WebSphere Application Server Version 4.0, to Version 1.0 which appeared in WebSphere Application Server Version 5.0, and Version 1.5, which will accompany WebSphere Application Server Version 6.0.

The J2EE Connector Architecture expanded upon the essential concepts embodied by JDBC and CCF and brought them together to provide support that would allow a client to access many types of EIS systems via resource adapters provided by vendors for their EIS systems. Some types of EIS resources are relational databases such as DB2, hierarchical databases such as IMS, procedural systems such as CICS, messaging systems such as WebSphere MQ Series*, and enterprise resources such as SAP resources.

Along with JDBC and CCF, a technology preview of J2EE Connector Architecture first appeared in WebSphere Application Server Version 4.0[5,6] (see Figure 6). The technology preview of J2EE Connector Architecture provided the following features:

- Managed access to EIS resources when used with the EJB container
- Connection pooling
- Limited connection cleanup support
- Mandatory connection sharing
- No connection association—that is, getting a connection and using and closing it should be contained within a transaction
- Limited security support—only component sign-on supported; no container managed sign-on
- Security permissions not processed—resource adapters manage all permissions
- Limited local transaction support—application responsible for transaction cleanup
- Limited XA[7] transaction support (XA recovery not supported)
- Packaging and deployment of stand-alone (not embedded) resource adapters

The production version of the J2EE Connector Architecture[8] (see Figure 7) was shipped in WebSphere

Figure 6 J2EE Connector Architecture (preview) in WebSphere Application Server Version 4.0

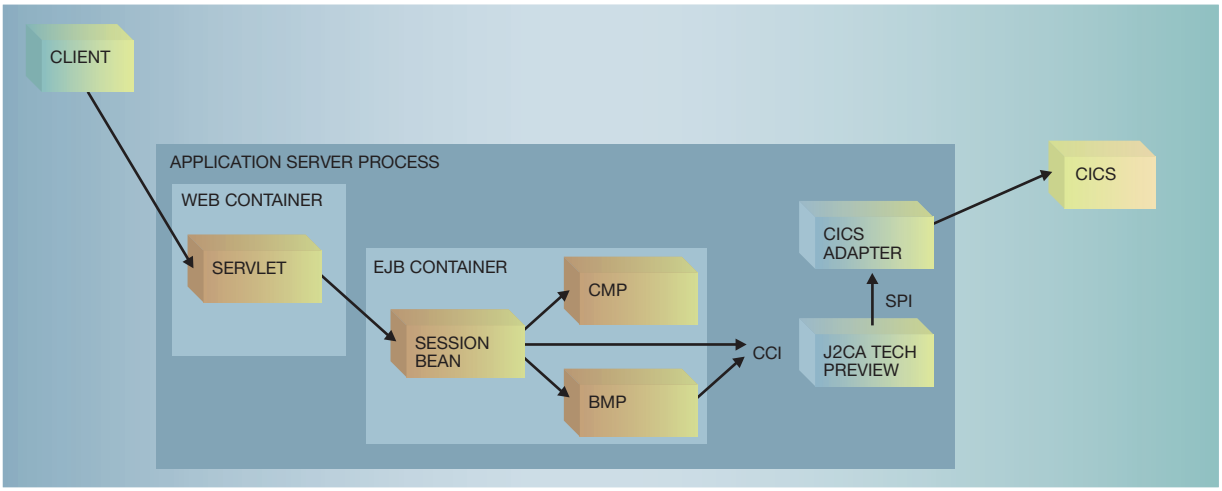
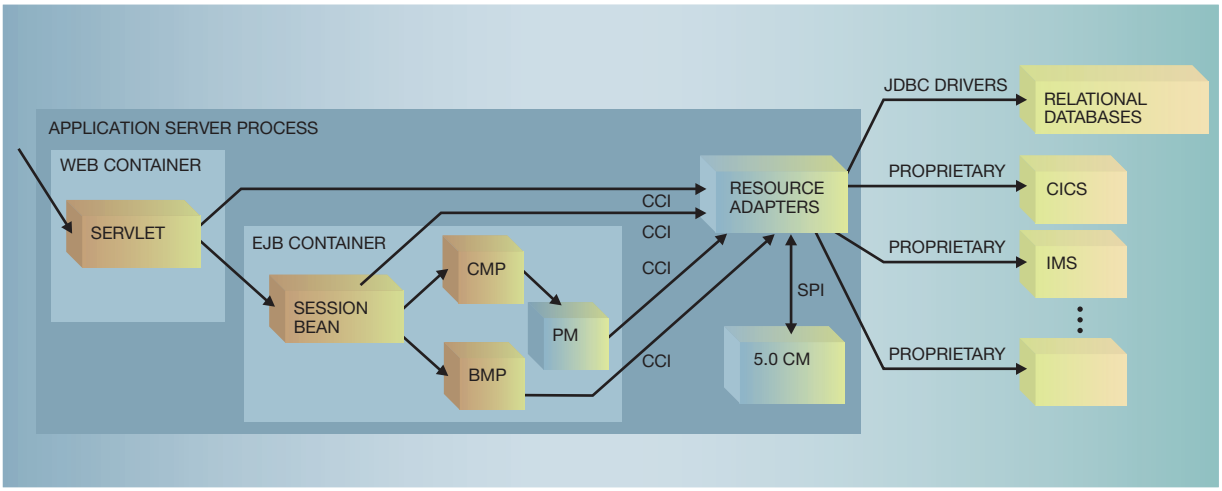


