Building Internet applications with Metaserver

A TECHNICAL WHITE PAPER
FROM METASERVER INC.
Introduction

E-business has changed the role of applications in an enterprise. Individual applications exist not only within business units, but also as components within business processes across the enterprise. Increasingly, business applications span multiple business units and even multiple enterprises (B2B e-commerce). These applications use the Internet as a communication medium.

The focus has shifted to building business applications out of loosely coupled ensembles of components (applications) tied together by the Internet (low QoS). These business applications are still critical to the functioning of the business as a whole. Real world examples are:

- Manufacturers integrating their operational systems with those of their channel partners and distributors while simultaneously providing access to the same information to their customers.
- Financial organizations integrating the underwriting of their products with the retailers of those products: e.g. insurance companies providing access to their underwriting systems to their independent agents.

This shift in focus has profound implications on the application architectures and the infrastructure requirements to support these new architectures. The role of the Internet infrastructure and coordination layers that support these new Internet-based business applications becomes vitally important. The key requirements are:

- Easily assemble business applications out of loosely coupled components spread out across different business units and even across enterprises.
- Cannot impose a single platform (Unix or NT) or standard (EJB, COM…) on the solution. This is especially disastrous in the B2B scenario but is usually a major hindrance even within the same enterprise or business unit.
- Must automate or dramatically simplify load balancing, scalability and fault tolerance of the overall business application. This is critical because the components themselves are not necessarily robust or high-performance, the Internet itself has relatively low QoS and the number of users of the system is highly unpredictable.
- Must support and tightly integrate with standards (HTML, XML, CORBA…) and products that implement those standards (Web servers, application servers, ORBs…)
- Must map to the existing skills of the people in most organizations.

The complex requirements of the new business application architectures resulting from the growth of B2B e-commerce and the ensuing dramatic shortage of skilled resources to build those requirements creates an acute need for products that simplify and automate the process. Metaserver’s web router product supplies a key piece of this Internet infrastructure.
Distributed Systems are Difficult to Build and Maintain

Consider Chicago’s O’Hare airport\(^1\). This big, complicated system needs to work with an extreme degree of reliability. The activities of hundreds of airplanes need to be coordinated with high precision to ensure that they don’t collide with each other. The planes need to be maintained and fueled. Pilots have to fly the planes, be warned of upcoming thunderstorms, etc. How does this system work? There is no single entity that understands the entire system. No one person is in charge of everything.

The system works because the airport has many *individual workers* who are very well trained. They each know how to do only one thing. They do that one thing, however, very well. Pilots know how to fly, air traffic controllers know how to direct traffic, weather forecasters know how to forecast, and mechanics know how to maintain the planes.

However, the workers need some support in order to function. So, O’Hare has an *infrastructure*, which provides services to the workers. It has radar for the controllers, wind vanes for the forecasters and screw drivers for the mechanics. It provides mechanisms for them to communicate with each other (telephones, radios, paper routing slips, etc.). The infrastructure also provides a *coordination mechanism* for the workers. It ensures that the right workers do the right things in the right order to perform a given task. Every task consists of a list of things to be done. For example, turning a plane around may consist of:

- a weatherman OK’ing the wind speed,
- an air traffic controller finding a landing window,
- a pilot landing the plane on the runway,
- a gate agent guiding arriving passengers off the plane,
- a mechanic checking and refueling the plane,
- a caterer cleaning and restocking the plane,
- a gate agent helping passengers board the plane, and so on.

There is a “to do” list for every routine activity that takes place in the airport. There is no supervisor controlling the weatherman, controller, pilot, gate agent, and mechanic for every plane that needs to be turned around. Each worker is trained to follow the routine described in the to do list.

The example of Chicago’s O’Hare airport illustrates the following essential concepts:

- The best way to build complex, distributed systems that need to be reliable, fault tolerant etc. is to use an army of individual, well-tested, well-trained and focused workers. The workers need to know how to do just one thing and they need to know how to do it really well. If all the

\(^1\) The airport example is based on an example from the article “Component-Oriented MIDDLEWARE” by Roger Sessions from the October 1998 issue of Component Strategies magazine. It has been modified significantly to add the coordination mechanism or layer concept.
workers at O'Hare needed to know how to fly a plane, service it, forecast the weather, control air traffic and load peanuts, you couldn't get a single plane off the ground. No single worker could accomplish the task.

- In order to achieve the bigger goal (which is to build the distributed system), the workers will need some support. This support is provided by the infrastructure. The infrastructure provides a context in which the workers operate. It provides a low-level mechanism that the workers use to communicate.

- Every task in the system is composed of the activities of the appropriate sequence of workers. The coordination mechanism must automatically coordinate the activities of the workers to achieve any given task.

- The infrastructure is far more important than the workers themselves. For example, if you replaced all the caterers at O'Hare with better caterers, the airport would probably continue to work fine. However, consider what would happen if you replaced the radios with quadro ray emitters. Quadro rays are far superior to radio waves but the pilots don't know how to use them. Chances are that O'Hare would turn into a giant fireball in seconds. If you replaced the "to do" routines with a human supervisor controlling the workers for every single task, the supervisor would need to be fail-safe. A single supervisor mistake could cause a fireball.
Distributed Software Systems Work in a Similar Fashion

The same principles are true for building large, complicated software systems. These principles become even more important for distributed software systems built out of [pre-existing] individual systems running over the Internet.

The software equivalent of a worker is a **component**. Components need support services (naming, persistence, communications, coordination etc.) in order to be useful. In the context of Internet applications, this support is provided by the **Internet infrastructure**.

