# WebSphere Programming Model and Architecture

The WebSphere Application Server is the leading J2EE application server product in the marketplace today [Snyder 2003]. WebSphere offers a scalable, secure, highly available, transactional hosting environment for shared, re-usable mission critical business services, components and web applications across a broad variety of server platforms. This paper surveys the programming model and architecture of the WebSphere Application Server. We will explore how the programming model enables a services-oriented architecture and relate the value that can offer to businesses. It will detail some of the fundamental design principles that have guided the structure of the application server, and will describe how some of those principles are manifest in key aspects of the application server runtime [IBM WebSphere].

#### Principles of Middleware Design

The WebSphere Application Server (WAS) is designed with three major engineering principles in mind: *strength*, *precision*, and *tolerance*.

The application server is a tool; a machine. Its utility comes from being strong enough to run the most demanding workloads. A modern enterprise computing environment may be subject to hundreds or even thousands of transactions a second, from thousands of concurrent clients, over millions of accounts and terabytes of data. The application server must be able to handle these loads without faltering.

The application server must also be precise; delivering provably correct results for the applications that are hosted on it. It is the responsibility of the middleware to ensure the integrity of the applications it hosts – coordinating the transactional commitment of data, protecting access to resources, identifying potential problems and bringing them to the attention of administrators or even automating the correction of those problems, managing the distribution of workloads with affinity to in-flight state, maintaining high availability even in the presence of failures, and ensuring efficient utilization of underlying computing resources. In many ways, this defines the application server as a *transaction monitor*.

On the other hand, customers will exploit the application server in ways that are limited only by the imagination and needs of those customers. The application server must be tolerant of the different circumstances in which it may be used. It has to be flexible, and adaptable – both in terms of enabling customers to customize or extend the application server for their particular needs, and in terms of being tunable for optimal performance in those different environments.

And WAS is designed with another major principle in mind: as middleware it is responsible for maintaining a clean separation of concerns between the business application and the underlying information technology on which it is hosted. This principle is motivated by the realization that most application developers are commissioned first and foremost to create value for their businesses. By contrast, modern information systems technologies are growing rapidly more and more complex [Schneberger 1995].

Many customers have had to install and deploy a large variety of different vendor systems - from Windows to Unix to mainframe based systems for one reason or another. The economics of computing have driven higher degrees of distribution. Acquired applications come with their individual and unique assumptions about the systems they depend on. Programming languages have evolved, and have introduced more sophisticated and complex technologies for gaining advantage - from procedural to scripted to message-oriented to object-oriented to aspect-oriented to service-oriented; from shared-logic to networked to distributed to internet-enabled systems. While each of these technology advances have their own advantages, it has also put more stress on computer professionals to maintain their skills, and has continued to force application developers to spend far more of their time resolving deep information technology issues, leaving less time to add value to their business.

The middleware has the responsibility of shielding the application developer from the complexity of the underlying information system. This is accomplished through a combination of component model design and a container environment for hosting those components. We'll return to the topic of component models later in this paper. For now, understand that the contractual boundary between the component model and its container forms the boundary that separates the application logic from the underlying information technology system. Fundamental to this is a deployment model that deliberately partitions the lifecycle of creating and maintaining components, assembling them into useful applications for a given business situation, deploying that against a particular instance of information technology, and then administering their execution so that different specialists can contribute their unique knowledge in a collaborative fashion to achieve an optimal balance of skills and overall systems objectives. It is within this framework that computer professionals can build synergy and value to the business. It lays the foundation for allowing the information system to be treated as a business asset rather than simply the cost of doing business.

Collectively, this attention to separating concerns, enabling collaboration between business and computer professionals, encapsulating the complexity of the information technology system, and ensuring a high degree of strength, precision and tolerance defines WebSphere as *middleware* – in essence a distributed systems platform for portable application designs over heterogeneous computing systems.

#### Structure of e-Business Computing

To achieve maximum utility, WebSphere makes basic assumptions about how applications are structured and deployed in the information system [IBM Patterns]. In this paper, we classify application deployment structures in to four basic categories:

- · Web Computing
- Multi-tier Distributed Computing
- Enterprise Application Integration
- Services Oriented Architecture

*Web computing* focuses on the use of Java Server Pages (JSPs) and Servlets for capturing presentation logic that drives an interaction with end-users through web browsers across an internet or intranet. JSPs and Servlets form two parts of the classic three part Model-View-Controller (MVC) application design pattern – JSPs render views to the end-user, and Servlets take input from the end-user to control the interaction with the application and to drive additional views [Röwekamp & Roßbach 2000].

Some of the earliest and simplest exploitations of the JSP and Servlet capabilities of the application server tended to involve placing the business logic right in the Servlet control functions. While this works fine for simple web applications it tends to mix concerns, and that reduces the re-use and maintainability of the logic.

WebSphere also supports Enterprise Java Beans (EJBs) for capturing the business logic of the application. EJBs come in two different types: Session Beans and Entity Beans. Session Beans are intended to capture procedural logic that performs specific business functions in the application – the *verbs* or activities of the business design. Entity Beans are intended to capture the objects of the business; things that the business works with – the *nouns* or entities of the business design.

All of the J2EE component models, but especially EJBs, are designed with the idea that they contain shared, re-usable componentized function [J2EE]. The idea is that they can be created within an application design in a way that allows them to be re-used in other applications, or at least shared amongst many different clients and end-users. A common variation on this theme occurs when the same underlving EJBs are shared amongst different client applications. For example, a set of business operations that you can perform on a banking account, such as deposit, withdrawal, transfer, etc., may be implemented on a composition of a Session Bean (for the banking operations) and an Entity Bean (representing the account). These same components can be shared between Teller, Call Center, Home Banking, and ATM clients. This is a good way of ensuring that all of the business channels of the enterprise are exposed to a consistent set of business policies and data of record.

WebSphere supports both bean-managed as well as container-managed forms of Entity Beans. Each has their trade-offs bean-managed Entities give the application more control over how the bean's state is handled with the data system. However, with the introduction of the abstract bean class, the EJB-Query Language for portable finders, and container-managed relationships introduced by the EJB 2.1 specification, and support for access intent policies in WebSphere R5.0, we most often recommend the use of container-managed persistence. There are several reasons for this, but the most notable is that it fulfills the principle of pushing down the concerns of the information systems technology to the underlying middleware.

More specifically, the bean container can combine its knowledge of the other things that are going on in the computing system - including, what other requests are being performed on other beans and applications, the priority of workload, the likelihood of traversing relationships or iterating over a set of similar records, etc. - to optimize the total throughput of the system. The container has built-in caches and recognizes common patterns of interaction to make decisions about how to combine I/O to the data system, retrieve data directly from in-memory caches, or how to prioritize certain work over other work. All of these advantages are often impossible to perform in the application without knowing a great deal about the other applications that may be installed on the same system, or at least it would take a great deal of very specific systems programming that has essentially nothing to do with the business logic of the application.

The characteristic of re-use and sharing that is intrinsic to the EJB component design can be leveraged in the application to form a natural distribution boundary allowing the EJB components to be separated from the presentation logic in another application server. And to the extent that business logic may be composed of other business logic, the principle of componentization, re-use, sharing and distribution can be recursively applied throughout the application design. When taken to its extreme, applications formed from an aggregation of components can be deployed across multiple tiers in the information system. We refer to this a multi-tiered distributed computing.

However, these partitions can also be treated as logical tiers - without separating them on different application servers. The trade-offs between localized vs. distributed topologies is a topic in itself, but one that should be considered carefully when setting up a distributed systems infrastructure. The latency and higher potential for failure when components are separated across a distributed system (including on different application server instances in the same computer) does have an impact on the application design and implementation. The EJB specification does allow the exploitation of distributed components to be constrained through the use of Local Interfaces. But, in the end, the choice of whether to distribute the components of an applications are a choice that can be made, first by the developer by their use of different component types and then, if not constrained, by the application assembler and/or deployer.

Few applications today can be deployed as an island in the business system. Most enterprises today already have a large variety of applications already installed and in use within their business. These applications implement key function that, like the very J2EE principles we've already described, should be shared and re-used even with new J2EE applications. WebSphere offers a number of techniques for *enterprise application integration*.