Figure 7 J2EE Connector Architecture (Version 1.0) in WebSphere Application Server Version 5.0

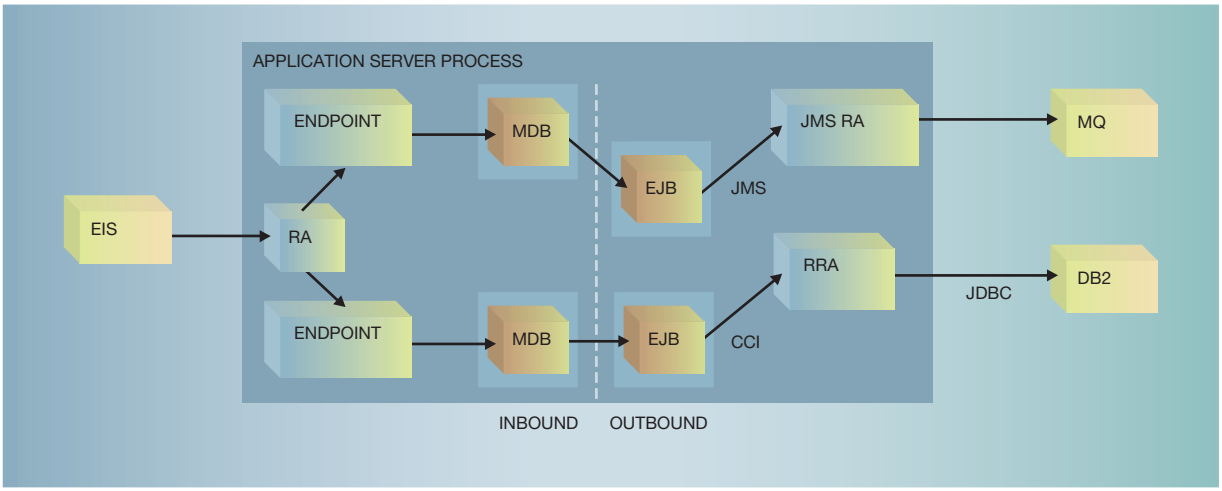


Application Server Version 5.0 and provided many functions that were missing in the technology preview, such as:

