US 20030135509A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2003/0135509 A1

Davis et al. (43) Pub. Date: Jul. 17, 2003

(54) EDGE SERVER JAVA APPLICATION FRAMEWORK HAVING APPLICATION SERVER INSTANCE RESOURCE MONITORING AND MANAGEMENT

(76) Inventors: **Andrew Thomas Davis**, San Francisco, CA (US); **Jay Parikh**, Redwood City, CA (US); **Srinivasan Pichai**, Foster City, CA (US); **Eddie Ruvinsky**, Redwood City, CA (US); **Daniel Stodolsky**, Somerville, MA (US); **Mark Tsimelzon**, Sunnyvale, CA (US); **William E. Weihl**, San Francisco, CA (US)

Correspondence Address:
AKAMAI TECHNOLOGIES, INC.
ATTN: DAVID H. JUDSON
8 CAMBRIDGE CENTER
CAMBRIDGE, MA 02142 (US)

(21) Appl. No.: **10/340,109**

(22) Filed: **Jan. 10, 2003**

Related U.S. Application Data

(60) Provisional application No. 60/347,481, filed on Jan. 11, 2002.

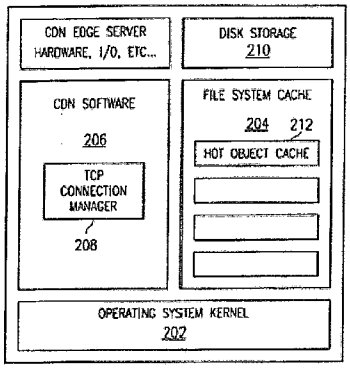
Publication Classification

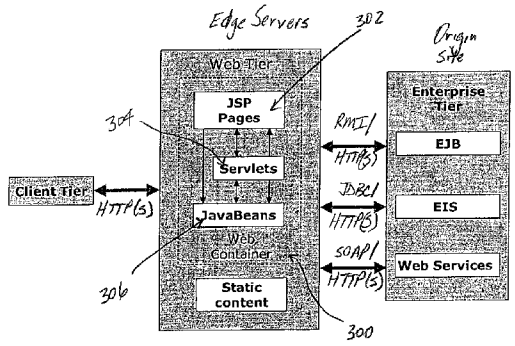
(51) **Int. Cl.**[7] **G06F 7/00**

(52) **U.S. Cl.** **707/100**

(57) **ABSTRACT**

An application deployment model for enterprise applications enables such applications to be deployed to and executed from a globally distributed computing platform, such as an edge server in an Internet content delivery network (CDN). In a representative embodiment, a CDN edge server supports application server code that executes a Web tier and/or Enterprise tier component of a given Java-based application. When multiple instances of the application server code are executed, given resources (e.g., memory, CPU, disk and network I/O) are monitored, and the application server instances are terminated or rate-limited to prevent over-utilization by any particular instance. In addition, a given application running in a given application server instance is restricted from taking certain actions, e.g., reading or writing from a file system, so that it cannot interfere with or access data from another customer's application.





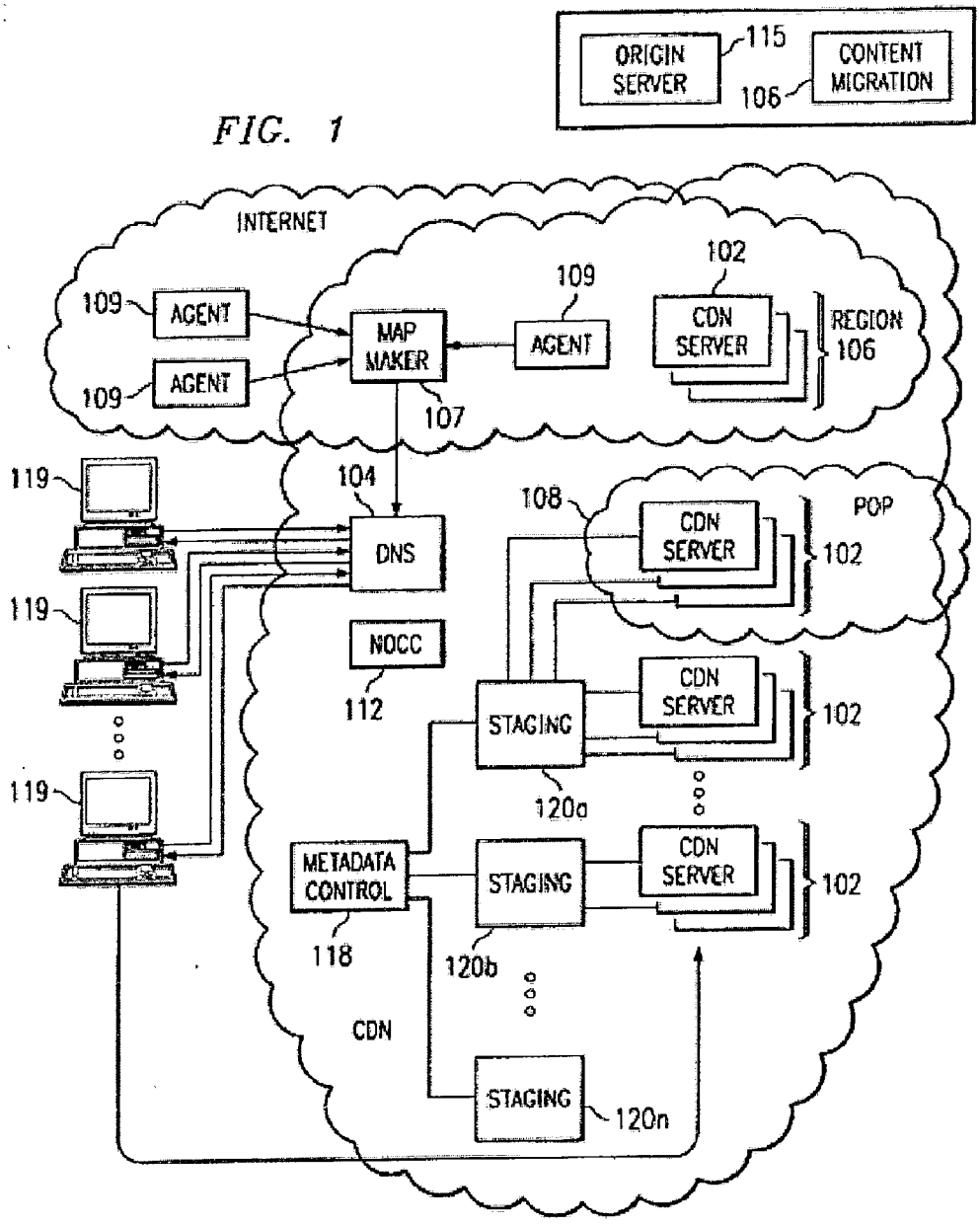

ORIGIN SERVER
115
CONTENT MIGRATION
106
INTERNET
109
AGENT
109
AGENT
MAP MAKER
107
109
AGENT
102
CDN SERVER
REGION 106
104
DNS
119
119
119
NOCC
112
108
CDN SERVER
POP
102
STAGING
120a
CDN SERVER
102
METADATA CONTROL
118
STAGING
120b
CDN SERVER
102
CDN
STAGING
120n