The Java 2 Connector Architecture (J2CA) allows you to adapt procedural business function that is implemented on other legacy systems such as CICS, IMS, SAP, PeopleSoft, Siebel, etc. for use within a J2EE application. The J2CA specification defines the Common Client Interface (CCI) - a standard interface for connectors that can be plugged into the framework. Each connector can be designed to the specific format and protocol mechanisms supported by the legacy system. However, the CCI is a very low-level interface. WebSphere Studio tooling has support for creating adapters that are specific to the application. An adapter is a Stateless Session Bean that implements the interface of the legacy business function being adapted, and encapsulates calls to the CCI to invoke and collect the results of the legacy business function. For the WebSphere application it appears that the business function is implemented

within the Session Bean – like any other business function the application might use.

Another approach to integrating legacy systems is through asynchronous messaging. WebSphere supports the Java Messaging Service (JMS) specification, and includes an integral provider that is based on the MQ-Series queue manager and event broker. In addition, WAS supports the full MQ-Series as a plugable provider, as well as other 3rd-party providers such as SonicMQ and Tibco - any messaging provider that supports the JMS specification. Asynchronous messaging provides an ideal way of integrating legacy applications that are already designed and implemented to support asynchronous messaging.

However, asynchronous messaging also offers an opportunity to design applications that are loosely coupled – where the parts are connected only by the messages that flow between them. EJBs can send messages using the JMS client interfaces, and can be invoked to receive a messages by registering an Message Driven Bean (MDB) with a message listener. Messages can be exchanged between application parts on a point-to-point queue basis or on a topic basis – the latter being the foundation for subscribing to notifications.

Loose-coupling between components is a key element in successful distributed computing design. The looser you couple the parts of your applications, the more tolerant your application will be to being distributed. We can measure the degree of coupling in three ways:

- · Temporal and granularity constraints
- Technology constraints
- · Organizational constraints

If the interaction between two components is time-sensitive - for example, the calling component depends on the target component to return a value within microseconds (or even a few milliseconds), or if there will be locks held on the data system while multiple interactions between the components take place - then these components probably need to be co-located. Distributed components should not have temporal constraints. One thing in particular that will tend to introduce temporal constraints is the granularity of the component and the interactions with it. If a component is too finegrained and results in either being created and destroyed frequently or requires lots of fine-grained interactions with it, then it will likely impose temporal constraints and therefore will not be a good candidate for distribution. Well behaved distributed components tend to be coarse-grained, and introduce coarse-grained interactions – introducing larger data structures in the request arguments to keep the number of interactions down.

The degree to which both the calling component and the target component must be implemented in the same underlying middleware technology or language becomes a constraint to distributed computing. Said differently, to be successful, distributed computing infrastructures must be able to support interworking between components that are implemented in different technologies and languages. The extent to which *technology constraints* are made transparent to the application will enable more opportunities for distributed interconnections between application components.

Often owners of different components in the distributed system will operate to different development and maintenance lifecycles. One organization may be forced to upgrade their server components more frequently than the client organizations that make use of those components. Whenever a target component changes it has the potential for breaking backwards compatibility with any client applications that use them. When compatibility breaks both organizations are forced to synchronize their updates. This forces collaboration and depending on the number of parties involved, and the flexibility each has to adjust to the other's needs, can affect the success of maintaining a relationship between the interworking applications. This is just one form of organizational constraint – a dependency that is affected by the interrelationships between organizations involved in the end-to-end distributed system. Distributed systems rely on there being few, if any, organizational constraints.

Services oriented architecture (SOA) is an application design principle that attempts to maximize the potential for distributed systems by deliberately and explicitly attempting to minimize temporal, technological, and organizational constraints [SOA]. SOA focuses on *business services* – that is, those services that a business might offer to its customers or business partners, or those services that one business unit might offer to another business unit - as the primary element of distribution [Burbeck 2000]. While this may be a bit extreme for some applications, it is a good starting point to ensure a minimum of temporal, technology and organizational constraints. At a gross level, businesses offer services to their customers through relatively coarse-grained channels such as through store clerks, agents, call-center operators, through web-pages on the internet, over dial-up or wide-area network protocols, or by mail. These channels tend to assume interactions that are measured in seconds, minutes, or even days - certainly not in microseconds.

They also tend to create very few assumptions about technology, and have a high toleration for changes at either end of the interaction – or, at least, changes can be accommodated within some degrees of a basic understanding of the semantics of the transaction, and with some degree of intermediation by humans, governing and standards bodies, or intelligent agents.

WebSphere provides integral support for services-oriented architecture with middleware facilities that conform to the Web Services standards - a set of specifications that have been written by IBM and Microsoft in collaboration with other vendors, and have been submitted for standardization by OASIS and W3C. The web services standards cover many important aspects of distributed coupling, including interoperable encoding formats, security, data integrity, workflow management, notifications, context, policy, and metadata exchange. Perhaps the most significant of these is the Web Services Definition Language (WSDL). WSDL provides a technology- and language-neutral abstract representation of a distributed service. WSDL is designed to enable a high degree of technology and organizational independence - allowing the service provider to express the variety of protocols and encodings supported by the service, and to express the message elements that are supported by the service, but without preventing the service provider from evolving or expanding the messages supported by the service in the future.

We will explore WebSphere's support for web services in more detail later in this paper.

For now, note that each of the four categories of application design supported by WebSphere - web computing, multitiered distributed computing, enterprise application integration, and services oriented architecture - are not fully distinct application design categories, but rather are touch-points on a continuous progression of application design patterns that range from simple through to sophisticated. WebSphere supports this entire range of application design patterns specifically because one size does not fit all - different business and deployment scenarios will require different design approaches to solve their specific needs. On the other hand, there is often tremendous overlap between those needs and it is important that application developers should not have to learn entirely different approaches to building application components when moving between points on the continuum of scenarios.

## Heterogeneous Platform Portability

An essential part of the WebSphere middleware proposition stems from maintaining a strong separation of concerns between the application logic and the underlying information system. To realize this goal, the application server must encapsulate the information system and operating system platform within its programming model. Moreso, the programming model must be completely consistent on all platforms. It must support the entire breadth of the programming model - including all standard and proprietary elements of the programming model - on all platforms. The customer should never be faced with the situation that having built an application they can not expect that application to perform the function is was designed for on different computing platforms. The functional behavior should be exactly the same; same services, same bugs, same work arounds, same fixes.

That's not to say that different operating platforms are completely equal. In fact, different operating platforms do have a number of non-functional differences. We refer to these as *quality-ofservice* (QoS) differences. Some operating systems are inherently more stable than others; some have a lighter footprint, some are more scalable, some are simpler, some are more secure, some are less expensive to operate, etc. And, outside of the Java, J2EE and WebSphere execution space, different platforms offer differing support for certain programming languages and technologies. Customers will leverage these differences when deciding which platform to install in the operation centers. And they want to leverage and benefit from these differences with their WebSphere applications.

WebSphere is architected to blend a high degree of functional consistency with platform-specific customizations that are designed to leverage the unique differences of the platform. More specifically, the vast majority of the application server implementation is exactly the same on all of the platforms that Web-Sphere supports. However, it is designed with specific plug-points to exploit the unique QoS advantages of each platform. For example, the security user registry and authentication services are plugable to use the native operating system registry and authentication services on each platform. This enables integration with the security subsystem for other applications on the same computer. The built-in transaction manager is replaced with the Resource Recovery Service (RRS) on z/OS to enable integration with other middleware systems such as IMS, MQ-Series and DB2 on z/OS [WASZOS].

The WebSphere application server provides this same blend of consistency and unique advantage on data center, departmental and work group platforms such as z/ OS, Linux/390, iSeries, AIX, Solaris, HP/UX, Linux and Windows/ Server, and on desktop systems such as Linux and Windows.

A subset of the WAS programming model is also being made available for pervasive devices such as PDAs and other hand-held and embedded devices. The underlying implementation of this support is different than the standard WAS product, although some components share the same common code base. Nonetheless, this version of WebSphere enables programmers to use many of the same programming techniques for creating presentation and business logic for execution on these devices - presentation logic can be created using JSPs and Servlets and business logic can be created using EJBs. This allows for a high degree of consistency in the skills, techniques and tools used to create applications, and even allows that some application components can be common to both environments (although the environments themselves tend to drive slightly different business application requirements – pervasive device applications tend to be much centered around the end-user to which the device belongs, whereas server-based applications tend to be more centered around how the enterprise wants to conduct its business consistently across all of its end-users and business channels) [Everyplace].