- Full local transaction and XA transaction support, including transaction cleanup
- Both shareable and unshareable connection support
- Connection association
- A common connection manager (In WebSphere Application Server Version 4.0, the JDBC CM and the J2EE Connector Architecture CM were separate.)
- Container-managed sign-on
- Security permissions processed
- Embedded resource adapters
- XA recovery

The J2EE Connector Architecture defines both a Common Client Interface (CCI) and a resource adapter/server set of contracts provided as service provider interfaces (SPIs). J2EE Connector Architec-

Figure 8 J2EE Connector Architecture (Version 1.5) in WebSphere Application Server Version 6.0



ture 1.0 allowed resource adapters to be designed and implemented for any resource. Two resource adapters were developed for use within WebSphere: the relational resource adapter (RRA) and the adapter for the Java Message Service (JMS),[9] which provides messaging support. IBM also developed resource adapters that are shipped separately from WebSphere, including adapters for CICS, IMS, Host On Demand (HOD), MySAP.com, J.D. Edwards, PeopleSoft Inc., and Oracle Corporation's Financial Services.

In addition to new resource adapters, a new connection manager was created to be used by all resource adapters. This CM has all of the benefits of the earlier CM, such as taking care of connection pooling for all adapters and providing additional connection management functionality as defined by the J2EE Connector Architecture. (The previous CM was retained in WebSphere Application Server Version 5.0 to support existing EJBs and servlets from prior releases.)

In J2EE Connector Architecture Version 1.0, the connection support was "outbound" only. This means that connections flowed from the application server to resources outside of the server. J2EE Connector Architecture Version 1.5[10] has added "inbound" support for both connections and transactions. Inbound support allows an EIS to interact with a resource adapter on a server, which in turn can pass on messages to "endpoints" that are installed on the server. Inbound transactions can be passed on to the Work Manager defined in the Version 1.5 architecture, which can then pass on the message to the endpoint. The Lifecycle Manager defined in Version 1.5 allows for the orderly deployment, startup, and shutdown of resource adapters on an application server. J2EE Connector Architecture Version 1.5, which is illustrated in Figure 8, provides a complete solution for many outbound and inbound connection scenarios.

Beyond the J2EE Connector Architecture

Despite the evolution of the J2EE Connector Architecture (and other related architectures), being able to configure a system with any communication protocol and any EIS resource still presents a challenge. Defining, setting up, and using a connection to a resource is only part of the story. Setting up and using a complete communication path from client application or EIS to the server machine and then to a resource external to the server process is a much larger issue. There are also challenges involved in handling the information that is passed along as part of the communication.

The J2EE Connector Architecture provides an abstraction for accessing legacy resources, but each resource adapter is tightly coupled with the protocol used to communicate with both the client and the server. As new, more efficient mechanisms are developed for performing network communications, we would like to be able to take advantage of these

mechanisms without modifying the adapter abstraction logic. The J2EE Connector Architecture allows third-party extensions at an abstraction level associated with accessing legacy resources. It would be useful to allow third-party extensions at abstractions both above and below this level.

What is needed is a way to plug in new protocols, new resources, and new extensions as they become available, without having to rework logic that is already proven and working. One should be able to do this seamlessly without stopping applications that are currently running. IBM is designing a new architecture, the *Channel Framework Architecture*, that will expand connection design and plugability capabilities while allowing the continued use of the J2EE Connector Architecture for connector design. This architecture will go beyond solving connector issues to connection handling and information handling in general.

Channel Framework Architecture

The Channel Framework Architecture (CFA) provides common networking services, protocols, and I/O operations for the WebSphere Application Server. The CFA extends the concept of a networking protocol stack to the WebSphere runtime. This architecture provides extended plugability along the entire chain of events involved in handling communication with the server and processing of the content of the communication at various steps in the server. This plugability is upwardly compatible with the J2EE Connector Architecture in that J2EE connectors could eventually be included as components in the CFA.