FIG. 1

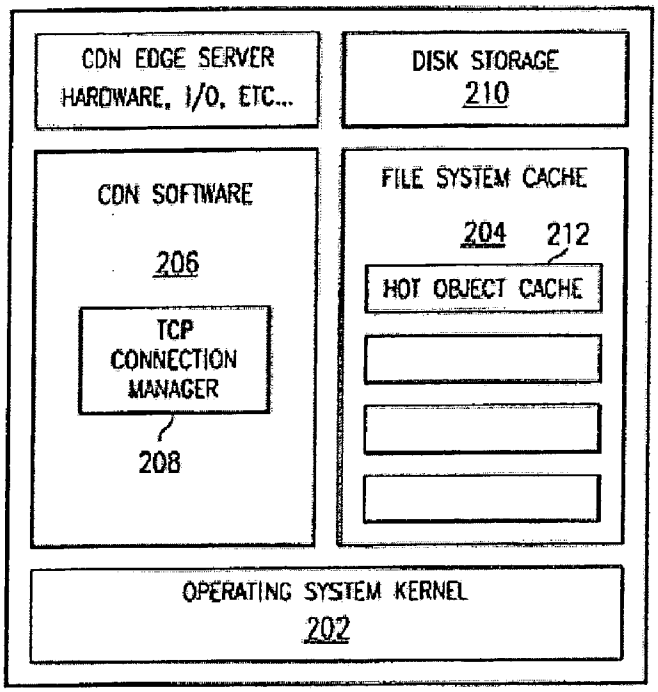


FIG. 2

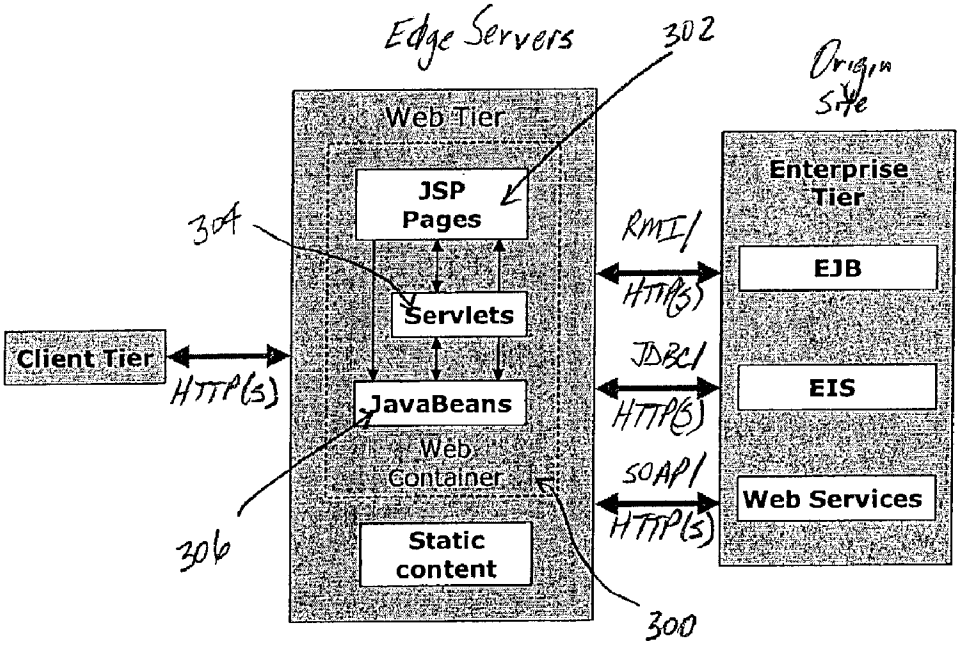


Figure 3

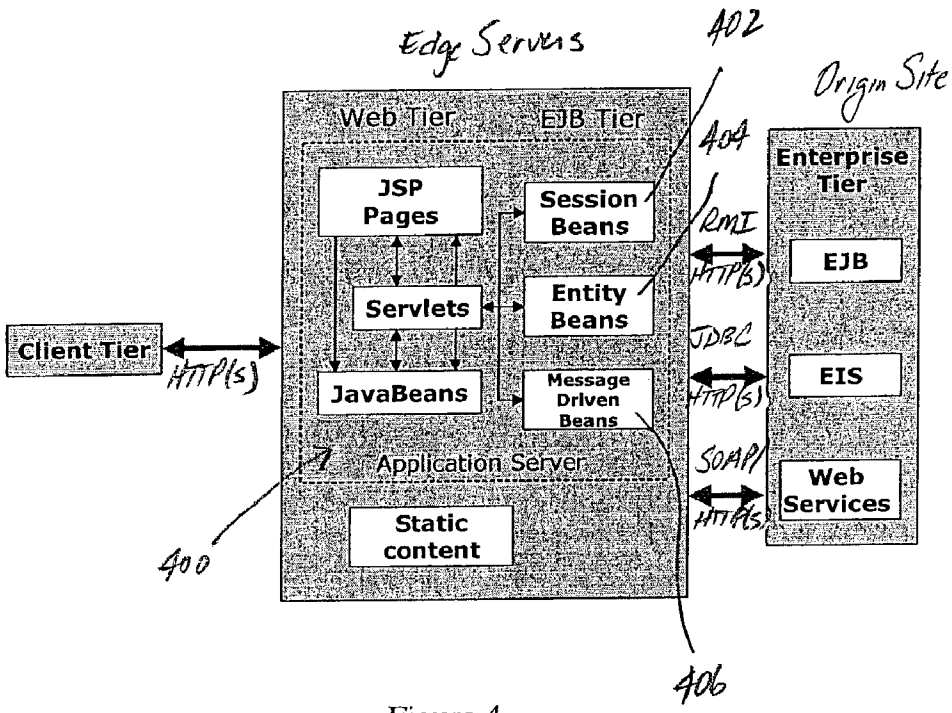


Figure 4

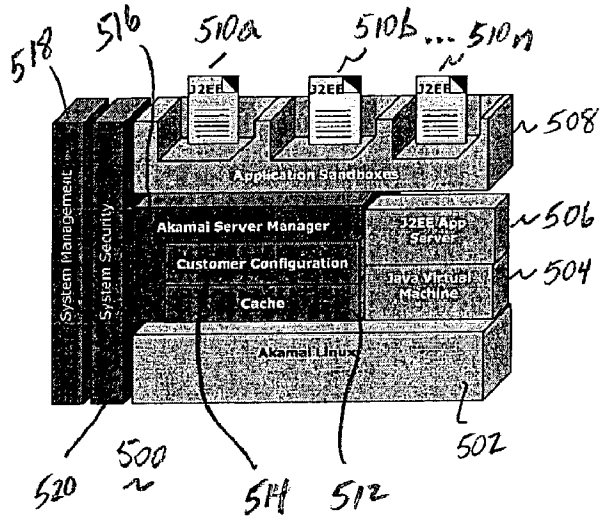


Figure 5

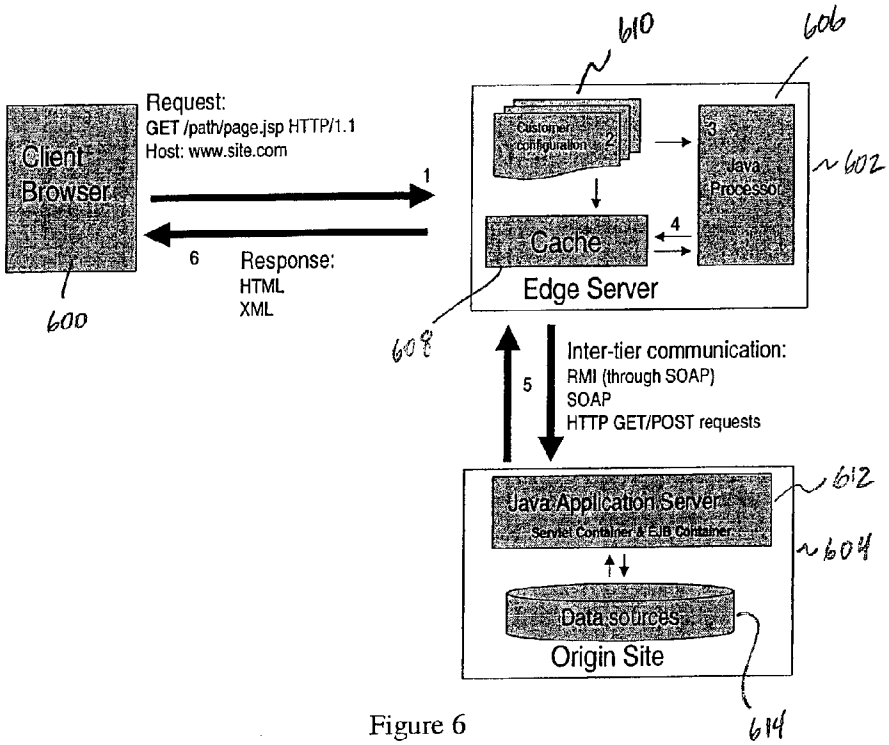


Figure 6

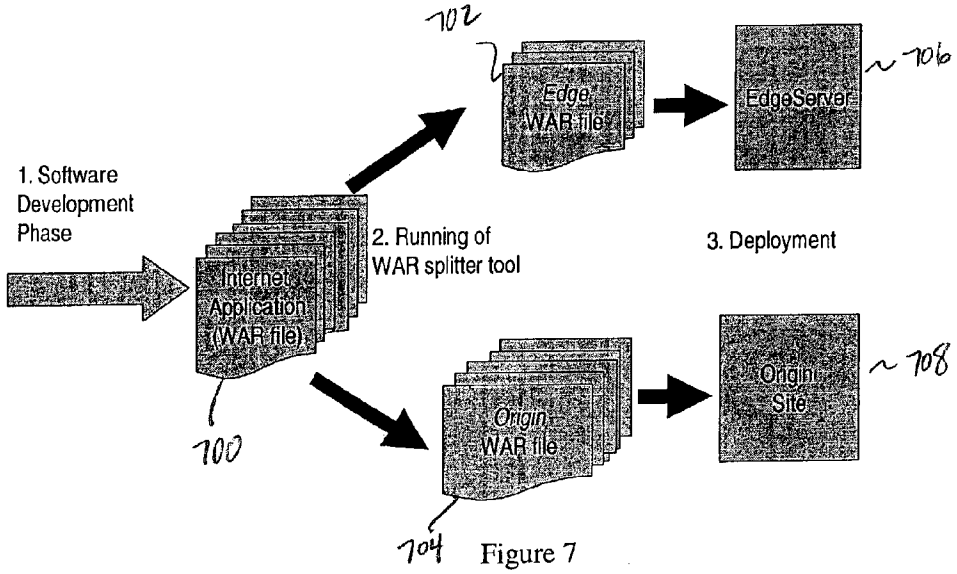


Figure 7

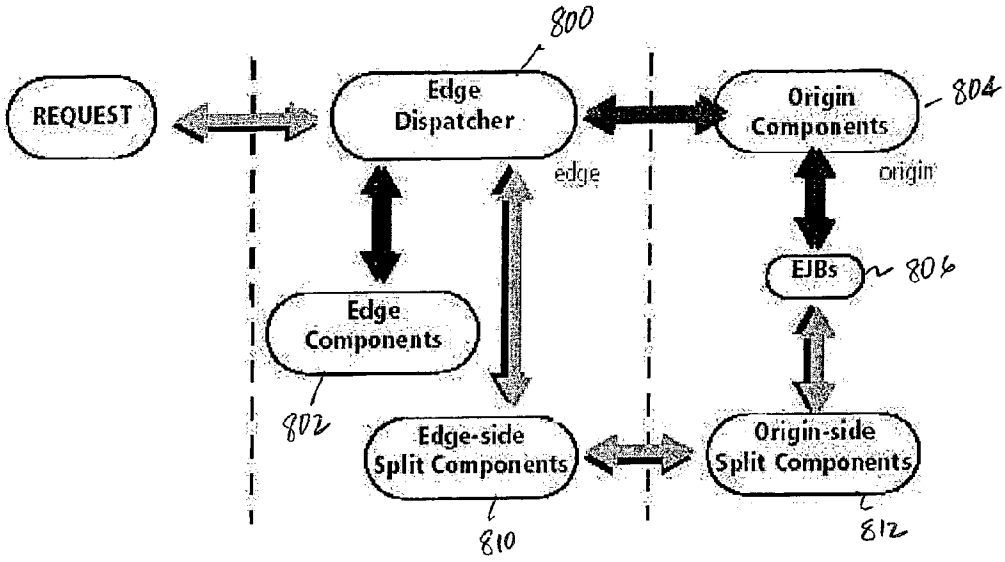


Figure 8

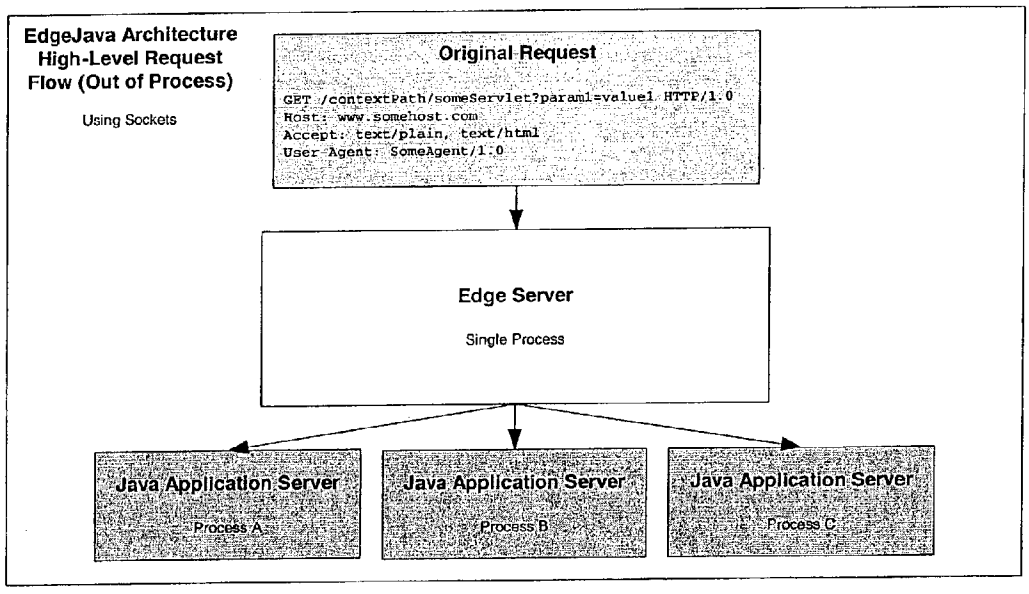


Figure 9

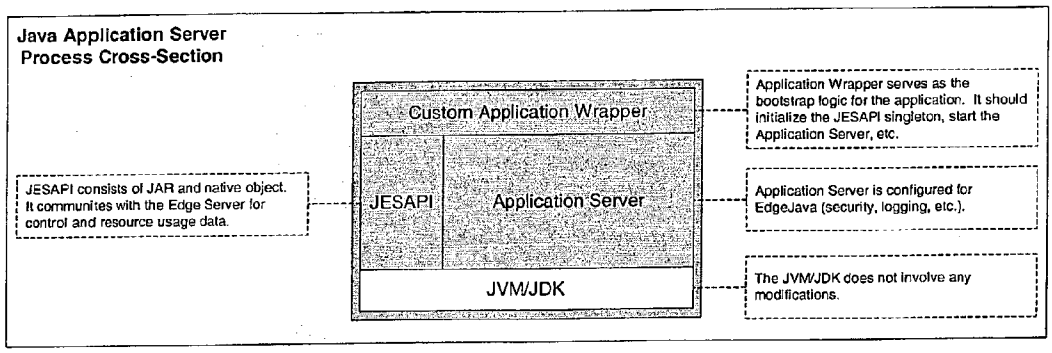


Figure 10

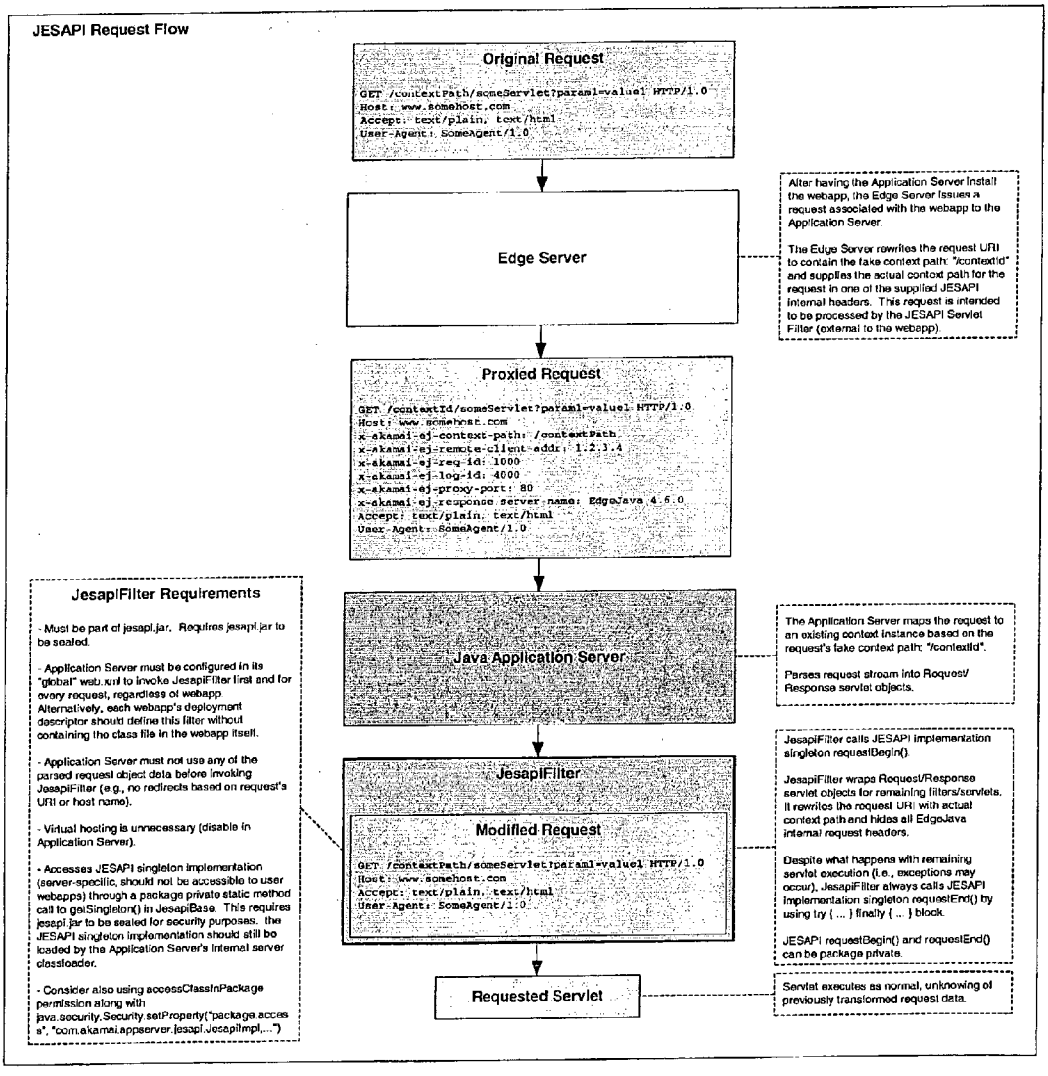


Figure 12

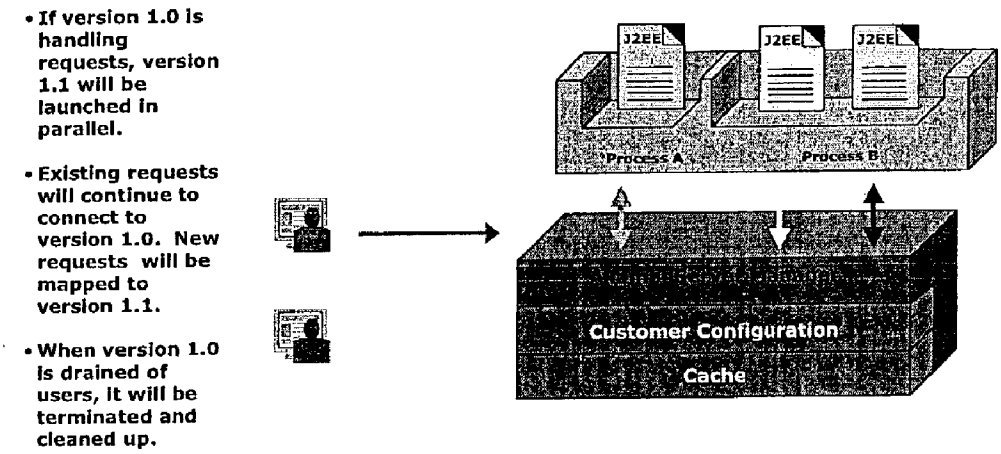


Figure 11

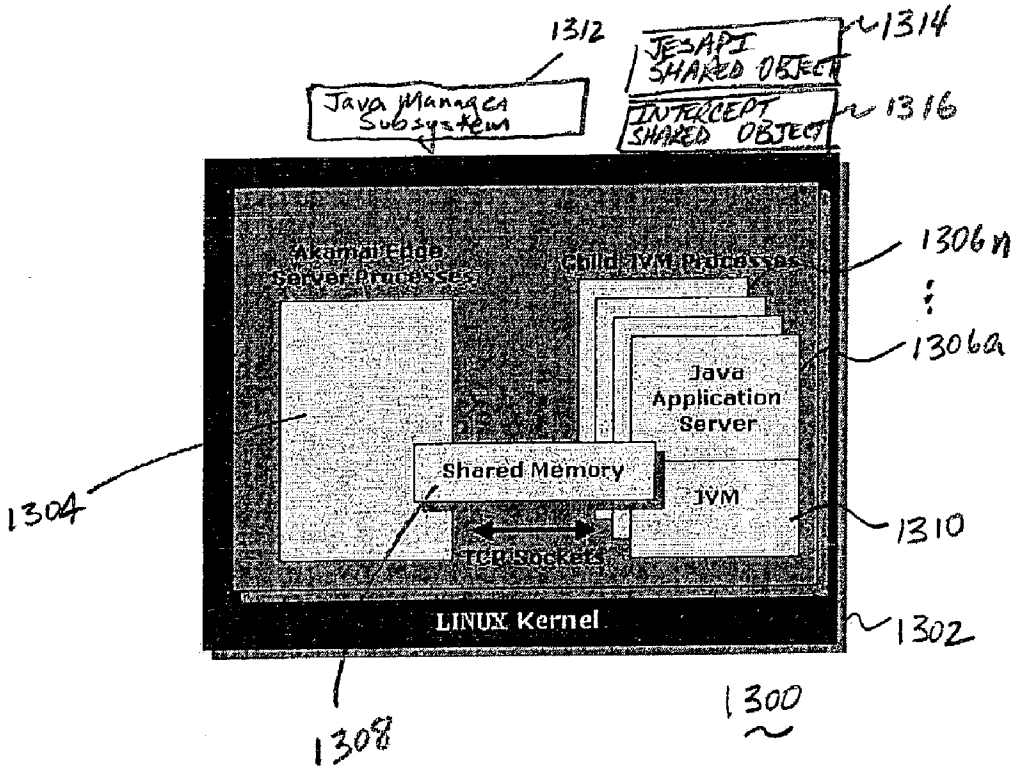


Figure 13

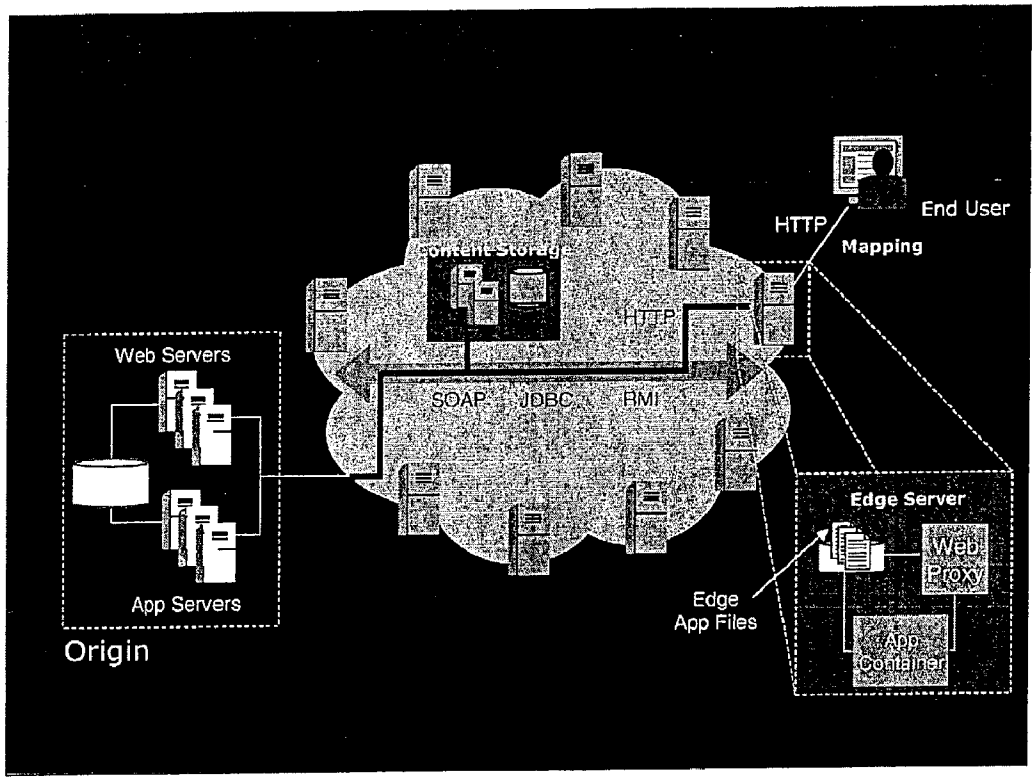


Figure 14

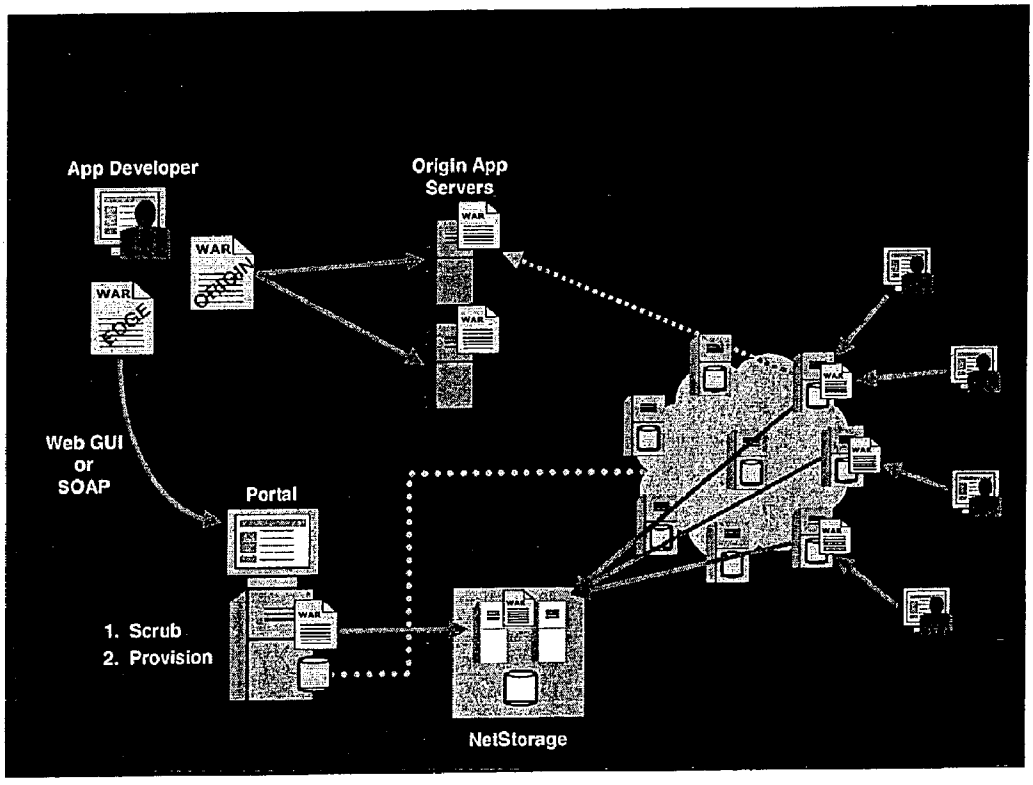


Figure 15

EDGE SERVER JAVA APPLICATION FRAMEWORK HAVING APPLICATION SERVER INSTANCE RESOURCE MONITORING AND MANAGEMENT

[0001] This application is based on and claims priority from Provisional Application Serial No. 60/347,481, filed Jan. 11, 2002.

BACKGROUND OF THE INVENTION

[0002] 1. Technical Field

[0003] The present invention relates generally to an application deployment model for use in a content delivery network.

[0004] 2. Description of the Related Art

[0005] Enterprises can expand their business, increase efficiency, and enable new revenue streams by extending their business applications over the Internet to customers, partners, and suppliers. One way to enable enterprises to shift the operational burden of running a