**What is a component?**

A component is a piece of software that often encodes a piece of business logic and obeys a predefined set of rules. For example, if it has a color the rules may require it to provide methods called `setColor()` to set the color and `getColor()` to get the color. In exchange for obeying the predefined rules, it gets to use the services of a sophisticated and powerful infrastructure.

A component usually performs a single function. For example, it may validate a user or add registration information to the customer database. It does this single function efficiently and reliably.

Components should ideally be designed to be as fine-grained and stateless as possible. Fine-grained components are usually easier and faster to develop because their function is tightly contained. They perform a single, small but useful function. At runtime, they are often better behaved, consume fewer resources and may be created or destroyed cheaply. Stateless components permit the coordination layer to switch between instances at will. This usually provides better performance, scalability etc.

Components are often created using some sort of “wrapper,” written in a standard programming language such as C or Java, around an existing application. The wrapper is a thin software layer that allows the application to expose its functionality in a manner that is consistent with the rules, and turn itself into a component. In addition, component-based systems have a description technology that provides a software-readable description of the component.

**What is the Internet infrastructure?**

Components are usually not self-contained programs that can run standalone. They are individual pieces of a larger Internet application. In order to successfully build and deploy these business critical Internet applications out of components, the components will need a spectrum of support services. These services are provided by the Internet infrastructure and are shown in Figure 1.

Many products on the market provide some of the functionality (services) shown in Figure 1. In particular, application servers and EAI (Enterprise Application Integration) products have received a lot of press.

The definition of an application server is still not clear. Application server products on the market today all provide some but not all of the services shown above. The generic interpretation of an
application server has often been any product that handles business components and is capable of servicing requests over the Internet (often through a web server). This broad definition is the source of much confusion as many of the services that are actually required for constructing a multi-tier Internet application are not provided by vendors of application server products on the market today. This is particularly true now that the focus is shifting to building applications out of loosely coupled ensembles of components (often existing applications) tied together by the Internet (low QoS). The emphasis then changes from creating and hosting components to coordination and deployment services at runtime.

EAI products mostly provide integration at the data or application level. These products typically include data transformation, message queuing, publish-subscribe technologies, and pre-built connectors. Many vendors are expanding their offerings to include integration at the business process level (process flow diagrams).

A more meaningful understanding of the market and its requirements can be obtained by examining the spectrum of services that are required for successfully building and deploying business critical Internet applications. We have classified these services into the categories shown in Figure 1: Security Services, Presentation Services, Runtime Services, Distributed Object Services, Transaction Services, and Integration Services.
Security Services
All distributed Internet applications need to secure data and applications that are available only to authorized users. Security services usually take the form of identifying the user (authentication) and then granting access to a subset of the available data and applications based on the user’s identity (authorization: either access control list or role-based). All vendors in the market, including Metaserver, provide security services (authentication and authorization) in some form.

Presentation Services
Internet applications developed for thin clients require a mixture of static and dynamic content. The information must be formatted for browsers (HTML), B2B applications (XML), Java, Excel, PDA's etc. Further, client-side state must be maintained over stateless protocols, in particular HTTP.

Most vendors in the market, including Metaserver, provide some sort of state and connection management. Several products also provide support for dynamic content (Cold Fusion from Allaire, XML server from Bluestone) while others rely on generic mechanisms such as ASP or JSP. Metaserver provides template processors for dynamic content using HTML, XML and XSL. Further, Metaserver also supports the use of generic mechanisms such as ASP or JSP.

Coordination/Runtime Services
As discussed earlier, it is rapidly becoming imperative that reliable, business critical applications be constructed as loosely coupled ensembles of unreliable components running on a low quality network. In such an environment, the runtime services layer becomes by far the most important layer.

The functionality in this layer includes:

Coordination: invoking the functionality in several [often-incompatible] components in the appropriate order to provide the desired business functionality at runtime. Most application server and integration server vendors leave this critical piece of functionality to the programmer. The components may be available as EJB or COM or CORBA components. In many cases, the component is actually constructed out of an existing legacy system (a mainframe program or a Visual Basic client-server application). Invoking them in the right order is the responsibility of the programmer. Once the program is written, the application server will manage the instantiation of the program itself at runtime.

Availability: Load balancing, scalability and fault-tolerance. Generally, all these services enable the system to maintain a specified quality of service in the face of changing numbers of users, changes in the environment (servers going down etc). Application servers often provide some functionality in this area but it varies widely and is often qualified by restrictive conditions on the application itself.

Metaserver’s web router technology totally automates these critical pieces of functionality. No programming is required to implement and deploy business applications by assembling them out of components. Several component models are supported, thereby removing the imposition of a single standard on the entire organization. Availability services are also provided out of the box with no programming required. This is detailed below.
Distributed Object Services

Business functionality is encapsulated into objects (often called business components). These components must usually follow one of three established standards: EJB from Sun, COM from Microsoft, or Java/CORBA from OMG. The component model establishes the rules that must be followed in order to create a component.

The existence of a component gives rise to the need for an entity to manage the component. These entities are called containers. The container manages component lifecycle, persistence, and thread/connection pools. In general, the container provides services that make it easier to create robust components. (Note that component lifecycle is managed by object monitors that are sometimes classified as being part of Transaction Services.)

Today’s application servers are primarily containers for components created using one of the three component models above. Microsoft’s MTS and Sybase’s Jaguar are the main offerings for the COM distributed component model. The EJB model has gained wide acceptance with a wide range of application server products that provide EJB support. BEA Weblogic and IBM WebSphere are examples. Most products impose a single standard on all components used in the application.