The CFA defines a set of interfaces that can be used to implement two main types of objects, *channels* and *channel chains*. Channels (see Figure 9) are used to transport data between the network and a WebSphere server component. Channels are an encapsulation of the logic for processing some part of a data stream or for interfacing with a component. The data stream may be part of an inbound request to an application server or an outbound request from an application server. Channel chains (see Figure 10) consist of a set of channels that are linked together and used to transport data from the network to a component or from a component to the network.

Currently, there are many implementations of network services in WebSphere. For example, each type of container implements a complete set of functions for reading and writing to the network. Without common networking services, new server components have to re-implement a complete set of network service functions. Not only is this inefficient, but this re-implementation also makes it difficult to add enhancements to networking services that are used by everyone. For example, Java Version 1.4 provides new nonblocking I/O network calls. Without common

Figure 9 Channel Framework Architecture

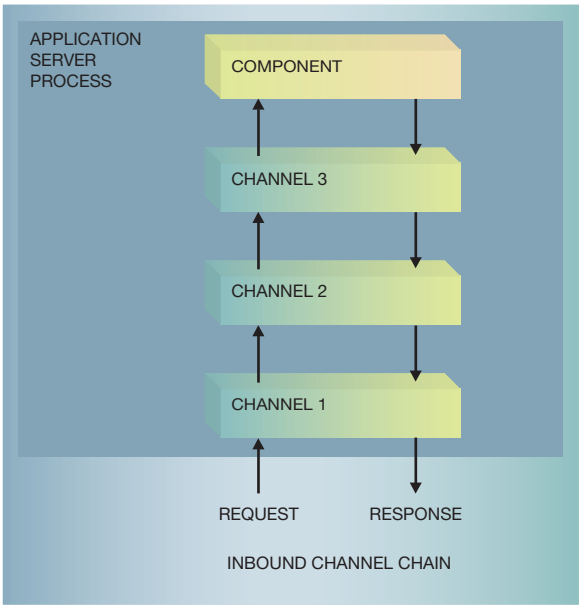


Figure 10 Channel Framework User with channel chains

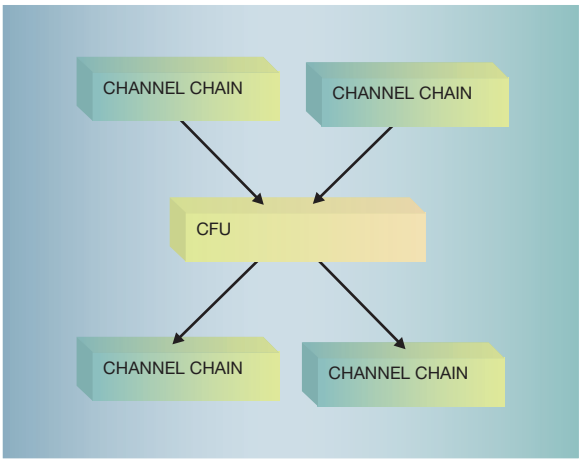
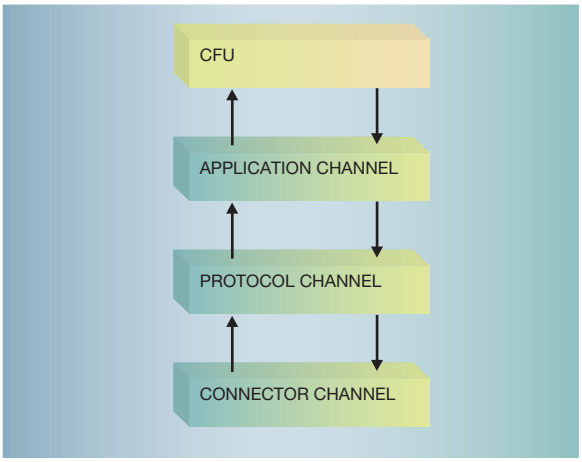


Figure 11 CCF channel types



network services, each existing and new implementation of network services has to add this function to its unique implementation.