reliable and secure Web presence is to outsource that presence, in whole or in part, to a service provider, such as a content delivery network (CDN). A content delivery network is a collection of content servers and associated control mechanisms that offload work from Web site origin servers by delivering content (e.g., Web objects, streaming media, HTML and executable code) on their behalf to end users. Typically, the content servers are located at the "edge" of the Internet. A well-managed CDN achieves this goal by serving some or all of the contents of a site's Web pages, thereby reducing the customer's infrastructure costs while enhancing an end user's browsing experience from the site. In operation, the CDN uses a request routing mechanism to locate a CDN edge server electronically close to the client to serve a request directed to the CDN. Sites that use a CDN benefit from the scalability, superior performance, and availability of the CDN service provider's outsourced infrastructure.

[0006] Many enterprises, such as those that outsource their content delivery requirements, also implement their business services as multi-tier (n-tier) applications. In a representative n-tiered application, Web-based technologies are used as an outer (a first or "presentation") tier to interface users to the application, and one or more other tiers comprise middleware that provides the core business logic and/or that integrates the application with existing enterprise information systems. The Java 2 Platform, Enterprise Edition (J2EE™) is a technology and an associated component-based model that reduces the cost and complexity of developing such multi-tier, enterprise services. The J2EE runtime environment defines several types of application components that can be used to build services. These include (a) Web tier components (e.g., servlets, JSP pages, Java beans, filters, and web event listeners), which are components that typically execute in a web server and respond to HTTP requests from web clients, and (b) Enterprise tier components (e.g., session beans, entity beans and message driven beans, which may be developed as Enterprise JavaBeans™ (EJB™)), that include the business logic and that execute in a managed environment to support transactions. Runtime support for J2EE application components are provided by so-called "containers," with a Web container supporting the Web tier components, and an Enterprise container supporting the Enterprise tier components. Containers execute the application components and provide utility services. J2EE-compliant servers provide deployment, management and execution support for conforming application components.

[0007] It would be desirable to be able to provide a framework by which such server-side Java applications as well as other Web services could be deployed in a distributed computing environment, such as a content delivery network, to enable application processing on the edge of the Internet.