Metaserver is complimentary in nature to these services. The web router can use EJB, COM or CORBA components but does not supply a container to manage those components. Instead, Metaserver relies on using a container from one of the application server vendors to manage these components. Metaserver also provides a container for components that are created using one of its language-level APIs. The container provides services similar to those provided by EJB or COM containers above and makes it very easy to create robust Metaserver components using the language-level bindings. Using the container supplied with Metaserver, it is possible to build a reliable Internet application without using any application server product.

Transaction Services

Most Internet applications need to reliably store and transfer data at some point in the application. A common example is the need to update enterprise data in a consistent fashion across multiple databases such as reducing inventory by one and simultaneously entering a new order into the order database.

A few large players such as BEA with Tuxedo, IBM with TXSeries and CICS dominate the market for transaction management systems although most organizations don’t use any transaction management system to keep their data consistent. Transaction management is being incorporated into the EJB, CORBA and COM specifications. In most cases, the actual implementation will be provided by integrating with an existing product.

Transaction services may include object monitoring services. This consists of handing out references to objects, re-using objects across numerous requests etc. This makes the system more efficient by reducing the total number of objects in the system, prolonging the useful life of an object etc. However, the distributed object container (application server) usually provides this functionality.

Metaserver provides object monitoring services out of the box via its built-in container. Distributed transaction management relies on integrating with an existing product such as Tuxedo or CICS.
Integration Services

An Internet application usually accesses data and applications that are part of an organization's internal systems. Internet applications may use mainframe applications running under CICS, ERP packages, SQL databases, stored procedures, existing Visual Basic applications etc.

Integration Services provide hooks or connectors into such applications and enable their functionality to be presented in some consistent manner (services, components etc). The back-end systems in the organization are then available for use as part of the Internet application.

Application servers usually differentiate themselves using functionality in this area.

Integration server products such as WebMethods, Novera and Amazon Integrator provide functionality in this area. For example, Novera's Integrator permits the creation of Novera Business Components, which are Java classes. These components are then available to clients as EJB's or CORBA objects.

Again, Metaserver is complimentary in nature to the integration server products since it can use services or components created using these products.

Development Tools

All vendors in the market, including Metaserver, have an IDE or permit the use of a third party IDE such as Symantec's Visual Café, Microsoft's Visual Studio, IBM's Visual Age etc.

Coordination services

The coordination layer allows components to be linked together to create a distributed, Internet application. It allows you to define the sequence in which the functionality of various components must be accessed in order to solve the problem at hand.

At runtime, it automatically ensures that the right components are invoked in the right order to perform the required task.

Application/Integration server products

Coordination must be expressed in the form of a program written in a programming language and compiled into an application or a component using a component model. The coordination layer must then provide a container in the context of which this new application can execute. When the coordination is created using a component, the application server becomes the container.

Application servers construct a thin layer on top of standard middleware such as CORBA and provide their own APIs for accessing component functionality via some sort of remote method invocation. They usually provide facilities for writing CORBA, COM or EJB components and containers (ORB, MTS or EJB server) to host them.

In all cases, one has to write a real program in a real programming language (often a system programming language like Java or C++) and generate a real application that must then execute in the application server. The main function of this program is to route pieces of work to the appropriate components at the appropriate time. The program is analogous to a supervisor who
would coordinate the airport workers for every single task. As mentioned earlier, the supervisor (coordinator) must be fail-safe, and hence, this program is not easy to write.

- It requires developers with expertise in COM or CORBA or Java programming. They often need to have distributed computing experience of some sort.
- It must handle errors and, due to the difficulty of doing so, becomes extremely complex. If one of the components you are accessing goes down in the middle of a method invocation, you must handle the error and take the appropriate action.
- If you want to balance the load among several instances of a particular component running in different containers on different machines, you must do so programmatically.
- If you want to invoke a COM or CORBA method asynchronously, you either are out of luck or must use unnatural technologies like message queues.
- If you want to execute several components in parallel, you must build in the parallelism programmatically. The environment does not automatically support parallel execution of several independent components.

Furthermore, at runtime the program must run somewhere. *Application servers exist solely for this purpose.* This comes, however, with the associated problems of managing these new servers. Many vendors claim that several instances of the application server may be seamlessly clustered together. This is usually not the case. In fact, if an application is in mid-execution on a machine when it goes down, the state of that application is invariably lost.

Most application servers do not support the creation of applications that coordinate the activities of components running in widely heterogeneous environments such as an EJB component and an MTS component and a PERL script.

**Web Router (Metaserver)**

Metaserver's coordination mechanism is implemented in the form of a web router based on the Linda® and Paradise® model of shared network memory (tuple spaces). The web router automatically performs the function of routing pieces of work to the appropriate components at the appropriate time. It removes the requirement of creating a new application that needs to run in an application server process on some computer to handle the routing function.

Metaserver components can be CORBA, COM, JavaBeans, or custom components such as legacy applications, created using a Metaserver API. They are created by wrapping the existing application or JavaBean or PERL script using a thin software wrapper called a MetaLink.

Metaserver provides a high-level mechanism for combining these components to create new web applications. It provides a visual diagramming tool wherein the developer simply drags and drops

---

2 MTS or an EJB server will automate the process if all instances of the component are running within that particular MTS or EJB server but not otherwise. This precludes instances running on different physical machines, which is essential for actually balancing the load.
components from a component palette and wires them together visually to define the route that needs to be taken in order to complete a given task. The route is called a MetaTask™. Creating a MetaTask requires no programming expertise. Error handling, parallelism, and other constructs can be expressed visually in a MetaTask. Metaserver’s web router automatically handles issues such as load balancing, asynchronous communication, fault tolerance, etc.