In addition to providing common networking services, protocols, and I/O operations, the CFA will provide a standard architecture and infrastructure for encapsulating network logic into channels. By defining a standard mechanism for combining channels, it becomes possible to plug in custom channels that support requirements unique to a particular customer or environment. Additionally, the CFA will define a common point in the message flow through the channels where future enhancements for workload classification and thread dispatching can be implemented. The architecture will also support the ability to share network ports among multiple protocols. Finally, the CFA will provide functions for configuring, administering, and initializing channels. It will also be responsible for establishing the flow of network traffic through the correct channels.

In the CFA, a component that is at the beginning or the end of a channel chain is known as the *channel framework user (CFU)*. In an inbound channel chain, the request originates at the network and ends at the CFU. In an outbound channel chain, the request originates at the CFU and ends at the network. Examples of CFUs are the Web container and the EJB container. CFUs may have many inbound channel chains leading to them and many outbound channel chains leading from them.

Types of channel. Channels can be classified into three types, as shown in Figure 11:

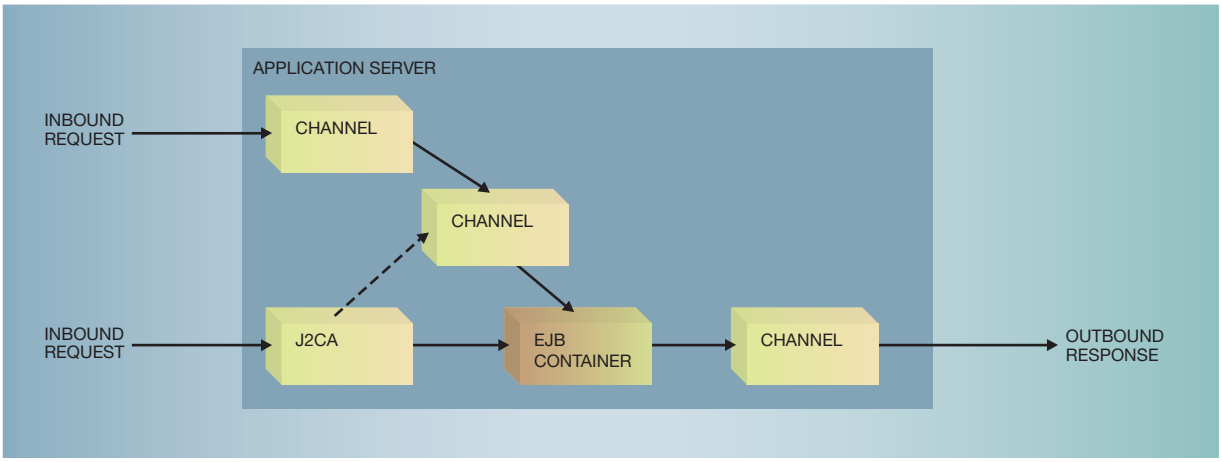
1. *Connector channels* are those channels closest to the network interface. A connector channel is the lowest channel in the channel protocol stack. An example of a connector channel is a TCP (Transmission Control Protocol) channel.
2. *Protocol channels* are responsible for abstracting information that is transferred through them. When reading information, a protocol channel will generally parse the information into high-level structures that map to constructs which are specific to the protocol. When writing information, a protocol channel will take information provided in protocol-specific structures and map it to the structures needed by the channel below it in the stack. There may be zero or more protocol channels in the link between the connector channel and the application channel. An example of a protocol channel would be an HTTP (HyperText Transfer Protocol) channel.
3. *Application channels* are the top-level channels. These channels are generally built on top of specific protocols in order to draw the correct data out of them. Examples of application channels include those for the EJB container and the HTTP Proxy.

Inbound channel chains and discrimination processing. At the time of program execution, the Channel Framework helps ensure that messages arriving at a connector channel follow the correct inbound channel chain to the appropriate CFU. The CFA accomplishes this by using a process called *discrimination*.