SUMMARY OF THE INVENTION

[0008] It is an object of the present invention to provide an application deployment model for enterprise applications to enable such applications to be deployed to and executed from a globally distributed computing platform, such as an Internet content delivery network (CDN).

[0009] It is a more specific object of the invention to provide a framework by which Java-based applications and Web services are deployed onto a distributed computing platform so that enterprises can take advantage of a multi-tier distributed application model.

[0010] Another object of the present invention is to provide a deployment model for a content delivery network that enables support for a Java-based Web container or Enterprise container, or both, so that applications or application components can be executed on the edge of the Internet.

[0011] A more general object of this invention is to provide a content delivery network with the ability to execute application code on an edge server. Using the present invention, content is created on the edge of the network by running application code.

[0012] A specific object of the invention is to provide an edge application deployment model that supports execution of Web tier components, e.g., Java server pages (JSP), servlets and Java beans, on the edge of the Internet close to end users, thus avoiding network latency and the need for costly infrastructure over-provisioning, while improving the performance and reliability of mission-critical enterprise applications.

[0013] In a preferred embodiment, the present invention is a CDN Java application framework offering comprising Java-enabled edge servers. This framework takes advantages and leverages the mapping, load-balancing and management systems that are similar to the ones used with known CDN offerings. In a first aspect, the present invention enables the offloading and execution of the presentation or Web tier of n-tier Internet applications. JSP, Servlets, Java beans and custom tags, which are executed within an application server's servlet container, are executed at the edge of the Internet, close to the end-user. In an alternate embodiment, in addition to the Web tier, at least some or all of the Enterprise tier of the application is also deployed to and executed on a given edge server. The Enterprise tier typically comprises middleware such as entity beans, session beans, and message-driven beans that implement the application's business logic and that provide local or remote database support.

[0014] According to another aspect of the present invention, developers preferably separate their Web application

into two layers: a highly distributed edge layer and a centralized origin layer. In a representative embodiment, the edge layer supports a Web container so that the following technologies are supported: Java server pages (JSPs), servlets, Java beans, Java helper classes, and tag libraries. Preferably, communications between the edge and the origin use conventional communication protocols such as RMI and SOAP. Any protocol that can be tunneled over HTTP, such as JDBC, can also be supported.

[0015] Preferably, an application is run on the edge server in its own application server instance in its own Java virtual machine (JVM). In a preferred embodiment, a content delivery network service provider operates a CDN with at least one edge server that includes multiple application server/JVM instances, with each instance associated with a given CDN customer. Resource utilization by the multiple application server instances are monitored, and application server processes that over-utilize given resources (e.g., memory, CPU, disk, and network I/O) are terminated. In addition to resource management, preferably security restrictions are imposed on applications running in each application server/JVM process. This is sometimes referred to as sandboxing. These restrictions include, for example, file system read/write restrictions, limitations on socket opening and usage, restrictions on thread starting, stopping and modification, as well as code restrictions that prevent applications from reading certain application server classes. Preferably, a given application cannot run or load code belonging to other applications, it cannot load data belonging to another application, it cannot read or write arbitrary files on the file system, and it cannot make native kernel calls or load libraries that make native calls.

[0016] By providing Web containers at the edge, the present invention provides the ability to off-load up to the entire Web tier of n-tier Internet applications. Web components executed within the application server's servlet container, can be executed at the edge of the Internet, close to the end-user.

[0017] In an illustrative operation, an end user makes a request that is directed to a CDN edge server. If the request calls for Java processing and is the first request for the particular application, the application is retrieved from the origin, unpacked, and loaded into the application server. If the application component (e.g., a Web application archive or "WAR" file) is already cached on the edge server, the appropriate servlet or JSP page is used to generate the response. As needed, the edge server contacts the origin site with those portions of the application that need to run on the origin, e.g., access to a central data resource or other non-edgeable servlet. The parts of the page that can best be served from the edge are processed at the edge, while those parts that need to be processed at the origin are processed at the origin, and the results are served back to the end user from the edge server.

[0018] Application components are delivered to the edge servers on an as-needed basis. In an alternate embodiment, it is desirable to pre-deploy an application or an application component based on some prediction of expected future need for that application or component, or for purposes of fault tolerance. Thus, a given application or component thereof may be delivered to a particular edge server and initialized and started irrespective of whether an end user request has been received at the server.

[0019] The foregoing has outlined some of the more pertinent features of the present invention. These features should be construed to be merely illustrative. Many other beneficial results can be attained by applying the disclosed invention in a different manner or by modifying the invention as will be described.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] For a more complete understanding of the present invention and the advantages thereof, reference should be made to the following Detailed Description taken in connection with the accompanying drawings, in which:

[0021] **FIG. 1** is a block diagram of a known content delivery network in which the present invention may be implemented;

[0022] **FIG. 2** illustrates a typical machine configuration for a CDN edge server;

[0023] **FIG. 3** illustrates a first embodiment of the present invention wherein a Web tier is implemented in an edge server;

[0024] **FIG. 4** illustrates a second embodiment of the present invention wherein a Web tier and an Enterprise tier are implemented in the edge server;

[0025] **FIG. 5** illustrates a representative edge server of the present invention for use in executing one or more edge-enabled applications;

[0026] **FIG. 6** illustrates a common request/response data flow for an edge-enabled application according to the present invention;

[0027] **FIG. 7** illustrates one technique for developing an edge application for use in the present invention;

[0028] **FIG. 8** is an illustrative communication data flow when an edge server dispatcher component receives a client request;

[0029] **FIG. 9** illustrates an illustrative high level out of process request process flow according to the present invention;

[0030] **FIG. 10** illustrates an illustrative Java application server process according to an embodiment of the invention;

[0031] **FIG. 11** illustrates how to upgrade an application version in the application server without interrupting the processing of client requests according to a feature of the present invention;

[0032] **FIG. 12** illustrates a representative request processing flow for the illustrative embodiment of **FIG. 9**;

[0033] **FIG. 13** illustrates a typical edge server concurrently executing multiple Java application server instances for a plurality of CDN customers;

[0034] **FIG. 14** illustrates a CDN in which an edge server provisioned with an application server container communicates with the origin server and vice versa through one or more communications protocols; and

[0035] **FIG. 15** illustrates a representative application provisioning method and system that takes advantage of the CDN service provider's secure customer portal.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0036] The present invention is a Java application framework that leverages Internet CDN architecture and functionality such as generally described below. Familarity with Java programming conventions and the J2EE architecture are presumed. Additional information about J2EE is available in the publication titled *Java 2 Platform Enterprise Edition Specification v1.3* (July 2001), which is available from Sun Microsystems. An online copy is available at the following URL: http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf.

[0037] By way of background, it is known in the prior art to deliver digital content (e.g., HTTP content, streaming media and applications) using an Internet content delivery network (CDN). A CDN is a network of geographically-distributed content delivery nodes that are arranged for efficient delivery of content on behalf of third party content providers. Typically, a CDN is implemented as a combination of a content delivery infrastructure, a request-routing mechanism, and a distribution infrastructure. The content delivery infrastructure usually comprises a set of "surrogate" origin servers that are located at strategic locations (e.g., Internet network access points, Internet Points of Presence, and the like) for delivering content to requesting end users. The request-routing mechanism allocates servers in the content delivery infrastructure to requesting clients in a way that, for web content delivery, minimizes a given client's response time and, for streaming media delivery, provides for the highest quality. The distribution infrastructure consists of on-demand or push-based mechanisms that move content from the origin server to the surrogates. An effective CDN serves frequently-accessed content from a surrogate that is optimal for a given requesting client. In a typical CDN, a single service provider operates the request-routers, the surrogates, and the content distributors. In addition, that service provider establishes business relationships with content publishers and acts on behalf of their origin server sites to provide a distributed delivery system.

[0038] As seen in **FIG. 1**, an Internet content delivery infrastructure usually comprises a set of "surrogate" origin servers **102** that are located at strategic locations (e.g., Internet network access points, and the like) for delivering copies of content to requesting end users **119**. A surrogate origin server is defined, for example, in IETF Internet Draft titled "Requirements for Surrogates in the HTTP" dated Aug. 9, 2000, which is incorporated herein by reference. The request-routing mechanism **104** allocates servers **102** in the content delivery infrastructure to requesting clients. The distribution infrastructure consists of on-demand or push-based mechanisms that move content from the origin server to the surrogates. A CDN service provider (CDNSP) may organize sets of surrogate origin servers as a group or so-called "region." In this type of arrangement, a CDN region **106** typically comprises a set of one or more content servers that share a common back-end network, e.g., a LAN, and that are located at or near an Internet access point. Thus, for example, a typical CDN region may be co-located within an Internet Service Provider (ISP) Point of Presence (PoP) **108**. A representative CDN content server is a Pentium-based caching appliance running an operating system (e.g., Linux, Windows NT, Windows 2000) and having suitable RAM and disk storage for CDN applications and content delivery network content (e.g., HTTP content, streaming media and applications). Such content servers are sometimes referred to as "edge" servers as they are located at or near the so-called outer reach or "edge" of the Internet. The CDN typically also includes network agents **109** that monitor the network as well as the server loads. These network agents are typically co-located at third party data centers or other locations. Mapmaker software **107** receives data generated from the network agents and periodically creates maps that dynamically associate IP addresses (e.g., the IP addresses of client-side local name servers) with the CDN regions.