#### **Topology Considerations**

Like the range of application design requirements driven by different business and deployment scenarios, customers also have a wide variety of topology requirements. Some small businesses may have all of their end-users and business operations co-located in one office, and may be able to serve their entire business applications and data systems off of one or a pair of computers. Other larger businesses may have multiple regional data centers, several call centers, thousands of stores, branches or agent offices, and need the capacity of many hundreds or thousands of computers. Likewise, the range of topology structures is a continuum of varying degrees of simplicity or sophistication.

WebSphere is designed and packaged to address this range in an incremental and progressive fashion.

The simplest scenario is that of a single application server instance running on a single computer. This might be employed in a small business that does not need any more capacity than a single computer, or it might be a sandbox on a desktop computer for a developer to create and test their application. Multiple instances of the application server may be created on the same computer, or even on several computers. This might be useful if there are many developers sharing the same computer and each needs their own sandbox to work in. In either of these cases, each application server instance can be treated independently, with its own administrative domain.

The admin services for each application server instance can be operated independently – a distinct administrator can be assigned to each application server instance and operate in complete isolation of any other application server administrator. In the sandbox case, each developer can be their own administrator – starting and stopping their own sandbox instance, and installing and configuring their own applications and resource dependencies.

Larger environments may need the power of multiple application server instances, or to cluster multiple instances into a single logical server to enable a high- and continuously-available production environment for their applications. Multiple instances of the application server can be configured to run on one or more computers, and aggregated under a single administrative domain (an administrative cell in WebSphere terminology). One or more administrators can be presented a centralized view of the cell, and can be granted different authorities to monitor, operate or configure applications or resources in the cell.

These environments may be composed of the same operating system platforms, or a heterogeneous mix of operating system platforms. The WebSphere management system presents a uniform and consistent view of administration in either of these situations. The admin console is a browser-based web application that runs on the WebSphere application server like any other web application, and presents the same administration model in either a single-server standalone environment, or in a multi-server or clustered environment. This enables administrators to easily transition between either environment extremes, and regardless of which operating platform is being used in the environment.

The same application server is installed in each of these scenarios. Installing the Network Deployment Manager on a computer in the cell then forms the basis of the cell. Individual nodes within then cell can then be federated into the cell, and any application servers that may have already been installed can be included in the cell and administered centrally. The deployment manager serves as a hub for the cell, providing a centralized view of all of the applications and application servers deployed in the cell. New application sever instances can be created from the deployment manager, and aggregated into clusters. Applications can be installed centrally at the deployment manager, and then configured and distributed to the application server instances where the will run.

However, every application server instance operates from its own local copy of the cell configuration. The deployment manager will synchronize the portion of the cell configuration that each application needs to operate successfully between its central location and the node through a node agent that is created automatically on each node in the cell. In this way, the deployment manager never represents a single point of failure in the system – each application server operates independent of the deployment manager.

# WebSphere Standard and Extended Components

WebSphere supports a variety of standard and proprietary application component types for different application design and integration situations. We will survey those component types in this section of the paper. However, before we do we should explore what a *component* is.

A component is a basic element of the business application. It exists within the application design to perform a specific role or function. It creates a framework for encapsulating the business design. An application may be composed of many components. Ideally, a component represents a unit of shareability or re-use within the application design. A given component may be used within the application for which it is designed, or may be re-used with many other applications. If the component is object-oriented, then many instances of the same component type may be created when using the application - each instance of the component will have a distinct identity that can referenced in the application. Most often, the instance identity will have a relationship to a relevant key value in the business application - such as an Account number or Purchase Order number. However, not all components need be object-oriented some may be procedural in nature and instantiable only in the sense of enabling reentrancy and survive only for the duration of a given session (a bracketed interaction) with the user or client application.

Modern component designs have a component model that maintains a strong separation of concerns. They encapsulate the logic of the application, and form a contract with the underlying infrastructure (usually represented through a container) that then is responsible for *managing*  that component - that is, handling the component's execution lifecycle, giving it identity and mapping that identity to the underlying resources that may be allocated to that component, managing the component's state. Depending on the quality of service assurances of the component model, the container is responsible for ensuring the integrity of the component instance - both transactional data integrity and correctness, as well as security integrity - and any other service processing that is assured as part of the containers quality of service contract. If the component model supports distributed components, then the component container is responsible for enabling remote-local transparency.<sup>1</sup>

Component models are designed to allow different container implementations provide different qualities-of-service. The value of a container is in its ability to balance the QoS it offers against a set of costs - including performance, throughput, resource efficiency, and administrative and maintenance impact. The component model and contractual relationship to the underlying container virtualizes the computing infrastructure for the application. The value of a given component model can be measured by the degree to which it maintains this separation of concerns; enabling the application developer to focus on their domain requirements; and enabling a higher degree of portability of that domain function to different computing environments.

Component models have two programming models – the programming model used by the component implementation, and the programming model used by clients of the component. The component implementation programming model defines what lifecycle operations that must or can be implemented by the component to assist the collaboration between the component and its container. It also identifies what the application logic within the component implementation is allowed to do and, just as importantly, what it can't do. For example, J2EE com-

The term »remote-local transparency« (as contrasted to »local-remote transparency«) implies that for transparency of local to be successful, the application must be designed to expect a component will be remote, and then may benefit from the additional optimization and efficiency that may come from the component being deployed locally.

ponent implementations should not use the Java Threads library to spawn its own threads. The client programming model describes how the client can gain access to the component, such as using JNDI java:comp references to find a component factory, or how to create new instances of the component. It also describes any assumptions or constraints the client can make about the statefulness of the component, whether it is re-entrant, can be shared concurrently amongst multiple clients, and, amongst other things, how to form an operational work context (session) for sustaining an interaction with the component.

WebSphere supports all of the J2EE standard component models, including:

- Web Components Servlets Portlets JSPs
- Business Components Stateful and Stateless Session Beans Entity Beans EAI Adapters
- SOA Components Web Services (pseudo component)
- Infrastructure Components Message Driven Beans J2CA EIS Connectors JMX MBeans

WAS, in combination with the WebSphere Business Integration product, also supports a number of proprietary component models:

- Choreography Business Processes (Workflows) Microflows
- Programming Assists Async Beans
- Startup Beans
- Business Policy Business Rule Beans

Other component models are emerging and will be supported in future releases of WebSphere, including SOA Resources and Service Data Objects.

We've already mentioned Servlets and JSPs. These are defined by the J2EE specification to support the MVC presentation design paradigm. More and more often these components are being combined with other frameworks such as the Apache Struts and Tiles. Struts is designed to automate the interaction between Servlets, JSPs and EJBs with the use of an Action Bean – a component that encapsulates the invocation of the EJB, and then establishes what to do next based on state transitions within an action table definition.

JSR 168 introduces the Portlet component model [JSR168]. In many ways, Portlets extend the Servlet model to include support for modes and window states. Portlets differ from Servlets primarily in that they represent individual frames (or windows) in the overall browser screen where each frame represents a distinct application. The idea being that a browser screen can concurrently aggregate multiple windows into different applications. The end-user can see all of their different applications at the same time, and interact within them independently, or even create visual interactions between them on the same screen.

Within the Portlet model, modes basically define the basic interaction states the end-user may have put the window in. Standard mode states include View, Edit, and Help. When a Portlet is in View mode, it is in a typical operational mode presenting the application content to the end-user following its basic application design. A Portlet that has been put in Edit mode is basically in a mode that allows the end-user to customize the view they want from the application. This might include defining what content they want the application to present, or changing the color, font or layout of the information they want the application to present. A Portlet in Help mode should be presenting help information about the application or how it can be customized.

Portlet windows can be sized – *nor-mal* (as determined by the layout of the overall screen), *minimized* (hidden from the main content of the screen – usually reduced down to some icon that represents its presence without exposing its full content), or *maximized* (consuming the entire screen).