Each channel in an inbound chain contains a discrimination process, which is implemented and assigned to the channel by the framework. In the CFA, this object will hold the discriminators of each of the channels above it in the stack. When a new connection arrives at a connector channel, it calls the discrimination process. The CFA calls the discriminator of each channel above it in the stack. The discriminator uses data provided to determine whether the channel should accept the connection. When a channel indicates that it will accept the connection, the CFA makes the connection for that channel.

After all the channels in an inbound channel chain have been linked, the CFA no longer participates in the processing of the message as it flows through the

Figure 12 Use of protocol channels to transform requests and responses



channels. The CFA does not impose any execution overhead beyond connection set-up.

Outbound channel chains. At runtime, the CFA is responsible for validating, creating, initializing, modifying, and removing any outbound channel chains used by components in the application server. Outbound channel chains have a preset configuration of channels. There is no discrimination process associated with outbound channel chains.

Outbound channel chains can be stored as named channel chains in the configuration. Components can access the preconfigured outbound channel chains by name. Components can also create named outbound channel chains dynamically, by specifying the named channels to use. Named channels can be created dynamically by specifying the Channel Factory to use and the configuration parameters of the channel.

Channel chain topologies. There is a difference in the possible topologies for inbound channel chains and outbound channel chains. A channel can appear in more than one inbound channel chain. The topology of an inbound channel chain is represented by discriminators. The discriminator of a channel defines the possible channels that can appear above it in the stack. The discriminators are used at runtime by a discrimination process in the CFA to select the specific channels that are to handle a particular connection.

Whereas a channel can appear in more than one inbound chain, a channel in an outbound channel chain can have at most one channel above or below it in the stack. The topology of an outbound channel chain is thus described by a linear list of channels. Because the list is linear, there is no need for a discrimination process to determine the channels that are to handle a particular outbound connection.

Benefits of the CFA. The CFA eliminates duplicate code for implementing network communication functions. Each developer requiring network services no longer has to implement a full network stack, making network services easier to maintain and enhance. All users of the CFA benefit from its enhancements without modifying code. For example, if the nonblocking I/O functions are eventually implemented that use higher-performance asynchronous I/O functions, all users automatically receive this performance benefit.

As new protocols are required for handling requests for WebSphere Application Server services, new protocol channels can be written. For example, a protocol channel that understands the format of some legacy application's requests could be written to transform the request into a format understood by the EJB container's application channel. When the response is delivered from the EJB container's application channel, the new protocol channel can transform the response into the format understood by the legacy application. In this way, new protocols can be accepted by the EJB container without changing any of the container code itself. In the future, the J2CA itself could become a channel, as depicted in Figure 12.

Conclusion

Access by applications to new resources outside of the application server will continue to be a critical requirement, and new network protocols that provide additional functionality and performance improvements will continue to be developed. To date, adding new connection protocols has been an exercise in extending the server runtime in various ways to add new protocols. As more protocols are added, it becomes more and more problematic to make the new protocols quickly available.

The J2EE Connector Architecture has evolved over time to give vendors the ability to design and deploy resource adapters that can provide access for application components to resources and access for resources to application components. The CFA will give vendors the ability to design and deploy new protocols that can be plugged into the application server with the same ease.