The MetaTask is stored in the web router’s shared network memory server called the MetaServe™. This layer provides the mechanism for the components to communicate. This communication takes the form of information (tuples) written to and read from the MetaServe shared network memory. Metaserver also provides containers for managing the execution, state, lifecycle, etc. of components created using one of Metaserver’s language level bindings.

The components coordinate their own activities using the MetaTasks stored in the MetaServe. This coordination technology is embedded in the MetaLink and is invisible to the component developer. It frees the developer from the burden of dealing with distributed error handling, load balancing, fault tolerance, parallel execution, etc.

In addition, the lack of a coordinating process removes the need for a separate container (application server) in which it needs to execute. Since there is no application server process, there are no associated issues of managing a new server. There are no performance issues or platform-specific restrictions.
An Example

We'll contrast the two approaches (web application server vs. web router) using an example. Consider the problem of a brokerage firm that wants to create a report describing the performance of a customer's portfolio. The securities in the portfolio are stored in a database that runs on a mainframe. There is a custom application written in C++ that runs on a Unix platform and prices the securities. Finally, a Microsoft Excel spreadsheet graphs the priced portfolio and generates some tables.

Say the brokerage firm wants to build an Internet application to provide access to its customers over the Internet. Customers should be able to use a standard Web browser, go to the firm's Web site, enter their customer identification information, and request the formatted securities report in their Web browser.

The figure shows how this might be implemented with and without Metaserver.

The two approaches are contrasted below.

**Without Metaserver**

Create components: The three pieces must first be wrapped using the wrapping technology required by the application server. For example, you could wrap the mainframe application and the C++ application using CORBA stubs. The spreadsheet is harder since Excel is a COM component. One could use some sort of COM-CORBA bridge technology to wrap the Excel graphing application. Application servers often force developers to use unnatural technologies to create their components.

**With Metaserver**

Create components: The three pieces must still be wrapped to componentize them. However, Metaserver does not place restrictions on the wrapping technology. Each piece may use whatever is most natural for it. Metaserver can componentize the mainframe application either through standards-based or custom screen scraping technology such as OpenVista and Java, it can componentize the custom C++ application using the Metaserver C++ API, and can use COM to talk to the Excel...
Create the Internet application: You must now write a real program in a real programming language and generate an Internet application that runs under the control of a web application server. The program might look something like this:

Create the Internet application: In the Metaserver environment, the Internet application is created by dragging the GetPortfolio, PricePortfolio, and GraphPortfolio components into a visual diagramming tool and connecting their inputs and outputs appropriately to draw a flowchart. For example, the output of GetPortfolio may be a table of the portfolio contents. This may be connected to the input of the PricePortfolio component. The resulting Internet application simply defines the route through the available components that must be taken in order to satisfy the user request. The route is called a MetaTask.

application. The Unix application continues to run entirely on Unix, the Excel spreadsheet resides on Windows, etc. Metaserver’s wizards, templates and simple APIs enable the creation of these components to be extremely straightforward.
import org.omg.CORBA.*;

class CreateReport {
    // Initialize ORB
    ORB orb = ORB.init();

    try {
        // Handle to GetPortfolio
        GetPortfolio gp = orb.bindTo("GetPortfolio");
        GetPortfolioException e1;
        getPortfolio (uid, pwd);
        } catch (BindException e1) {
            // Handle error getting handle.
        } catch (GetPortfolioException e2) {
            // Handle error in GetPortfolio
        }

        // Convert types
        ConvertGToP (p1);

        try {
            // Handle to PricePortfolio
            PricePortfolio pp = orb.bindTo("PricePortfolio");
            PricePortfolioException e1;
            PricePortfolio p3 = pp.PricePortfolio (p2);
            } catch (BindException e1) {
                // Handle error getting handle.
            } catch (PricePortfolioException e2) {
                // Handle error in PricePortfolio
            }

            try {
                // Handle to GraphPortfolio
                GraphPortfolio gph = orb.bindTo("GraphPortfolio");
                GraphPortfolioException e1;
                GraphPortfolio p4 = gph.GraphPortfolio (p3, &img);
                } catch (BindException e1) {
                    // Handle error getting handle.
                } catch (GraphPortfolioException e2) {
                    // Handle error in GraphPortfolio
                }

            }
}
The program must resolve all type conflicts between the methods that were invoked on the various components. In most real cases, there are conflicts and you must convert from one type to the other before invoking the next method. This is because the component models are all strongly typed.

Metaserver takes care of type conversion automatically. The Metaserver component model is not strongly typed. The data will morph according to its context.

There is no error handling support. If the GraphPortfolio component is unavailable, the program will fail. If the component goes down while it is drawing graphs, the program will fail. It must handle all these errors using some sort of exception handling.

Metaserver's web router handles errors automatically. If one of the components is unavailable, Metaserver can start it automatically. If a component goes down while it is processing the request, it is automatically re-routed to another instance. No information is lost.

If there are a thousand simultaneous client requests, there are a thousand instances of this program executing on the web application server. The program must carefully manage resources and ensure that it is thread-safe. Application server products do provide significant functionality in this area.

If there are multiple instances of the GraphPortfolio component, the program must include code to efficiently balance the load between all available instances in order to maximize throughput. Application server products usually provide no support in this area requiring programmers to create complex distributed scheduling code.