The Portlet component model is not currently included in the J2EE platform specification, but is expected to become a standard part of J2EE application servers in the future. The WebSphere Portal Server already supports the Portlet component model.

The EJB specification defines Stateless and Stateful Session Beans. Session Beans, in general, should be used to capture the verbs of the business application the activities that the application *does*.
Entity Beans should be used to capture the nouns of the business application – the things the application *does it on*.

WebSphere supports all of these components. Nonetheless, the most scalable application designs are ones that remain stateless within the execution model that is, they gain access to the things they're going to operate on (effectively, retrieving their entities from persistent storage), perform their function, complete and clean up the resources they used during the function (storing their entity state back to persistent storage). Applications that follow this basic design premise are generally able to handle far more concurrent transactions from different endusers than those that attempt to cache lots of state in their application between transactions (caching may improve the performance of an individual application, but will not generally improve total systems throughput or scale).

Stateless Session Beans help fulfill this design preference because semantically they are state-less - they emphatically presume that each interaction with the Session Bean is independent. Any state they might depend on is either received from the client as part of the request, or is got from the underlying persistent entity model. Applications that are stateless can be cleaned up faster and they can be recovered much more efficiently. They are tidier; they don't accumulate a lot of resource that then requires a lot of complex management, and if they fail they can be restarted on another server without losing state (they don't have any) or without having to recreate a lot of cached state.

Stateful Session Beans, on the other hand, are inherently flawed and an unfortunate part of the J2EE specification. First, they violate the principle of stateless application design - they specifically presume to retain state between operations. Secondly, and perhaps more significantly, they're not recoverable - the container can not assure the integrity of state retained by the Stateful Session Bean. The EJB standard specifically prevents the container from passivating (writing out to persistent store) Stateful Session Beans within the transaction context of the business application. Thus, the state of the Session Bean can be lost between the time the application has committed its entity state, and before the Stateful Session Bean has persisted its state. The client application must be written to presume that at any given point the Stateful Session Bean may be invalidated and if it is the client must be prepared to recreate the operational state of the Session Bean – in other words, the client must retain a parallel copy of the Session Beans state, and reload that in the Session Bean if it should ever receive a NoSuchObjectException.

We recommend that if you must use Stateful Session Beans in your application design, you do so by implementing it to retain its state in an underlying Entity Bean, and arrange for the client application to initialize the Stateful Session Bean with the primary key of the Entity Bean that holds its state. In that way, the client does not have to retain a parallel copy of the Stateful Session Beans state, it only has to retain the primary key of the Session Bean's stateful Entity Bean, and can recreate a new instance of the Session Bean by merely re-initializing it with the primary key of its stateful Entity.

WebSphere has invested a great deal of effort in to supporting container management of Entity Beans. In many scenarios WebSphere can provide better total systems throughput with container managed persistence (CMP) than an application can achieve on its own with direct JDBC calls. WebSphere has introduced support for declaring access intent policies on CMP Entity Beans. These policies can be used to declare whether a given method of the CMP is going to update the state of the bean. If the state of the bean is never updated then the database does not have to be updated. Another policy can be used to indicate whether the deployed application must be protected with a pessimistic concurrency model. If concurrency can be managed optimistically, then locks are not held on the database, the before image of the database state is retained, and then used to perform an overqualified commit on the database at the end of the transaction. If any changes occurred in the database then these will be detected in the over-qualified commit and the transaction will be rolled back. Further changes are expected in the future to make over-qualified commits even more efficient.

Access Intent policies can also be used to identify which entity relationships are likely to be navigated. The container will combine activation of the entity and those related entities that are mostly likely to be navigated into a single I/O to the database.

Further the EJB container has been implemented with a transactional data cache for CMPs. This can help reduce the number of I/Os that need to be performed between operations in the same transaction. Moreso, if you don't need absolutely current entity state, the state of the entity held in cache from a prior transaction may be used in subsequent transactions (yes, the EJB container does cache state between transactions, but does so in a way that limits resource consumption, and maintains strong failover and recovery).

However, if Entity Beans are shared amongst different client applications there is a good chance that how each client will use the entity will be different - each following its own usage patterns; selecting invocation options that may or may not traverse different object relationships, or causing state to be updated where other clients do not, etc. The efficiency of the CMP can be further refined through the use of Application Profiles. An application profile is a particular access intent template representing a particular usage pattern. Multiple application profiles can be created for the same Entity Bean each representing a different usage pattern. Clients of the bean are tagged with specific Task Identities. This Task Id flows with any requests from that client to the Entity Bean. Task identities can then be mapped to an application profile that best represents its desired access pattern. The container will then manage the bean in accordance to the selected application profile for that specific client. In this way, you can tailor access patterns that best match the needs of different client rather than assume the worst case for all of them. The result is optimized total systems throughput for that mix of applications and client usage scenarios.

There are only a few cases where we would not recommend the use of CMP Entity Beans in your application – primarily only those cases where you are subject to a very complex data schema, perhaps driven by another existing application. Otherwise, *CMPs are ready for prime-time* and should be presumed within any contemporary J2EE application design.

Turning now to Adapters, the Java 2 Connector Architecture suggests that applications should not code directly to the Common Client Interface, but rather should exploit container-specific extensions that provide a higher-level standard interface to the resource-adapter. We refer to this extension as an *adapter* (short for the EIS resource-adapter interface). Web-Sphere Studio provides tooling support for creating adapters. The adapter is just a Stateless Session Bean that supports an application specific interface representing the business operations that are being adapted, and then is implemented to encapsulate the lower-level calls to the Common Client Interface defined by the J2CA specification.

Web services are really not a component model, but rather provide an abstract representation of a canonical component model – one that can be generally mapped to a variety of underlying concrete component models and related client programming models. We'll discuss web services in more detail a little later.

Message Driven Beans (MDBs) are defined as EJBs in the J2EE specification. However, the functional role of an MDB is to map a received message to an operation on a business function component (usually a Stateless Session Bean). The pattern followed by most applicationwritten MDBs is to receive the message through the onMessage operation, and then perform a select statement on the message to determine which Session Bean operation to invoke. In this way, the MDB generally does not contribute to the business logic of the application, but rather is really just a mapping layer between the messaging infrastructure and the application. It is in many ways the inbound corollary of adapters in the J2CA framework. We refer to them as infrastructure components.

Other infrastructure components include J2CA resource adapters – that is, the J2CA component responsible for the actual mapping to the specific protocol, security, transaction, and connection management constraints and capabilities of the underlying enterprise information system (EIS) supported by that resource adapter. Java Management Extension (JMX) Management Beans (MBeans) is also an infrastructure component. The MBean component model is introduced by JMX to represent individual computing resources in the middleware system. WebSphere provides a number of MBeans to represent the Containers, Transaction Manager, Connection Manager, Applications, etc. in the WebSphere execution environment. These MBeans enable applications (and other system utilities) access to the system resources to control their operational state and configure their behavior. WebSphere allows applications to introduce their own MBeans and register those with the Web-Sphere management system. In this way, an Application can be programmed with conditional behavior that is controlled through a corresponding MBean by an administrator of that application.

Like Portlets, MBeans are not explicitly required by the J2EE platform specification. However, they are a pre-requisite of the J2EE Management Model defined by JSR-088 which will be required by J2EE 1.5.

WebSphere also introduces a number of proprietary component models. These include a Business Process (workflow) and Microflow component model. Business Process Choreography is based on the emerging Business Process Execution Language for Web Services (BPEL4WS) specification written by IBM and Microsoft with support from BEA. Both Business Processes and Microflows encapsulate activity flows - scripted definitions of how to process one activity after another, or in parallel with other activities, and how data flows between those activities. This will be discussed in more detail below. The primary difference between a workflow and a microflow is whether the state of the flow is persisted between activities, and thus whether the flow can be interrupted. Microflows are intended for scripting interactions over short-lived sessions with a single, or small number of, target components. It is used primarily within the implementation of J2CA adapters to describe the interaction with the EIS for a given business function supported on the interface of the adapter. Business Process workflows, on the other hand, are intended to script a series of high-level business activities, and may take a long time - on the order of minutes, hours, or even months - to complete. As such, their execution state is persisted between activities in the flow, and thus their flow can be interrupted, suspended, and resumed later.