*Trademark or registered trademark of International Business Machines, Inc.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. R. Monson-Haefel, *Read All About EJB 2.0,* IBM DeveloperWorks (2000), http://www-106.ibm.com/developerworks/java/library/j-jw-ejb20/.
2. B. Shannon, *Java 2 Platform, Enterprise Edition (J2EE) Specification Version 1.4*, Sun Microsystems, Inc., Santa Clara, CA (2002), http://java.sun.com/j2ee/1.4/download.html#platformspec.
3. M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, Third edition, Addison-Wesley Publishing Co., Reading, MA (2003).
4. T. Oya, B. Brown, M. Smithson, and T. Taguchi, *CCF Connectors and Database Connections Using WebSphere Advanced Edition: Connecting Enterprise Information Systems to the Web*, IBM, Austin, TX (2000).
5. S. Minocha and P. McMillan, "Migrating Enterprise Access Builder Components from VisualAge for Java to WebSphere Studio Application Developer," *IBM WebSphere Developer Technical Journal* (2002), http://www7b.boulder.ibm.com/wsdd/techjournal/0201_minocha/minocha.html.
6. S. Minocha and P. McMillan, "Migrating Enterprise Access Builder Components from VisualAge for Java to WebSphere Studio Application Developer Integration Edition 4.1," *IBM WebSphere Developer Technical Journal* (2002), http://yangcontent.svl.ibm.com/wsdd/techjournal/0207_minocha/minocha.html.
7. *Distributed TP: The XA Specification,* The Open Group (February 1992), http://www.opengroup.org/publications/catalog/c193.htm.
8. K. Kelle, P. Schommer, and K. Sutter, "J2EE Connector Architecture Extensions in WebSphere Application Server V5," *IBM WebSphere Developer Technical Journal* (2003), http://www7b.software.ibm.com/wsdd/techjournal/0302_kelle/kelle.html.
9. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, *Java Message Service*, Sun Microsystems, Inc., Santa Clara, CA (2002), http://java.sun.com/products/jms/docs.html.
10. R. Jeyaraman, *J2EE Connector Architecture Specification*, Sun Microsystems, Inc., Santa Clara, CA (2002), http://java.sun.com/j2ee/connector/download.html.

Accepted for publication December 8, 2003.

Stephen M. Fontes *IBM Software Group, 3039 Cornwallis Road, Research Triangle Park, NC 27709 (fontes@us.ibm.com).* Mr. Fontes is a senior software engineer in the WebSphere Application Server development group. He obtained a B.S. degree in management and an M.S. degree in computer science from Worcester Polytechnic Institute in Worcester, Massachusetts. After graduating in 1984, he joined IBM Endicott, working in computer-aided engineering. While in Endicott, he obtained an M.S. degree in mechanical engineering from Cornell University in Ithaca, New York. In 1996, he relocated to Research Triangle Park (RTP) to be the lead programmer for IBM's edge load-balancing solution. While in RTP, he has worked in a variety of roles, including security architect for IBM's Tivoli division. He is currently the lead architect for WebSphere's edge components and Channel Framework Architecture.

Cathlene J. Nordstrom *IBM Software Group, 3605 Highway 52 North, Rochester, MN 55901-7829 (cjn@us.ibm.com).* Ms. Nordstrom is an advisory software engineer in the WebSphere Application Server development group. She earned a B.S. degree in computer science in 1981 at North Dakota State University. She joined IBM in 1984 after working as a computer analyst in the manufacturing industry. Her work in architecture and development has spanned a broad range of products including PC Support, OfficeVision/400™, AS/400® Integrated File System (IFS) and currently connection management for the WebSphere Application Server. She has presented various topics at COMMON (an international professional association for ongoing education and discussion of IBM's mid-range systems).

Kevin W. Sutter *IBM Software Group, 3605 Highway 52 North, Rochester, MN 55901-7829 (sutter@us.ibm.com).* Mr. Sutter is a senior software engineer in the WebSphere Application Server development group. He started his computer career with Unisys in 1983 after graduating from University of Wisconsin - La Crosse with a B.S. degree in computer science. In 1988, he joined IBM Rochester in the PC Support area (client/server product support for the AS/400®). Since that time, he has worked on several projects, including Host Print Transform for the AS/400, SOM/DSOM (System Object Model/Distributed System Object Model) Persistence and LifeCycle, Component Broker, and now WebSphere. Mr. Sutter is the lead architect for the J2EE Connector Architecture implementation within the WebSphere runtime. He has authored several papers and presentations pertaining to connection management.