Metaserver's web router automatically balances the load across all available instances of any given component. No programming is required. If there are several requests for the services of a particular component, they are simultaneously routed to as many instances as available.

If two of the components can execute in parallel, the program must contain code to run them simultaneously. This requires time and expert programmers.

If two components can execute in parallel, you simply draw it in the flowchart. The web router will automatically route them in parallel. No programming is required.

Configure and Deploy: The web server and web application servers must be configured to run the Internet application. Usually, they must be re-configured for every application. Once deployed, the application can be accessed from any Web browser using the appropriate URL.

Configure and Deploy: When the flowchart is created, simply click the Deploy button. The web server must be configured once. There is no web application server to configure. Once deployed, the web application can be accessed from any Web browser using the appropriate URL.
The comparison clearly demonstrates the advantages of building distributed web applications using Metaserver. The major difference between the two models is the complete absence of a real coordination or routing mechanism in the web application server solution. This difference impacts every aspect of web application development, deployment and execution. The lack of a coordination layer imposes significant additional burden on the developer. It complicates the runtime environment and creates a requirement for web application servers to manage these complications.

In contrast, Metaserver’s web router removes the need for a web application server. The runtime environment is dramatically simplified. The development burden is drastically reduced.
**Metaserver Web Router**

**Introduction**

The remaining sections describe Metaserver’s web router product called Metaserver. It is designed in several layers as shown in the Figure on the next page.

We first describe Metaserver’s component model. Components are created using wrappers called MetaLinks. Metaserver’s powerful data model (the mechanism by which components access and exchange data) is also detailed below. Metaserver frees developers from the requirement of translating between different data formats used by components running on incompatible platforms.

Metaserver’s communications infrastructure is built using one or more shared network memory servers called MetaServe servers, which act as web routers. Metaserver also provides containers for managing the execution of components. Metaserver’s infrastructure is detailed below.

Metaserver’s powerful coordination layer based on MetaTasks is detailed next. MetaTasks define web routes using a visual diagramming tool that frees the developer from the burden of writing a complex distributed program.

Metaserver offers an integrated development environment used for developing MetaLinks and MetaTasks and for deploying the components, routes and templates that make up the web application.

Finally, Metaserver offers a centralized management console that may be used to administer the entire system. The management console is used for remote administration of all parts of the system, for assigning access control etc.
Metaserver web router architecture
Component Model

What is a Metaserver component?
A Metaserver component usually performs a single backend service or process. It encapsulates the business logic that encodes the functionality provided by the backend process. For example, it may validate a user or add registration information to the customer database. However, it does this one thing very reliably.

The actual implementation of the logic that performs the service is completely hidden by a thin wrapper called a MetaLink™ and is never exposed to the outside world. It could be written as a mainframe application or an Excel spreadsheet. As long as its externally visible functionality does not change, it makes no difference to Metaserver. A component also does not need to know anything about other components.

Components are described via their inputs and outputs. For example, a ChargeCreditCard component has two inputs (the card number and the expiration date) and one output (the authorization code). The inputs and outputs may have types (ms/long, ms/Date etc.) but they are not strongly typed.

Design-time behavior
Metaserver offers an integrated development environment (MIDE), that features a visual diagramming tool into which components are dragged and dropped. The MIDE is described Page 26. Components are wired together to create the distributed application called a MetaTask. (MetaTasks are detailed on Page 23.) When a component is dropped into the MIDE, its visual appearance reflects its characteristics. For example, if the component has three inputs and two outputs, its appearance visually reflects that signature. An example is shown to the right. If the mouse is moved over one of the input/output triangle stubs, a floating textbox displays its name and type. If there is a default value, it is also displayed.

The user can connect the output of one component to the input of another component by drawing a line between the two. The connection is visually represented by drawing an arrow from the output to the input. The input and output stubs change visually to indicate that a mapping has been created.

There are several pre-defined components, including decision boxes (IF-THEN-ELSE logic), fork and join boxes (PARALLELISM), while/wend boxes (LOOPING), etc. They may be used to easily

In practice, a component may have several methods. The GUI allows the user to select one method and changes the visual appearance of the component to reflect the chosen method. For example, if the method has three inputs and two outputs, the component might have the visual appearance shown below.
represent complex constructs such as parallel execution of several components.

**Creating a Metaserver component**

Component = MetaLink + Attached Application

A Metaserver component is constructed by writing a small wrapper program called a MetaLink that isolates all knowledge of the MetaServe server software. It runs between the MetaServe and the attached application/program containing the actual business logic. The MetaLink provides the linkage between Metaserver’s programming model environment and external applications that are reached using application-specific protocols. In addition, the MetaLink embeds an interpreter that handles coordination when running as part of any MetaTask. This is detailed below.

The MetaLink may communicate with an existing application running inprocess, as a separate process, or on a separate machine. You can write a new application as well. In all cases, the model is the same. The MetaLink is the piece (embedded lines of code, subroutine, separate process on the same machine, or separate process on a different machine) that links the MetaServe server software with the attached application.

Metaserver provides a variety of mechanisms for implementing MetaLinks. CORBA, COM or JavaBeans components may be used directly via their respective containers. In this case, the attached application is a CORBA, COM or JavaBeans component.

MetaLinks may also be generated using wizards. Database query or stored procedure components can be automatically generated using a query-generation wizard. Wizards can also generate HTML, XML or XSL templates for dynamic presentation.

Metaserver provides templates for creating a variety of components. In addition, language level bindings (APIs) are provided for Java, C++, Visual Basic and several other languages on the Microsoft platform (through COM).