BPEL is the execution language of a Business Process or Microflow component. However, from a Java programming perspective, these components are implemented as a Stateless Session Bean and Entity Bean – the Session Bean to represent the process and the Entity Bean to capture the process' execution state.

Async Beans are designed to provide the application a safe way of spawning multiple threads of execution. The Async Bean captures the execution context of the application, and copies that to each thread that is spawned by the application. In this way, the threads can be accounted for by the container, and the thread can execute within the fullness of an application context; allowing it to exploit the standard J2EE programming model and operate as though it is part of the main thread of the application that spawned it. The Async Beans service also offers a Scheduler facility that allows asynchronous work to be created and scheduled to be executed at a later time – even hours, days, or weeks later.

Startup Beans provide the application notifications of different server states – specifically, when the server is started and stopped. If starting the server is tantamount to starting the business represented by the application hosted by that server, then subscribing to these notifications allows the application to perform start-of-business and close-of-business operations that may be relevant to it. For example, retail store applications like to be able to take a snapshot of their cash position at the opening and close of business. They can do this in a Startup Bean.

Business Rule Beans (BRBeans) can be used to isolate business logic that may change more often than the rest of the application design. For example, business policies and government regulations may be much more dynamic than the core business logic of an application. These policies can be isolated in a BRBean and managed on a lifecycle that is independent of the rest of the application. Moreso, the policy selection can be conditioned by application-specific entity classifications. For example, if customers can be classified as being Gold, Silver and Bronze level customers - perhaps based on the level of business they conduct with the enterprise - then they may be subject to different policies. The BRBeans framework is a policy selection router that will consider application defined entity classifications to pick the policy that is most relevant to the entity in question. Further, certain policies may not go in affect immediately, but rather may be postdated. The BRBeans framework will consider this before selecting a particular policy implementation.

WebSphere is working on a couple of emerging component models. Service Resources represent a stateful grid service as defined by the Open Grid Services infrastructure (OGSi) specification as published by the Global Grid Forum (GGF). This is a merger of web services and a stateful component model such as EJB Entity Beans. A Service Resource can be instantiated through an explicit or implicit factory, and is assigned an identity that allows the application to differentiate distinct instances of the service.

Another emerging component model is designed to assist in encapsulating and communicating state - Service Data Objects (SDOs) [SDO]. A Service Data Object captures business state acquired from an underlying persistence source in a way that can be easily serialized and transported between components of a distributed application. The SDO is acquired through a data mediator that is responsible for mapping the SDO state to its underlying persistence form. WebSphere will introduce mediators for JDBC-based relational data systems and for EJB Entity Beans (representing the most common BMP and CMP application designs for J2EE applications).

#### Web Services

A great deal of the original discussion in the industry about web services has been centered around the use of SOAP over HTTP. In fact, web services has much broader value than that. Web services is first and foremost about services oriented architecture. Web services is an XMLoriented specification of SOA. It's primary value is in enabling a loosely-coupled distributed systems specification for service-based interactions between participating applications using internet technologies [WSA].

As we've already state, the core of web services is WSDL. WSDL leverages XML as a highly expressive specification of what the service does and how to communicate with it. WSDL is partitioned into different segments to distinguish the abstract specification of the services' interface (the PortType) from the particulars of the protocols, encodings and endpoint addresses of the service (the Binding). This separation allows programmers to focus on what they have to code in their programs, and allows deployers to know what they need to arrange a physical connection to the service.

Web services has been promoted as leveraging internet technologies – specifically HTTP and XML (in the form of a SOAP message structure). Certainly, one significant usage scenario for web services is to enable connection of business partners and home users to the enterprise. This, in turn can make valuable use of the internet as an underlying backbone. HTTP and XML is ubiquitous, and network infrastructures are already well formed for enabling the communication between institutions. However, there are a couple of problems with this focus.

First, HTTP is an unreliable communication protocol. Messages may need to be resent in case they are lost, and there is no assurance that having done so that duplicates of the message won't show up at the service. Also, much of the firewall support that web services over HTTP exploit to gain access to inter-enterprise services is designed with the assumption that HTTP traffic is just carrying harmless web pages to a browser, and some fieldlevel user input that is used to determine the next page to send. On the other hand, web services offer the opportunity to remotely drive powerful business functions and to exchange large quantities of potentially valuable business information. This suggests that the firewall support for HTTP traffic may be entirely inadequate for protecting the enterprise from attacks that can be mounted using the power of web services flowing with SOAP over HTTP

Further, where web services are being exploited within an enterprise for intraenterprise application integration, the I/T infrastructure may already be layered on other robust, efficient, and fully deployed network backbones. For example, many enterprises already have a MQ-based or similar messaging backbone in place, along with instrumentation, tuning procedures, intermediation, content-based routing, and other mechanisms for managing the enterprise I/T infrastructure that web services could benefit from. This represents one of several cases where an enterprise will want to be able to flow web services requests over network protocols other than just HTTP. WSDL allows that through alternate bindings, and Web-Sphere provides support for communicating web services requests over HTTP, JMS, and locally optimized in-memory Java Object calls (the latter is useful when the service happens to be deployed in the same JVM as the client that will use it).

In addition, WebSphere supports a web services gateway - a communication proxy that can receive web services requests flowing over one protocol and convert them to flow over another protocol. This might be useful, for example, if you want to deploy a web service for use both within an enterprise and also be accessible over the internet to other business partners or home users. The service may be deployed for intra-enterprise communication with support for JMS bindings. The web services gateway can then proxy that same web service to the internet with an HTTP binding. This also helps protect other services from being exposed over the internet that perhaps you want to remain private for internal intra-enterprise use.

WebSphere R5.02 introduces support for the Java standards for web services. Specifically, it supports the JAX-RPC (JSR-101) client programming model for invoking a web service within a J2EE application, and supports the Web Services specification for J2EE (WSEE - JSR-109) for deploying web services implementations. In WebSphere, web services are implemented as Stateless Session Beans. With few exceptions<sup>2</sup>, any Stateless Session Bean can be made into a web service. This fits well with the idea that Session Beans represent the activities of the application. The highest level activities are, in essence, the business services of the application. A Session Bean can be made into a web service in the Web-Sphere Studio tooling. This will create WSDL for the Bean, and then generate the deployment descriptors for the Bean that the container needs to manage the Bean as a web service.

It is worth noting here, that any of the component types identified earlier that

are rendered in WebSphere as Stateless Session Beans, including application defined Session Beans, J2CA Adapters, and Business Process components can be deployed as web services. This enables, for example, legacy application functions such as CICS transactions, etc. to be adapted through a J2CA connector, and then expressed as a web service. Even if you don't use WebSphere to build J2EE applications, you can use it to expose your legacy functions as web services.

#### **Business Process Choreography**

WebSphere supports the execution of business processes that can be defined using a rich set of constructs. A business process consists of a number of steps, called activities, which represent invocations of tasks on behalf of the process. Those tasks can be web services in the broad sense of the previous section, i.e., any piece of code invoked by a suitable binding, from a synchronous local Java call to an asynchronous SOAP over JMS invocation [Leymann et al. 2002]. Tasks can also be human tasks, allowing for the incorporation of people into business processes - from the perspective of the business process, a human task is an asynchronous service with a special binding. Actual execution of the activities of a business process is prescribed by partial ordering constraints on the set of activities. In other words, a business process is a directed graph where the nodes are activities, and the edges ostensively show which activities need to be executed in which order, or can be executed concurrently. WebSphere's business process container uses those constraints to minimize the elapsed time for the execution of a business process by executing activities in parallel wherever possible.

As has already been mentioned, the Business Process Execution Language for Web Services (BPEL for short) [BPEL4WS] is the language used for the specification of business processes. BPEL provides the syntactical means to describe a business process where the tasks are web services, and where the business process itself is a web service – typically, but not necessarily, a stateful one. For that, it provides constructs for the composition of elementary inbound web service calls (receive, receive-reply) or outbound web service calls (invoke)

<sup>2.</sup> WebSphere does not currently support custom serializers and so the arguments of the Session Bean must be serializable and support standard mappings to XML types.

into more complex activities, such as sequence (a number of activities needs to be executed one after the other), switch-case (based on the evaluation of a predicate, one of several activities is executed), while (an activity is looped as long as a predicate is evaluated to true), or flow (a construct allowing to directly specify a flow graph using activities and control links). To allow dealing with long-running and thus stateful business processes, BPEL introduces the ability to identify a concrete instance of a business process using correlation information from key fields of the invocation message itself - it is then the responsibility of WebSphere, rather than the client application, to route the message to the right instance [Kloppmann & Pfau 2003].