[0039] Content may be identified for delivery from the CDN using a content migrator or rewrite tool **106** operated, for example, at a participating content provider server. Tool **106** rewrites embedded object URLs to point to the CDNSP domain. A request for such content is resolved through a CDNSP-managed DNS to identify a "best" region, and then to identify an edge server within the region that is not overloaded and that is likely to host the requested content. Instead of using content provider-side migration (e.g., using the tool **106**), a participating content provider may simply direct the CDNSP to serve an entire domain (or subdomain) by a DNS directive (e.g., a CNAME). In either case, the CDNSP may provide object-specific metadata to the CDN content servers to determine how the CDN content servers will handle a request for an object being served by the CDN. Metadata, as used herein, refers to a set of control options and parameters for the object (e.g., coherence information, origin server identity information, load balancing information, customer code, other control codes, etc.), and such information may be provided to the CDN content servers via a configuration file, in HTTP headers, or in other ways. The Uniform Resource Locator (URL) of an object that is served from the CDN in this manner does not need to be modified by the content provider. When a request for the object is made, for example, by having an end user navigate to a site and select the URL, a customer's DNS system directs the name query (for whatever domain is in the URL) to the CDNSP DNS request routing mechanism. Once an edge server is identified, the browser passes the object request to the server, which applies the metadata supplied from a configuration file or HTTP response headers to determine how the object will be handled.

[0040] As also seen in **FIG. 1**, the CDNSP may operate a metadata transmission system **116** comprising a set of one or more servers to enable metadata to be provided to the CDNSP content servers. The system **116** may comprise at least one control server **118**, and one or more staging servers **120***a-n*, each of which is typically an HTTP server (e.g., Apache). Metadata is provided to the control server **118** by the CDNSP or the content provider (e.g., using a secure extranet application) and periodically delivered to the staging servers **120***a-n*. The staging servers deliver the metadata to the CDN content servers as necessary.

[0041] **FIG. 2** illustrates a typical machine configuration for a CDN content edge server. Typically, the content server **200** is a caching appliance running an operating system kernel **202**, a file system cache **204**, server manager software **206**, TCP connection manager **208**, and disk storage **210**. Server manager software **206**, among other things, creates and manages a "hot" object cache **212** for popular objects being served by the CDN. It may also provide other CDN-related functions, such as request routing, in-region load balancing, and the like. In operation as an HTTP cache for

example, the content server **200** receives end user requests for content, determines whether the requested object is present in the hot object cache or the disk storage, serves the requested object via HTTP (if it is present) or establishes a connection to another content server or an origin server to attempt to retrieve the requested object upon a cache miss. Typically, the edge server operates in a "pull" manner, wherein an object is pulled into the cache initially upon the first request to the cache—which will generate a cache miss since the object is not present.

[**0042**] The present invention is a CDN Java application framework offering comprising Java-enabled edge servers. A given edge server (the machine) such as illustrated above in **FIG. 2** is assumed to include application server code. As is well-known, an application server is a software platform (sometimes called middleware) on which applications can be deployed. It provides useful utility services and functions to applications. There are currently several major types of application servers, Java-based (J2EE) and Microsoft .NET. Java, of course, is a programming language and a platform, and the programming language is object-oriented and platform independent. Applications written in Java are translated into Java byte code, which code is then run on (intepreted by) a Java Virtual Machine (JVM). In a preferred embodiment of the invention, given edge servers in the CDN are provisioned with a Java application server and additional code to enable Java applications or application components to be executed from the edge of the Internet. The framework can take advantage of and leverage the mapping, load-balancing and management systems used with known CDN offerings, such as the CDN illustrated in **FIG. 1** (which is merely representative). In a first embodiment, the application server is a servlet container (e.g., Apache Tomcat), in which case the present invention enables the offloading and execution of the Web tier of n-tier Java-based applications. JSP, servlets, Java beans and custom tags, which are executed within an application server's servlet container, are executed at the edge of the Internet, close to the end-user. The Web tier is typically the front end of a J2EE server. In an alternate embodiment, in addition to the Web tier, at least some or all of the Enterprise tier of the application is also deployed to and executed on a given edge server. The Enterprise or "business" tier typically hosts application-specific business logic and provides system-level services such as transaction management, concurrency control, and security.

[**0043**] The present invention advantageously enables a J2EE-compliant application to run in an edge-origin server environment. In particular, the inventive framework preferably leverages a distributed computing platform by distributing the application across the origin and the CDN. As noted above, typically the application contains servlets, JSPs, filters, tag libraries and Java beans/helper classes in a Web tier, and enterprise beans in an enterprise tier. Separation of the Web tier from the Enterprise tier, with execution of the Web tier (e.g., in a Web container) on the edge servers and the Enterprise tier (e.g., in an Enterprise container) on the origin site, is illustrated in **FIG. 3**. In this embodiment, the edge-enabled version of the application typically comprises two cooperating applications: an edge-side application and an origin-side application. Components of the Web tier may be packaged as a Web Archive (WAR), and components of the Enterprise tier may be packaged as an Enterprise Archive (EAR). As described above, the creation of these two applications typically requires decisions based on knowledge of the application, namely, decisions about which processes should run on the origin and which at the edge.

[**0044**] The inventive framework is not limited to running the Enterprise tier in an Enterprise container on the origin, however. As illustrated in **FIG. 4**, the Enterprise tier may also be distributed out to the edge servers and executed with the Web tier in an application server **400**. In this embodiment, the Enterprise tier (for illustrative purposes only) comprises one or more Enterprise JavaBeans (EJB) elements as session beans **402**, entity beans **404** and message driven beans **406**. To support the Enterprise tier, session beans preferably are persisted into globally coherent state. Entity beans can be used to replicate (at the edge server) read-only databases and to provide transparent tunneling (e.g., using JDBC over SOAP) to an enterprise database.

[**0045**] In a representative embodiment, an application server is IBM WebSphere 5.0 application server (WAS). IBM WebSphere uses JVM (Java Virtual Machine) 1.3.1, available from IBM. In **FIG. 3**, in contrast, the Web tier is executed in a Web container **300**. In this example, the Web tier comprises such elements as JSP pages **302**, servlets **304** and JavaBeans **306**. A representative Web container is provided by Apache Tomcat servlet container, which uses the JVM in JDK 1.3.1_04 available from Sun Microsystems. Of course, these components are merely exemplary and are not meant to be limiting. Preferably, a Web or Enterprise container runs in multiple instances on CDN edge servers, preferably under application isolation as will be described.

[**0046**] In particular, preferably each application is run in an isolated environment via a sandboxing mechanism implemented, e.g., in the JVM. Generally, sandboxing is accomplished by monitoring the resource (e.g., CPU, memory, disk, network I/O) utilization of each application server process. If an application server process over-utilizes resources, it is terminated, and a new application server is started. If an application server induces multiple restarts dues to excessive resource utilization, it is blocked from causing another restart. Preferably, a separate application server process is used for each CDN customer, as this prevents one customer's application from stealing resources from another customer's application. It also isolates application server restarts. In addition, each application server process preferably is run within its own sandboxed directory, outside of which it cannot read or write files. This prevents one customer's application from interfering with another customer's application, or one customer's application accessing another customer's data. Additional details regarding resource management and sandboxing are set forth below.

[**0047**] **FIG. 5** illustrates an edge server architecture. The server **500** preferably runs on commodity hardware running an operating system (e.g., a modified form of Linux) **502**. The Java stack includes a Java Virtual Machine (JVM) **504** and preferably a J2EE-compliant application server **506**. For Web tier components (such as illustrated in **FIG. 3**), the application server **506** may be implemented with Apache Tomcat servlet container as noted above. For Web tier and Enterprise tier components (such as illustrated in **FIG. 4**), the application server **506** may be implemented with IBM

WebSphere Application Server (WAS). These products, of course, are merely exemplary. According to the invention, the framework (preferably the JVM) creates and maintains application sandboxes **508** for each of the applications **510***a-n*. A given customer may run application **510***a*, while another customer runs application **510***b*. Generalizing, the edge server **500** supports multiple discretely-executable applications. The edge server **500** implements a cache **512** and maintains customer configuration data **514** that controls when application components are used. The server manager **516** overlays and controls the cache, using the customer configuration data. Application sandboxing prevents applications from hurting each other, the server, or gaining access to the code and data of other customers. As noted above, sandboxing also facilitates resource allocation to enable the server to monitor and control the use of CPU, RAM, disk, bandwidth and the kernel. System management **518** and system security **520** modules are also provided to facilitate these and other functions.

[**0048**] **FIG. 6** illustrates how an end user client browser **600** interacts with a content delivery network edge server **602** and an origin site **604** to facilitate execution of the application (and, in particular, its Web tier components) on the edge of the network. In this example, it is assumed that the Web tier components of the application are available for deployment and execution on the edge server. As described above, the edge server **602** has a Java processor **606**, a cache **608**, and a set of customer configuration data **610**. The origin site **604** executes a Java application server **612** and includes data sources **614**. To utilize the platform, an enterprise creates a DNS alias (e.g., a canonical name or CNAME) that points their Internet domain name to the Internet CDN service provider. Consequently, any lookup for the customer's domain name (e.g., www.site.com) results in a lookup for the IP address of an aliased domain. Because the CDNSP's DNS is responsible for resolving these queries, the CDNSP returns the IP address of an optimal edge server, in this example edge server **602**. This is step (1). The decision about which server to resolve the user to typically is based on network congestion, network proximity, server load and bandwidth utilization. At step (2), the edge server **602** applies the customer's configuration data **610** to the request, determining if the request should be serviced using the edge server's local cache **608** or Java processor **606**, or forwarded (e.g., via tunneling) to the customer's origin server **604**. Thus, when the edge server receives a request from a client, preferably it first matches the request with an appropriate customer configuration file. If the customer configuration file associates Java processing with the request, the Java processor **606** is engaged. If the request is for a servlet or a JSP page, the Java processor **606** fulfills the request. This is step (3). In particular, when a request is received from an application whose WAR file is already in the edge server **602**, the Java processor **606** uses the applicable servlet or JSP page (for example) to generate a response for the incoming request. A standard deployment descriptor (e.g., web.xml) may be used to properly map the request(s) to a servlet. If this is the first request that uses this particular web application, the application components (e.g., a WAR file) are retrieved from the origin site or a CDN staging area. As an optimization, the first request can be tunneled to the origin site for processing, while the edge server asynchronously retrieves the WAR file to handle future requests. If the servlet requires a data resource, it may obtain that resource from cache **608**. This is step (4). Alternatively, if the servlet is forwarding a request to another (possibly non-edgeable) servlet, the Java processor **606** on the edge server contacts the origin site. As indicated in step (5), communication between the edge server and the origin server is through RMI, SOAP or explicitly through HTTP. RMI enables an edge application to use a remote object as if it was local. SOAP provides an XML-based RPC mechanism for communicating with remote objects. Alternatively, a servlet may retrieve data through an HTTP request in any other format. Preferably, the CDN service provider provides classes that can be used to query XML data. The retrieved data may also be cached in cache **608**, eliminating inter-tier latency for future requests. As indicated at step (6), the edge server **602** completes the processing of the request and returns the response to the client. Preferably, the executed servlet remains in memory, ready for a request from a next user that is mapped to the edge server.