**Run-time behavior**

When a component first starts up (managed by the container and associated infrastructure), the MetaLink notifies the web router that an instance of the component (say, its name is GetPortfolio) is running and waiting for requests. When a request for GetPortfolio arrives, the web router sends the request to the MetaLink, which accepts the request, creates the necessary data structures, and then invokes the logic in the attached application via its natural communication mechanism.

When the attached application has completed execution, the interpreter embedded in the MetaLink interprets the control flow information contained in the MetaTask, and returns the request to the
web router with the name of the next component in the route. The web router will then route the request to an instance of that component.

The attached application does not know that it is part of a MetaTask, or that it is running as part of a distributed Internet application. It simply knows how to query the inventory. The other issues are irrelevant and are automatically handled by the surrounding infrastructure.

Data Model

The same information often takes incompatible formats in different applications. The MetaLink automatically presents data to each attached application in its natural format. In object-oriented environments (CORBA, COM, JavaBeans), the appropriate set of objects is created. For shell scripts or batch files, the appropriate environment variables or command-line arguments are supplied. Thus, the Metaserver enables data exchange between components running in completely incompatible environments (for example, a COM component can transparently exchange information with a CORBA component).

Properties

Metaserver presents data in the form of collections of name-value pairs called properties. Property names are hierarchical. Collections hold other collections as well as properties. Effectively, Metaserver creates a tree where the nodes represent the collections and the leaves represent the properties. For example, the property with full name Request.Form.Customer is the Customer property in the Form collection, which is itself in the Request collection. A property can hold data of any type and can morph to behave according to the context in which it was being used.

Components that are part of a MetaTask are free to create any properties they need. There are no restrictions imposed. However, the data is not directly visible to other components unless a connection is made between the output property and the other component’s relevant input property.

How does a component access data?

Components access data using the MetaLink API, which provides functions to get and set the value of properties. A component simply reads the value of a property by asking for it by name. It writes the value by supplying a name and a value. All properties that a component gets (reads) are its inputs. All properties it sets (writes) are its outputs.

This follows the Linda model of orthogonal communication and computation. Components have no knowledge of the entity that creates or owns a property. They don’t have to invoke a method or send a message to that entity in order to exchange data with it. This means a

Metaserver’s Internal Scripting Language

Metaserver’s internal scripting language (the MetaTask flowchart is converted internally to a script) uses a typeless approach to achieve significant component reuse, transparent data exchange and very rapid application development. For example, InterestRate might be an integer to one component, a String to another and a floating-point value to a third. In the Metaserver environment, the InterestRate component would morph to behave according to the context in which it was being used. This is in stark contrast to the strongly typed nature of the CORBA, COM or JavaBeans component models. These components inhibit reuse because they often have a variety of incompatible interfaces, each of which requires objects of specific types. The compiler prevents other types of objects from being used with the interface. The programmer must usually write real code (at a system programming language level) to translate between the type of the object and the type expected by the interface. This requires recompilation (which may not be possible), slow application development and an assortment of other problems.
component does not have to know anything about that entity, it does not have to get a handle to the object or know the hostname/IP address of the machine on which it is running, etc. The component reads data from "out there" and writes data "out there" and some other entity that is interested in accessing the data gets it by reading it from the same "out there." The "out there" is, of course, the web router's shared network memory or MetaServe.

**Data Persistence and Naming**

The properties and collections created by components persist only for the lifetime of the MetaTask. When the MetaTask completes, the data is destroyed. The Metaserver API also provides the ability to create data that persists beyond the lifetime of the MetaTask in which it was created.

Every component implicitly executes in its own namespace. It sees some standard collections and any data that it creates. It cannot read or write data outside this namespace. This ensures that property names used in two separate components do not collide. A component can create any data (including collections of properties) in its own namespace.

The Metaserver provides mechanisms to indirectly access data in the namespaces of other components. The visual diagramming tool, which is used to create the MetaTask, permits an output from one component to be connected to the input of another component. The output property is then mapped into the namespace of the second component.

**Infrastructure**

Components communicate with each other through the web router's shared network memory (tuple space) server called the MetaServe. This model is radically different from remote method invocation. Components are not aware of the existence of other components. They simply write information out into the MetaServe for other components to read when they are ready for it. The communication model fully supports asynchronous and synchronous communication.

Information is written to the MetaServe in the form of tuples. A tuple is simply an ordered collection of fields. It contains a certain number of [non-unique] elements in specific locations (fields). For example, the tuple

```
("foo", 12, 22.223, false)
```

contains the String foo as its first element, the integer 12 as its second, the double 22.223 as its third and the Boolean value false as its fourth field.

The MetaServe provides the ability for components to write tuples to its memory and read/take tuples from the memory. It can also hold tuples persistently in memory for long periods. It matches requests with available tuples. If a match is found, the contents of the matched tuple are returned to the requestor. If not, the requestor either blocks until one is available or may continue doing something else and issue a new request later. For example, a request might look like

```
```
This request matches the tuple above because the first field is the exact String foo, and the remaining fields are variables of types integer, double and Boolean respectively. The MetaServe matches the request to a tuple in its memory and returns the contents of that tuple. When the request is completed, the variables x, y and z have values 12, 22.223 and false respectively.

Metaserver uses the above tuple matching mechanism to implement a sophisticated coordination layer for MetaTasks. It is the basic communication mechanism between components.