Human tasks are special kinds of activities whose implementation consists of two parts. First, the people who should perform the activity are resolved. Rather than statically assigning user IDs to activities, WebSphere allows to define people responsible for an activity using staff queries, which are abstract query verbs against the WebSphere user directory. This enables assignment of people based on dynamically determined properties, such as the membership in a certain role or organization. The verb set is extensible to allow supporting directory schema extensions. It is also possible to incorporate context from the business process instance into staff queries, e.g., to have an activity performed by the same person who performed an earlier activity of the same instance. Once the responsible people have been resolved, in the second part WebSphere needs to interact with them so that they can actually perform their tasks. This is done using a browser-based client that presents tasks to users, allows them to claim tasks to exclusively work on them and to complete tasks passing a result, thus letting the business process to continue. WebSphere comes with such a client out-of-the-box that can generically work with any process, but it also provides all the APIs needed to produce a custom client easily.

BPEL is a web services specification that only uses other web services standards, such as WSDL, XML schema or XPath, and is completely agnostic of possible hosting environments [Leymann & Roller 2002]. Hence, WebSphere provides a mapping of business processes into a J2EE environment, and also provides extensions to enable direct access to the hosting J2EE environment from within business processes. A process is made available to a J2EE programmer as a stateless session EJB implementing the operations that the process provides, according to its BPEL interface (as we have seen in the previous section, it is through this stateless session EJB that the business process will be provided as a web service to its clients). For stateful processes, there also is an associated CMP entity EJB, responsible for storing the state of the instances of that process, and for retrieving a particular instance based on correlation information. WebSphere extends BPEL to support activities that are implemented by inline Java code (»Java snippets«) - these activities run in the J2EE environment of the EJB representing the process. WebSphere similarly supports to specify expressions (mainly condition predicates, but also timeout expression) directly in Java, in addition to XPath.

While business processes in general are long-running and stateful, WebSphere also supports the execution of stateless, short-running flows, called microflows. Microflows are used to script together a number of service calls such as J2CA invocations or message transformations. A microflow is also specified using BPEL, but given that it is stateless, it can only synchronously invoke other services, and does not allow for the incorporation of people. Also, its J2EE representation is via a stateless session EJB only - no entity EJB to keep state is required. A microflow always implements exactly one operation.

Business processes are not typically created using a text editor; BPEL is not intended to be a language written by a developer. Rather, the WebSphere Studio tool set features a graphical editor that supports the creation and modification of business processes, visually representing the flow graph with its activities and control links. A new activity can be added by dragging and dropping a service onto the process, and then wiring it with the already existing activities. The graphical editor duplicates as a debugger, allowing to set breakpoints, debug concurrent paths of a process, single step through a process activity by activity, look at and manipulate process data, and in general perform all

standard debugging tasks known from Java debuggers, but at the graphical business process level. Seamless debugging at the business process level and at the Java code level is possible within a single tool.

#### WebSphere EJB Container

As we've indicated at several points throughout this paper, the EJB container is responsible for managing EJB components – for supporting the systems infrastructure encapsulation boundary designed into the EJB component model. We will spend a few moments in this section of the paper detailing some of the basic architecture for the EJB container.

When the container is first initialized, it interacts with the WebSphere management system to determine what applications are 'installed' on the container, and what resources those applications depend on. It loads the metadata associated with the application and other relevant container configuration data, such as the default datasource, bean cache size, cleanup interval and passivation directory into internal control structures. Part of the container and runtime initialization includes allocating the work-threads that the container will use to execute in-bound EJB requests, and putting those threads aside in the thread pool, and allocating a number of bean instances - Object shells and putting those aside in the bean pool.

Requests that are targeted to an EJB are received by one of the end-point listeners for the various communication stacks supported by WebSphere. These end-point listeners are not literally a part of the container, but are a part of the Web-Sphere runtime environment in which the container exists. The end-point listeners are responsible for receiving in-bound messages that encode a request to an EJB. A part of the communication stack will include the processes for de-marshalling the execution service context that may have flown in the request message. This can include, for example, the security session information for the requesting principle, the client's internationalization locale, any transaction context that brackets the request, etc. A minimum amount of context de-marshalling occurs before scheduling the execution of the request. This minimum context is used to classify the request to determine it's execution

priority and to determine which worker thread it should be put on.

When dispatching the request, the runtime will migrate the request to a worker thread. To keep the cost of this movement as small as possible, only the minimum amount of context is demarshalled as needed to determine a requests dispatching priority. The remaining context and request arguments are demarshalled after the request has been aligned to its worker thread. That also frees the end-point listener to then move on to asynchronously process the next in-bound request message.

To maintain a high degree of flexibility and modularity, the runtime does not demarshal the execution service contexts directly. It delegates this to the various service managers that are installed in the runtime through a set of interceptors. That way, each service manager can take responsibility for interpreting their own service context formats and semantics. And additional service headers can be handled by simply adding new service manager interceptors. After processing, each service manager will form a service context control structure for the request and associate that with the thread of execution.

The container is invoked by an object adapter in the message listener to resolve the identity of the EJB targeted in the inbound request. The container then demarshals the encoded object identity in the request, including its object type, and attempts to resolve that against the previously activated objects in the bean cache. If the referenced object has not already been activated, and if there is room still available in the cache, then a bean instance is allocated from the cache. That bean instance is mapped to the target identity of the request. If the class for the targeted object has not already been loaded, then the appropriate class loader is instructed to load the bean's class, which in turn may, if this is the first time the application has been started, activate initialization of the class loader and the bean's manifest environment in it's component archive

The container maintains a mapping of the external component identity as indicated by the request to the specific bean instance in the bean pool. Subsequent requests to the same component then are routed directly to the cached bean instance.

Once the bean instance has been activated, its skeleton can then be used to demarshal the remaining arguments - mapping them then to the argument types expected by the bean implementation. Invoking the operation on the target bean results in a container-breach. This is a signal to the container and its supporting service managers to process the qualityof-service policies for the target bean. Again, each of the service managers are engaged through a collaboration framework to test the service context against the state of target bean. Authorization policies are verified; local JNDI java:comp context is created, transaction policy is executed; internationalization policy is applied; etc.

The target bean is then subjected to any remaining component lifecycle requirements. For example, CMP bean state may be loaded and mapped from the schema of the persistent store for that Entity.

A significant portion of the EJB container includes the persistence and connection managers. The persistence manager is responsible for processing the schema mappings between the beans abstract schema definition and the concrete schema of the bean's persistent data store. The persistence manage also manages an inmemory transactional cache that can be used to reduce I/O to the data system, as well as fulfill some of the access intent optimizations that are possible. The connection manager is responsible for creating, pooling, validating and handling shared connections to the underlying data system.

Finally, once all of the execution context has been established and validated, and the bean state has been loaded and initialized, the request may be dispatched on the bean's method implementation.

If the bean implementation should invoke a request on another component, even if it is in the same application server JVM address space, it undergoes another container breach and is subject to the same processing again to validate the policies of the new bean target.

As you can tell, the overhead of dispatching a component can be quite high – although most of what's been described here is a worse-case scenario; much of the processing can and is optimized out for cases that do not need it all. This contributes to the concerns discussed earlier about how important it is to choose remote-local component boundaries carefully - the granularity of the component model will determine how much of this component management overhead shows up in the overall execution path of the application. However, assuming the right level has been chosen for the application design, container management of the component will go a long ways towards ensuring the integrity of the application execution, and greatly simplifies the I/T infrastructure concerns for the application developer - basically the application developer can ignore integrity issues and put that burden on the middleware; freeing them to concentrate their attention on their domain requirements for the application.