[**0049**] **FIG. 7** illustrates one way in which an application can be developed and deployed to facilitate edge processing. An application (or component thereof) that is designed for execution on an edge server of a content delivery network is sometimes referred to as an "edge-enabled" application. As illustrated in **FIG. 7**, after an application **700** has been developed through a software development phase, it may be split into two parts, e.g., by running a splitter or other code development tool, producing, for example, an edge WAR file **702** and an origin WAR file **704**. In an illustrative embodiment, the edgeable components **702** are then prepared for deployment on the CDN edge server(s) **706**, while the full application is prepared for deployment on the origin site **708**. Any convenient technique to allows the developer to specify which components are edgeable, and which are dependent on the centralized resources, can be used with this invention. Preferably, the application developer creates the application using n-tier design principles. Of course, the application development process need not include creation of a single, original WAR file, as the edge and origin components can be designed and built separately in the first instance.

[**0050**] The following are some additional guidelines for edge-enabling an application for the framework in an embodiment in which just the Web tier is located on the edge. In this embodiment, enterprise beans run at the origin, and calls to the enterprise beans (including use of home or remote interfaces) preferably do not exist in edge-located filters, servlets, helper classes or beans. Preferably, direct calls to origin-based system resources, such as a database, do not exist in edge-located servlets, helpers or beans. In such case, however, database connectivity is provided, preferably using a Type 3 JDBC driver. Also, any filters, servlets or JSPs that require servlet context preferably do not access the ServletContext of a different web application. In this embodiment, Web applications can use ServletContext attributes to store state. For security reasons, certain web components may need to run at the origin. The web application preferably adheres to the "distributable" conventions described in Servlet Specification 2.3, including marking the web application as "distributable" in its deployment descriptor. Web components in an execution sequence followed in response to a request preferably run entirely at the origin or entirely at the edge in response to this request. A web application edge component that uses request dispatching (include/forward) preferably can only dispatch to another

edge web application component; the same is true for an origin component. However, the source or target (dispatched) edge component is free to contact the origin to send data, retrieve data, or the like.

[0051] An execution sequence normally consists of filters, servlets and JSPs that are involved in response to a request, but preferably it does not include external resources that are used via connections to the origin (such as HttpURLConnection). Preferably, the same request and response argument are shared by the filters that are executed, and by servlets and JSPs that include or forward to each other to form the execution sequence. The definition is dynamic, because a servlet could be included in edge-side and origin-side execution sequences without contradiction.

[0052] With knowledge of the legal execution requests in the application and the set of requests that cause these execution sequences to be followed, a developer can edge-enable the application. In one embodiment, this process involves identifying components as origin-only, edge-only or both. Origin-only components can run on the origin, preferably unchanged. Edge-only components run only at the edge. The both designation applies to a servlet that could be on an execution path to an origin-only servlet and also on an execution path in which all servlets are edgeable. In this case, the servlet needs to be installed at the origin as well as the edge. The both category might also apply to a servlet serving a comparable function at the edge and at the origin. Some components may best be split into edge and origin components.

[0053] To construct the request sets and corresponding execution sequences, the deployment descriptor (web.xml) can be used to obtain servlet-mapping values and URL-patterns corresponding to them. For those components that should be split into edge and origin components, it is desirable to create an edge-side component of the same type and one or more origin-side servlets. This can be done by factoring out the origin-side functionality to create the edge-side component and using servlet facades for the origin-side system calls. Components needed both at the edge and at the origin are marked both, and the remaining components are marked edge.

[0054] An edge dispatcher is then constructed. An edge dispatcher is a single entry point into the web component at the edge. This dispatcher servlet examines an input request and decides to proxy it to the origin or to forward it to a servlet/JSP on the edge. If the pre-edge-enabled web component (i.e., without the dispatcher) already has a single entry point, then the dispatcher functionality can be built into this entry point itself. To construct this component, consider each request set and its corresponding execution sequence. If the execution sequence includes a component marked origin-only, then the corresponding request set must be proxied to the origin (and the filters at the edge must be configured to ignore these requests). Otherwise, the request can be satisfied at the edge and the edge dispatcher forwards it to the first edge-side servlet or JSP in the execution sequence.

[0055] In addition, to edge-enable the application, some deployment information in the web.xml deployment descriptor must be altered, in particular the servlet-mapping and filter-mapping values to make sure that all requests are routed through the edge dispatcher. Also, filters preferably are not applied twice (e.g., first at the edge, and then at the origin) on requests that are proxied to the origin. Alternatively, one could set up edge-filters and origin-filters. The webapp must adhere to the "distributable" conventions described in Servlet Specification 2.3, including the fact that it must also be marked as "distributable" in its deployment descriptor. The deployment information in the deployment descriptor is altered (particularly the servlet-mapping and filter-mapping values) to make sure that all requests that routed through the edge dispatcher, and that filters are appropriately applied.

[0056] Typically, the edge dispatcher receives the request and determines its handling. As illustrated in **FIG. 8**, which is merely exemplary, the request may be processed entirely at the edge by the edge components **802**. Alternatively, the dispatcher **800** may serve as a proxy and send the request to the origin **804**, which might in turn call origin processes such as enterprise beans **806**, which return the response to the proxy which in turn responds to the client. In a split scenario, the dispatcher **800** sends the request to the edge-side component **810**. The edge component **810** communicates with the origin-side split component **812**, which in turn may call origin processes such as the beans **806**. The response return via the edge side component **810**.

[0057] In the above approach, a servlet/JSP on the edge (the proxy) marshals arguments and sends them to a servlet at the origin (the broker), which parses the arguments and performs the requisite method invocation. Then, the broker marshals the return value and sends it back to the proxy. The broker exports origin-side functionality to the edge and serves as a facade for this functionality. In particular, any communication between an edge servlet/JSP and an enterprise bean is preferably via a servlet facade at the origin. An alternative to the design is to have a single origin-side servlet that mediates between the edge and all serlet façades at the origin. This provides a single entry point for edge-origin requests. An origin dispatcher could itself provide all the functionality of all servlet facades that would otherwise exist at the origin.

[0058] The following describes modifications to a Java application server, specifically its servlet container component, to integrate into the inventive framework. This application server is executed on an edge server, which, as noted above, is a machine running commodity hardware and an operating system. As illustrated in **FIG. 9, a** preferred architecture is implemented via out of process architecture and comprises an edge server process **900** and multiple Java application server processes **902***a-n*. An edge node in the content delivery network preferably has a single edge server application that can spawn multiple child processes each containing an application server instance, as was illustrated in **FIG. 8**. Each child process preferably is configured for a Java Edge Services API (JESAPI), which according to the invention is an integration framework for a Java application server. Generally, JESAPI interfaces the edge server manager process to the application server instances to facilitate various administration functions, namely, the starting, stopping and reloading of WAR and EAR files, the monitoring of the health of the various application server instances, the monitoring of resource usage by the application server instances, and the collecting of data from the instances to facilitate reporting and billing for use of the platform. As illustrated in **FIG. 9**, an HTTP/HTTPS request first connects

to the edge server process **900**. The edge server process **900** preferably maps the request to a context path that is preferably specified in a metadata configuration from the customer configuration data. The edge server process **900** then fetches and unpacks an associated web application archive (WAR) on a file system, and installs the archive. Finally, the edge server process modifies the request to be handled by an application server instance and proxies it using sockets. Additionally, the edge server process **900** preferably employs bi-directional communication with each JESAPI application server child instance, transmitting such information as control data and resource usage.

[**0059**] **FIG. 10** illustrates a Java application server instance. The Java application server child process **1000** contains the application server core logic **1002** and is enabled for JESAPI support **1004**. An application wrapper **1006** process component is specific to the application server. Its purpose is to integrate and orchestrate the various components of the process. The JVM/JDK **1008** is conventional should not involve any modifications. An external shared object in the JVM intercepts system calls made in the application server process **1002**. It monitors resource usage and performs security access checks, as will be described in more detail below. The JESAPI **1004** preferably comprises a set of Java classes and a native library, and it defines the core integration framework for the Java application server process **1002**. Although not meant to be limiting, preferably JESAPI relies on the application server process **1002** providing a custom JESAPI implementation singleton object that extends a provided JesapiBase abstract class.