In addition, Metaserver provides a container in the context of which its components execute. This container provides services similar to MTS or an EJB server. It manages remote invocation of components, threading, fault tolerance, etc. Metaserver offers a centralized management console using which the container may be configured remotely. The management console is described on Page 26. Components may be started or stopped using the management console or in an automated manner using a built-in resource manager. They run in their own threads (or process, if the component is written as an independent executable) without being aware of it. An authentication facility is built into Metaserver using access control lists.

**Failover subsystem**

Metaserver has a sophisticated failover system that ensures no single point of failure in the distributed system.

If any individual component or MetaLink fails, its container notices the failure and may attempt to restart it. If the component was in the middle of processing a request, the request tuple is recreated in the MetaServe in its original state and no information is lost. If another instance of the component is available, the web router will automatically route the request to the new instance.

Metaserver also has backup MetaServe servers. If any individual MetaServe server fails, all activity automatically rolls over to the backup server. All state is replicated on both servers. No executing MetaTask, component or client is affected.

**Metaserver Task Model**

**What is a MetaTask?**

A MetaTask describes the distributed application that must be executed to process the request created by the client. It contains a route that describes the flow of execution between the various processes that make up the distributed application.

For example, consider the online portfolio problem discussed earlier. As you will recall, the securities in the portfolio are stored in a database that runs on a mainframe. There is a custom application written in C++ that runs on a Unix platform and prices the securities. Finally, a Microsoft Excel spreadsheet graphs the priced portfolio and generates some tables. A MetaTask would describe these processes (GetPortfolio, PricePortfolio, and GraphPortfolio) and the
sequence in which they should be executed (the route). The MetaTask has a name, say portfolioReport.

When clients attempt to get portfolio reports, each request is mapped to the portfolioReport MetaTask. The MetaTask would “execute” (described below), the report would be generated and the client would instantly get the report.

**How is a MetaTask created?**

A MetaTask is created by dragging the required components into a visual diagramming tool and connecting their inputs and outputs together to draw the route. The route describes the order in which the components need to be executed to perform the task. Once the route is created, you simply give it a name (portfolioReport) and click the Deploy button. A new distributed Internet application has just been created. The MetaTask is stored in persistent storage and as a tuple in the MetaServe.

It can be executed simply by accessing the appropriate URL on the Web server from any standard Web browser. The syntax of the URL depends on the Web interface that was used. For example, if the servlet interface was used, the URL might be

```
http://<webserver>/servlet/MetaServlet/myWebApp?_Cmd=portfolioReport
```

**How is a MetaTask executed?**

When the Web server receives this request, the servlet recognizes it as a request for the portfolioReport MetaTask. It generates a unique “jobid” for this task and issues a request for its execution by writing a tuple to the MetaServe. This tuple is called the “request tuple” or the “task tuple.” Each task has one (or more) of these during its execution. The task tuple looks something like this:

```
("V2.0", "Converter", "jobid", data, MetaTask,...)
```

The servlet then blocks waiting for a response. It blocks by attempting to read a tuple from the MetaServe that matches:

```
<"V2.0", "jobid", "jobid", ?data,...>
```

While it is waiting, the servlet consumes no resources on the Web server.

The task tuple contains a version number in its first field, the String “Converter” in its second field, the unique jobid in its third field, data (described later) and the MetaTask itself in its fifth field.

The web router routes the tuple to an internal Metaserver component called the Converter. The Converter is waiting to process a request by attempting to match a tuple containing the String “Converter” in the second field. The MetaServe supplies a tuple if one is available. Otherwise, the Converter blocks until a matching tuple is created by some other entity. Thus, the routing happens because the second field of the tuple contains the String “Converter” in it and there is an instance of the Converter component waiting for a matching tuple.
When the Converter receives the tuple, it examines the data field and finds the name of the MetaTask that the client has requested. It then reads a tuple containing the MetaTask from the MetaServe, returning an error if the requested tuple could not be read. (This MetaTask tuple was created earlier when the user clicked the Deploy button in the MIDE.) When the MetaTask (say portfolioRequest) is loaded, the Converter examines it and finds out that the first component in the route is GetPortfolio. It creates a tuple in the MetaServe that looks like

\[("V2.0", "GetPortfolio", "jobid", data, MetaTask, \ldots)\]

At this point, the Converter can go back to process additional client requests to run MetaTasks. It is not concerned with the remaining execution of this MetaTask.

If the GetPortfolio component has already been started, its MetaLink has already asked the router for tuples that match the above tuple. If the GetPortfolio component is not running, the system will automatically start it.

The web router will route the tuple to the GetPortfolio component. It examines the data field, finds its inputs (userid and password in this case), queries the database and writes its outputs (portfolio table) to the data field. The MetaLink then examines the route information in the MetaTask and finds out that the next component in the route is PricePortfolio. It creates a tuple in the MetaServe that looks like

\[("V2.0", "PricePortfolio", "jobid", data, MetaTask, \ldots)\]

GetPortfolio then goes back to the MetaServe and requests another tuple with “GetPortfolio” in its second field. It is not concerned with the remaining execution of this MetaTask.

In this manner, the task tuple is routed to the PricePortfolio component and from there to the GraphPortfolio component until the MetaTask has completed.

The final step in every MetaTask is always Responder, which is an internal Metaserver component. It returns the response (the data field) to the servlet by writing a tuple to the MetaServe that matches the servlet’s request.

\[("V2.0", "jobid", "jobid", data, \ldots)\]

This time, the tuple has the unique jobid in the second field. The tuple is routed it to the servlet, which can read the response from the data field.