#### Scaling and Clustering

Container management represents one dimension of scaling – classifying workload, prioritizing dispatching, maintaining a stateless execution environment all go towards increasing the throughput of applications hosted on the application server. However, at some point, the application server can only process work as fast as the computer processor, memory bus, file system channels and network adapters will allow. At some point, scale will be constrained in a single application server instance by the underlying computer hardware, operating system, and available resources on that computer.

However, WebSphere supports vertical and horizontal clustering [Modjeski et al. 2001]. Vertical clustering (multiple application server instances running on the same computer) can be leveraged to overcome bottlenecks or constraints in a single operating system process. Often we see constraints on the amount of buffer space or the heap size that can be allocated to a single operating system process. We also see cases where allocating more threads of work to a single JVM just increases the size of hash tables and other execution control structures and results in them working less efficiently. In any of these sorts of cases, it may be beneficial to increase the number of application server processes on a single computer rather than to continue to increase the amount of work being directed to a single process instance.

Horizontal clusters (multiple application server instances running on different computers) can be leveraged to increase the number of computers and related resources that can be allocated to serving an application. This might be useful to overcome hardware constraints, including the number of CPUs or amount of physical memory, on a single computer.

A cluster can be composed vertically, horizontally, or a mix of the two. A cluster is defined to be a collection of application server instances that act as a single logical application server - all of the cluster members are configured to run the same applications and use the same application resources (that is, J2CA resource adapters, data sources, JMS queue and topic factories, etc.). A cluster should have equal access to the underlying data systems that it depends on - although how this achieved is a subject for another paper on data system scaling technologies, including through the use of data system clustering, mirroring and replication. Nonetheless, the application server cluster appears to the administrator to be one logical application server hosting a set of applications. WebSphere clients the Edge Server, Web Server plug-in, Web Application Container, EJB Container, and the J2EE Client Container - are all implemented to understand WebSphere clustering. These clients will distribute workload across the cluster in either a round-robin or random fashion.

Each member of the cluster can be assigned a different weighted value in which case workload will be distributed to the different members proportional to those weightings. This is useful if you determine that some of the computers in the cluster have more capacity than others perhaps due to differences in the hardware, or due to some of the computers being shared to host other workloads. R5.02 of WAS-Enterprise edition introduced support for dynamic workload management. The workload system monitors the performance of each application server. If it sees that one server is slowing down, it will automatically reduce its weighting value, reducing the amount of work that is routed to that server instance. When the server picks back up, the weighted value is restored and more work will be routed to it.

#### On Demand and Grid Computing

One of the most significant problems facing I/T shops today is with containing their costs - getting more utility from their computing infrastructure [Ackermann et al. 2002]. As we look at the computer industry today, we see that computers tend to be heavily under-utilized. One major reason for this stems from the fact that applications tend to be set up as islands - with their own set of dedicated computing resources. A set of computers will be installed to host an application and will be provisioned with enough capacity to handle the greatest demand that application will ever see. Over a number of applications, and at any given point in time, this results in an tremendous amount of »spare« capacity.

On demand computing seeks to change that situation. On demand computing focuses on key areas of computing with the intent of making computing infrastructures behave more like a utility grid, and for enabling businesses to respond more efficiently to their own business demands. The focus for on demand computing includes [IBM ODOE]:

- Operating environments
- Infrastructure management and provisioning
- Virtualization
- Autonomics
- · Utility computing
- · Business responsiveness

If you watch the utilization of an application over a period of time you will begin to notice it oscillates throughout the day and over the week or month. This is often the result of the nature of the application and its user behavior. Discount trading applications, for example, tend to cater to day-traders and home consumers - who tend to use these applications during the lunch hour or in the evenings, after work. Accounting applications tend to hit their peak utilization at the end of reporting periods when financial results need to be calculated and summarized. Bank Teller applications tend to be at peak utilization at mid-morning and late-afternoon. And so forth. Over a large enough business area, the peak utilization curves of the aggregation of applications in use by that business tend to cancel each other out the total utilization tends to be less than the peak sum of the parts.

With this in mind, the on demand operating environment is being designed to enable I/T data centers to pool all of their computing resources for a set of applications, and then manage workloads over a dynamically configured system [ODOE]. This will leverage clustering, workload distribution and automated configuration management to automatically increase or decrease the number of application servers that are assigned to support a given application in a cluster. As utilization demands increase for a given application, the number of application servers, and thus the amount of computing resource, assigned to the application will be expanded. Other applications that are experiencing less demand at the time will be scaled back to consume less resource - giving that resource over to those applications that need it.

This sort of dynamicity can not be entirely reactive. In some cases, the reaction time of the system may be too slow for certain critical business functions. For those functions, it may be important to always reserve a certain amount of un-used capacity just so that it is there for when demand spikes. In other cases, careful analysis of the utilization history may reveal that demand always increases at a certain time of day, and so you can anticipate this need in advance. Moreso, overall utilization trends may suggest that demand for the entire portfolio is increasing (business is good!) and that the capacity of the entire system needs to be increased.

Tivoli is working on provisioning and policy systems that maintain service-level policies for each application, and will govern the allocation of on demand resources to meet the service-levels required for each of those applications [IBM Tivoli]. These infrastructure management and provisioning systems will plug-in to the open management APIs provided by WebSphere in the form of JMX and the resource configuration service. They will drive WebSphere to reconfigure its clusters and adjust its workload routing decisions to ensure service level objectives are being met. This management system will plug-in to the Web-Sphere Performance Monitor Infrastructure (PMI) and Application Response Monitoring (ARM) instrumentation, to collect response time and resource utilization metrics that inform the management system on where bottlenecks may

be forming in the system or what sort of system utilization is actually being experienced, and feed that back into the provisioning decisions.

To benefit from such a dynamically configured system, applications must be shielded from the underlying information system. This returns us to an earlier theme - there needs to be a strong separation of concerns between the business application logic and the I/T infrastructure. In this case, the middleware needs the flexibility to migrate the application to use different parts of the system in accordance to capacity, utilization, and demand requirements that are formed outside of the application. The middleware needs to virtualize the underlying information system so that, like a virtual-memory system, it can manage the logical dependencies of the application against the physical resources at its disposal.

As large scale systems grow increasingly complex - as more computers and resources are added to the environment; as more heterogeneous variety is included; as workload disparity increases; as the mix of underlying infrastructure technologies and application dependencies becomes more intertwined; as more and more resources become virtualized - the task of managing the system is slowly drowning the administrators that have to keep these systems going. The only way that enterprises will be able to benefit from large scale, heterogeneous distributed computing systems acting like a utility grid is if the grid environment is autonomic - that is, self-managing; self-healing, self-configuring, self-tuning, and self-protecting [IBM Autonomic]. Autonomic systems are needed to reduce the burden on administrators.

Elements of the MAPE-loop - that is, Monitoring, Analysis, Planning and Execution of autonomic responses to systems management, all centered around a common Knowledge base of system semantics and administrative policies - have already been instrumented into Web-Sphere. Again, PMI and ARM instrumentation has been built in to WebSphere to monitor some hundred or more operational metrics - from the size and utilization of the thread pool, to the number of in-flight transactions, to the number of connections open to a given database. These can be used to inform the monitoring system precisely what's going on in the WebSphere runtime for a given application.

A knowledge base of common performance and problem symptoms is being accumulated from real customer experiences. These symptoms can be applied through the WebSphere Log Analyzer and Performance Advisor included with the Tivoli Performance Viewer - both included in WAS R5 - to diagnose a situation and collect advise on how to correct it. In the future, these utilities will be transitioned from simply providing advice, to automatically acting on the advice to adjust the system for peak performance. As with the provisioning system for service level agreements, tuning and problem response management can be controlled through a set of policies. For examples, a policy might be set to condition the performance tuning system to make certain basic configuration changes within a normal operating range, but to get confirmation from an administrator if certain thresholds are exceeded. For example, an error in a new application may be inadvertently generating more demand for a certain service than should be expected. Rather than reconfigure the system to satisfy that demand, the administrator may want to intervene to prevent that application from taking over their entire system.

Similar mechanisms are being put in place for problem management and security.

Employing large scale distributed computing grid technologies for efficiently hosting enterprise applications within an I/T data center is one form of utility computing. A broader form of utility computing can be formed as an outsourcing offering by 3rd-party service providers. We expect a number of service providers will take advantage of the even larger economies of scale that can be realized by hosting not just the business applications of a single enterprise, but rather, hosting the applications of many enterprises, including small and medium businesses. These environments will introduce new problems for data, process, administration and failure isolation - no one business being hosted in the utility wants to be affected by a failure or rogue applications introduced for another business. Again, WebSphere is incorporating infrastructure to enable this type of isolation within a utility computing environment.

Which brings us, finally, to business responsiveness [IBM Global]. If there is any ideal goal for information computing, it is to transition from being a cost-of-doing-business to being a business asset that is, something that not only enables a business to conduct business, but also something that provides a return on investment by enabling a business to conduct business better; in ways that are more profitable: in ways that could not be performed without it. On demand computing is about integrating enterprises across their lines of business - to make the enterprise as a whole more efficient and synergistic, and more responsive to the marketplace.

Integrating the lines of business can reduce costs by streamlining operations and enabling parts of the business to realize downstream affects of their business decisions. For example, integrating the order entry system with the supply system allows the marketing organization to focus their advertising campaigns on real supply and demand information. It helps the pricing organization to make adjustments in the supply chain before making a pricing change. It helps the human resources organization make better workforce decisions.

But more importantly, when this integration is then coupled with business process management it allows business analysts and business managers to adjust their business processes rapidly and in response to real-time shifts in the market place. If world oil-prices change, business managers can adjust their order fulfillment and logistics processes to compensate, or shift their sales processes to emphasize certain regional markets. Using business process choreography and scripting tools, the information system can be used by business managers to change the way their business operates, enabling them to respond much more rapidly to market conditions and opportunities to given them competitive leverage.

These are the real goals of on demand computing – to leverage on demand resource management within the I/T infrastructure to enable business that respond on demand to their market and business conditions and opportunities.

# Conclusions

The primary goal for WebSphere is to merge the world of web-based computing with the world of enterprise-computing, to enable customers to exploit emerging and rapidly expanding business opportunities, across a range of platforms and scale, at the lowest cost of ownership, with a consistent and synergistic end-toend experience for information computing. We can return application programmers back to the task of building sustainable value and return on investment to the business by enabling them to spend more time creating flexible, durable and re-usable business assets and less time creating I/T infrastructure. WebSphere accomplishes this by incorporating fundamental design principles for middleware with a programming model that separates concerns, and then layers that on an infrastructure that enables scaling from the smallest of pervasive devices through to high-end corporate I/T data centers and utility service providers. WebSphere is the foundation of on demand computing to reduce the cost of computing, and to enable information systems as a competitive advantage to businesses.

## References

- [Ackermann et al. 2002] Ackermann, J.; Agrawal, A.; Tinaikar, R.: Viewpoints: A Plan of Attack for Cutting Information Technology Costs. McKinsey and Company, 2002. Available online at www.mckinsey.com/knowledge/articles/viewpoints.asp.
- [BPEL4WS] Business Process Execution Language for Web Services (BPEL4WS). Version 1.0. IBM, Microsoft, BEA, July 2002. Available online at http://www.ibm.com/developerworks/library/ws-bpel/.
- [Burbeck 2000] Burbeck, S.: The Tao of e-Business Services, IBM Corporation, 2000. Available online at http://www-4.ibm.com/software/developer/library/ws-tao/index.html.
- [Everyplace] Everyplace Toolkit for WebSphere Studio – Product Overview – IBM Software. Available online at http://www-3.ibm.com/ software/pervasive/everyplace\_toolkit/.
- [IBM Autonomic] IBM Autonomic computing. Available online at http://www-3.ibm.com/ autonomic/index.shtml.
- [IBM Global] IBM Global Services: Tighter budgets, better business: Extracting value from the enterprise in a downturned economy. Available online at http://www-1.ibm.com/ services/files/GSEE510301500F-budgetpressure.pdf.

- [IBM ODOE] The IBM on demand operating environment. Available online at http://www-3.ibm.com/software/info/openenvironment/.
- [IBM Patterns] IBM Patterns for e-business Resources. Available online at: http://www-106.ibm.com/developerworks/patterns/library/.
- [IBM Tivoli] IBM Infrastructure Management. Available online at http://www-3.ibm.com/ software/info/openenvironment/infra-mgmt/ solutions.html.
- [IBM WebSphere] IBM WebSphere middleware, application server, e-business, infrastructure software. Available at http://www-3.ibm. com/software/info1/websphere/index.jsp?tab =highlights.
- [J2EE] Java 2 Platform, Enterprise Edition. Available online at http://java.sun.com/j2ee/.
- [JSR168] The Java Community Process (SM) Program – JSRs: Java Specification Requests – detail JSR# 168. Available online at http:// jcp.org/en/jsr/detail?id=168.
- [Kloppmann & Pfau 2003] Kloppmann, M.; Pfau, G.: WebSphere Application Server Enterprise Process Choreographer – Concepts and Architecture, IBM, 2003. Available from Process Choreographer section on WebSphere Developer Domain: http://www7b.software.ibm.com/wsdd/zones/was/wpc.html.
- [Leymann et al. 2002] Leymann, F.; Roller, D.; Schmidt, M.-T.: Web services and business process management, IBM Systems Journal Vol 41, No 2, 2002. Available online at http:// researchweb.watson.ibm.com/journal/sj/412/ leymann.html.
- [Leymann & Roller 2002] Leymann, F.; Roller, D.: Business processes in a Web services world. 2002. Available online at http://www-106.ibm.com/developerworks/webservices/library/ws-bpelwp/.
- [Modjeski et al. 2001] Modjeski, M. et al.: Failover and Recovery in WebSphere Application Server. 2001. Available online at ftp:// vadd1:sunwsfj4@207.25.253.53/1/wsdd/pdf/ modjeski.pdf.
- [ODOE] The on demand operating environment. Available online at http://www-3.ibm.com/ e-business/doc/content/evolvetech/operating\_ environment.html.
- [Röwekamp & Roßbach 2000] Röwekamp, L.; Roßbach, P.: JSP Tutorial, part 2: Model View Controller and Data Base Integration. 2000. Available online at http://www.heise.de/ix/artikel/2000/08/148/.
- [Schneberger 1995] Schneberger, S. L.: Software maintenance in distributed computer environments: system complexity versus component simplicity. Proceedings International Conference on Software Maintenance. October 17 -20, 1995, Opio (Nice), France. Available online at http://csdl.computer.org/comp/proceedings/icsm/1995/7141/00/71410304abs.htm.
- [SDO] Services Data Objects. Available online at http://www.ibm.com/developerworks/library/ j-commonj-sdowmt/.
- [Snyder 2003] *Snyder, B.:* Report: IBM Takes Lead in App Servers, 05/07/2003. Available

online at http://www.thestreet.com/tech/billsnyder/10085744.html.

- [SOA] Service Oriented Architecture. Available online at http://www.serviceoriented.org/service\_oriented\_architecture.html.
- [WASZOS] Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server. Available online at http://publib-b.boulder. ibm.com/Redbooks.nsf/RedbookAbstracts/sg 246283.html?Open.
- [WSA] Web Services Architecture W3C Working Draft. Available online at *http://www.* w3.org/TR/ws-arch/.



**Rob High** is Chief Architect for WebSphere Application Server foundation. He is Distinguished Engineer and a member of the IBM Academy of Technology. He started his career with IBM in 1981, and

has extensive experience in object-oriented and component-based programming, distributed computing, and application integration middleware. He currently works in Austin, Texas, United States.

Robert H. High Jr. IBM Software Group 11501 Burnet Road Austin, TX 78758 USA highr@us.ibm.com http://www.ibm.com



Matthias Kloppmann is an IBM Senior Technical Staff Member and the lead architect for WebSphere Application Server's Process Choreographer component. He joined IBM in 1986 and has worked on a variety of projects, with a

focus on data bases, workflow and business process management. He studied computer science and electrical engineer in Hagen and Stuttgart.

Matthias Kloppmann IBM Software Group Schönaicher Str. 220 71032 Böblingen Matthias-Kloppmann@de.ibm.com http://www.ibm.com