This sophisticated underlying infrastructure is completely transparent to end-users and developers. Developers simply create components and wire them together to create MetaTasks. End-users issue requests for the execution of MetaTasks by clicking a URL. Everything else happens under the covers. Note that there is no centralized process running on a web application server that controls the execution of the MetaTask and invokes the appropriate methods on the components. This makes Metaserver a unique and powerful web router.

There is no coding to worry about when creating the MetaTask. There are no type conflicts. Metaserver’s typeless scripting approach handles the details for you. There is no error handling to
deal with. The web router automatically handles this for you. If two components can run in parallel, you simply draw that into the route. They will automatically run in parallel. If one of the components goes down in mid-processing, the web router will automatically route the request to another instance that may be running on a different machine with a different OS and a different programming language implementation. This failover is totally transparent to the developer and the Web server. Further, the Web server is doing nothing on behalf of Metaserver. If there are a thousand simultaneous requests, the Web server simply generates a thousand tuples in the MetaServe. The surrounding components deal with the rest. If there are several instances of the GraphPortfolio component, they will all run simultaneously and increase the overall throughput. The web router will automatically route requests to as many instances of the component as are available. Consequently, the Metaserver architecture is highly scalable and high performance.

Metaserver Integrated Development Environment (MIDE)

The Metaserver comes with a powerful IDE that is used for developing MetaLinks and MetaTasks. The MIDE is also used to store MetaLinks and MetaTasks in the Metaserver's central repository. The MIDE can also deploy components to remote computers on the network.

The MIDE provides a number of wizards for creating MetaLinks. Wizards can generate complete components for several important functions (SQL Query, HTML/XML processing etc.) Wizards can also create a MetaLink template in one of the supported programming languages. The template needs minimal modification to create the complete MetaLink.

As described earlier, the MIDE has a visual diagramming tool that enables components to be linked together to define routes called MetaTasks.

The built-in persistence subsystem stores components and tasks in a central repository. When a MetaLink or MetaTask is created, it may be persisted and deployed simply by clicking the Deploy button in the MIDE. This copies the necessary information to the central repository. It is now available to all machines connected to Metaserver. MetaTask and MetaLink definitions may be retrieved from the repository as required.

Component implementations must ultimately be deployed on one or more machines running a container. This happens dynamically at runtime. When a component needs to be started up on a remote machine, all the necessary information (binary files, configuration information etc.) is automatically retrieved from the repository.

Metaserver Management Console

Metaserver has a centralized management console for administering the entire Internet application system. All backend machines may be remotely configured to run components. Components may also be started or stopped remotely.

The management console can also be used to assign access control rules for tasks. Access control is uid/gid based. Every user must have a unique uid. In addition, there may be an arbitrary number of groups. Each group has a unique gid and may contain any number of uids.

©Metaserver, Inc.
Authentication
Access control is built around users first authenticating themselves to a built-in Authenticator component. This is accomplished by running the __ms_authenticate MetaTask, which comes prepackaged with the Metaserver. This task consists of Converter, Authenticator and Responder.

The Metaserver includes an LDAP-based Authenticator. It performs user authentication by attempting to bind to an LDAP server using a supplied username and password pair.

The default Authenticator may be replaced with a custom Authenticator that authenticates using a completely different mechanism. However, it must conform to the signature of the built-in Authenticator. This enables users to deploy custom authentication schemes but allows the rest of the Metaserver access control mechanism to remain unchanged.

MetaTask ACLs
MetaTasks may have access control rules associated with them. These access control rules are created using the Metaserver Management Console. A MetaTask can have a list of uids and gids that may execute it, and a list of uids and gids that may modify it. This ACL information is stored in the central repository and is presented as part of the MetaTask itself.

When the Converter receives a request to run a particular MetaTask, it checks the ACL information in the task tuple. If there is no ACL, then the Converter uses a predefined system policy that either always allows or always denies access to MetaTasks with no ACL. If an ACL object is found, the Converter looks in the Client collection for authenticated uid and gids. It determines whether the user making the request has permission to run the MetaTask. If a match is found, the MetaTask is allowed to run. Otherwise, it is not allowed to run and an error is returned immediately to the client.
Summary

The drive toward Internet-enabled business presents a new set of challenges to most organizations today. Today’s Internet applications must often be built out of existing legacy applications that run in incompatible environments spanning multiple business units. These existing applications were never designed to run in the Internet environment. Further, the Internet applications must use the relatively unreliable Internet as a communications medium. In spite of this, they are business critical and need to be robust and reliable.

This represents a huge shift in focus from building client-server and even component-based applications that use a single standard for all their components and run in a relatively controlled environment (number of users, reliable network etc). These new business application architectures place far greater emphasis on the Internet infrastructure than on the components (applications) themselves.

Metaserver’s web router product supplies a key piece of this Internet infrastructure. It enables organizations to use the technologies and applications they currently have in place to rapidly build the new Internet applications they need. It does not force all business units that participate in the Internet application to choose a single standard, which can be very expensive and is usually impractical.

The web router totally automates features such as load balancing, scalability, and fault tolerance of the overall business critical Internet application. This is vital because the components themselves are not necessarily robust or high-performance (they are often constructed out of pre-existing applications that were never designed to run in this environment), the Internet itself is relatively unreliable and low quality, and the number of users of the system is unpredictable. This, by itself, represents a tremendous saving in time and money for any organization.

The web router supports a wide range of industry standards (EJB, LDAP, HTML…) and is capable of leveraging products that implement those standards (application servers, directory servers, web servers…).

Finally, the web router allows organizations to use the existing skills of their people most effectively. This is critical in today’s environment with its dramatic shortage of skilled resources.

Metaserver can bring your business to the Internet faster than you ever thought possible.