[**0060**] The application wrapper **1006** acts as the bootstrap logic for the application server process **1002**. The wrapper **1006** is customized to the application server type and acts as "glue" code connecting all the various components of the process. The wrapper component **1006** provides a JESAPI implementation singleton specific for the application server type, which may vary. In particular, the wrapper **1006** initializes JESAPI **1004**, performs any necessary runtime configuration of the application server process **1002**, starts the server, and notifies JESAPI when the server is ready to process requests. Because it is the entry point for the application, the wrapper must initialize JESAPI and the application server with the data supplied to it by the edge server process (element **900** in **FIG. 9**) (in the form of arguments, Java system properties, and the like). The data includes, for example: an application server instance id (used by JESAPI) and the socket port the servlet container must be on for HTTP connections. The application wrapper **1006** preferably configures the edge server to only accept HTTP socket connections. In an illustrative embodiment, the application server process must accept connections bound for the local loopback host and on the port specified by the edge server process. Additionally, the application wrapper provides and registers any handlers with the application server necessary for integration, such as protocol handling and logging. Preferably, the application wrapper receives each application server log event (server and per webapp) and routes it to JESAPI. The log handling API provided by the application server preferably invokes the handler in the same thread that issued the log message, and this thread forwards the message to JESAPI. Because application server log messages are redirected to JESAPI via the application wrapper log handlers, file logging can be disabled in the application server. Other standard data streams from the application server likewise are redirected to JESAPI via the application wrapper.

[**0061**] Preferably, and as described below, the application server process **1002** uses J2EE security policies to restrict the functionality of web applications as well as server code itself. Preferably, the server code is locked down as much as possible to avoid security loopholes. Also, the JESAPI implementation singleton and any classes that are part of the application wrapper preferably have the same protection as server classes. In addition, preferably there are appropriate security restrictions imposed on the entire process (including server and web application logic).

[**0062**] Aside from the features offered by the standard J2EE security permissions, additional restrictions should be imposed for the applications (sometimes referred to as "webapps"). Preferably, web applications are prevented from creating or modifying threads and thread groups. If a web application runs in a non-system thread, the application server process provides a way to address security permissions. A web application also should be allowed to perform JNDI and file read-only operations recursively from its base path (the unpacked WAR file directory root). Preferably, the application server dynamically creates security permissions for the web application at runtime.

[**0063**] Because web applications from different customers preferably can run on the same server, the servlet container preferably is configurable to allow/disallow a web application in one context to access the ServletContext instance of a different context; when servlets attempt to call ServletContext.getContext( ), depending on the configuration for the web application, null may be returned. Preferably, this operation is specified per web application at install time. As an added level of security, an external shared object preferably traps system calls made in the application server process and performs access control checks, as will be described below.

[**0064**] Prior to forwarding any HTTP requests for a particular web application in the application server, the edge server process (element **900** in **FIG. 9**) is responsible (if necessary) for unpacking the associated WAR to a base directory on the file system and installing the web application components in the application server. The edge server process notifies the application server process to install and invalidate a web application using JESAPI, supplying the web application configuration at runtime. The edge server process is also responsible for managing which contexts are installed in each application server instance. When the edge server process requests the application server to install a web application, the edge server process sends a control request to JESAPI supplying the web application's context path, its base directory path, and a flag that determines if the web application will be able to access other servlet contexts, e.g., using ServletContext.getContext( ). The JESAPI implementation singleton then processes this request, e.g., by invoking the application server's dynamic web application installation mechanism. After the web application gets installed, the edge server process sends requests for it to the application server. When the edge server process is ready to invalidate a particular web application, it stops sending requests to that web application instance and sends a web application uninstallation control request to JESAPI identifying the web application with its context path.

[0065] To support web application revisions by hot swapping, the edge server process preferably generates an artificial context path used for web application installation, invalidation, and regular requests in the application server. The context path preferably consists of the context id, a hash of various values that identify the web application instance including the host name, original context path, WAR contents, and revision number. If a new application version (e.g., Version 1.1) is published while an old application version (e.g., Version 1.0) is active, the new application version is placed in the same process (as the original version), and new requests are directed into the new application version. When the old application version drains of requests, that application version is terminated and appropriate clean-up effected. Preferably, both versions of the same customer application run in the same process, although this is not a requirement. This "hot swapping" technique is illustrated in **FIG. 11**.

[0066] With explicit web application installation, the edge server process thus sends a web application install command to JESAPI and waits for a successful response before forwarding any HTTP requests associated with the web application to the application server. If an explicit install request occurs because the edge server process encounters a request for a not yet installed web application, there is added latency for processing that initial request because of the explicit install roundtrip. As an alternative, an implicit web application install may be performed to minimize the delay incurred by the first request for a web application that is not yet installed. Instead, the edge server process forwards the request to pre-installed JESAPI webapp (at JESAPI startup) in the application server that will both install the specified web application and have that web application process the original request. This is achieved in a single pass between the edge server process and the application server process. To accomplish this, the edge server procsess modifies the original HTTP request to provide the added data to the JESAPI web application so it can install the application and then have it process the request.

[0067] A preferred request processing operation is illustrated in **FIG. 12**. Preferably, JESAPI requires the application server to assign a distinct thread to process each request. The same thread preferably makes all JESAPI calls for processing that request. After the edge server process receives a request and takes care of installing the associated the webapp as necessary in the application server, but before forwarding it to the installed webapp context, the edge server process modifies the HTTP request to correctly get processed by the application server and a JESAPI Servlet Filter. Specifically, the edge server process alters the request's URI to contain the artificial context path (so the application server can correctly map the request to the previously installed unique context instance). The edge server process also inserts various JESAPI internal headers that provide the JESAPI Servlet Filter with more data about how to handle the request.

[0068] **FIG. 13** illustrates a preferred implementation where multiple application server instances are instantiated, preferably one per CDN customer that is using the edge server. Thus, there is preferably one application server per JVM instance per customer, although this is not meant to be limiting. In this example, edge server **1300** is a machine having an operating system **1302** such as the Linux kernel. An edge server manager process **1304** communicate with the child Java application server instances **1306***a-n* preferably via TCP sockets and using a shared memory **1308**. Each Java application server instance runs atop its own JVM **1310**. Thus, in this embodiment, there is preferably one application server/JVM instance per customer, and the application server/JVM instances are run out of process from the edge server manager. Preferably, the child application server processes are forked from the edge server manager, after which they are tightly monitored and controlled by a Java manager subsystem **1312**. The edge server manager forwards a client request that require application server processing over a local TCP socket to a child application server process, which processes the request, and sends the response on the same connection. In addition, resource utilization load is reported from each application server process, across a shared memory segment **1308**, to the Java manager subsystem **1312**. The manager subsystem **1312** tightly monitors resource utilization of each child application server process, and it will kill application server processes that over utilize resources.

[0069] In particular, resources consumed by each child application server process are monitored, preferably by shared object components that are loaded by each application server process at startup. These include a Java Edge Services API (JESAPI) shared object **1314**, and an intercept shared object **1316**. The JESAPI shared object **1314** implements specific JESAPI Java native calls, and it is responsible for communicating across the shared-memory segment **1308** with the Java manager subsystem **1312**. The intercept shared object **1316** preferably loads various "intercept" system calls such as "open,""close,""gethostbyname" and the like. By intercepting system calls, the manager subsystem **1312** can prevent access to some calls, or make intermediate calculations, or accrue statistics, or the like, before making the "real" system call that the application server intended to make. The Intercept shared object reports any resource utilization to the JESAPI shared object, which then reports it across the shared memory segment to the Java manager subsystem.

[0070] The following resources may be monitored for each application server process: memory—the memory used by the JVM's internal Java heap (i.e. the heap in which it does memory management for Java objects allocated by the application server, and the webapps that run in the application server); CPU—the CPU time consumed for each request while it was active inside the application server; disk—the disk operations that the application server performs, including disk operations done as a result of a client request (the JESAPI shared object may also check whether a disk read was from disk or from buffer cache so that flits can be properly attributed to the request); and network—the number of sockets that are opened by each application server process to fetch include URLs. The Java manager subsystem **1312** performs resource management, e.g., through a set of policies based on resource utilization. Thus, for example, the Java manager will kill a child application server process for over-utilization of the following resources in the following ways: memory—if the application server's Java heap uses more memory than a configurable amount set in customer metadata, it will be killed; runaway requests—a runaway request is a request that has been processing for an "unreasonible" amount of time (a configurable number), and if an application server generates a certain configurable number

of runaways, it will be killed; open sockets—if an application server reaches a configurable limit of open sockets (for which it has never called close), it will be killed, or the like. This rate limiting of resources ensures that no application server instance can become an exclusive user of the server's resources.

[0071] In addition to the above-described resource management, the Java Security Manager framework facilitates sandboxing by imposition of security restrictions to web applications running in each application server process. Preferably, this is achieved through a combination of a security policy file, and a Java Security Manager implementation. The following restrictions preferably are placed on Java web applications in this manner: file system—customer web applications cannot read or write to the file system (although they can read files from within their own WAR file such as static html); socket—customer web applications cannot open Java sockets; threads—customer web applications are not allowed to start/stop/modify Java threads; and code—customer web applications are prevented from reading JESAPI or application server classes. In the case of sockets, preferably a customer webapp can fetch include files through the HttpURLConnection Java class that is intercepted by JESAPI code and that forces the include to go only through the edge server manager process (and monitors the number of open connections). In addition, preferably the framework allows customers to open raw Java sockets. This is because the previously mentioned intercept shared object will intercept all of the socket API calls, and monitor the number of connections made by the application server process. The intercept object will then connect to the edge server manager process using the HTTP CONNECT method, and the edge server manager process will then open a socket to the desired host.

[0072] The resource management, sandboxing and security features described above are merely exemplary. Other techniques may be used, for example, resource management by user ID. In such case, after each application server process is launched, a setuid is performed, setting the process to a unique user ID. Once set to this unique UID, other operating system kernel features for resource management can be used. These include total thread limit, file sytem quotas, socket filters, and the like. In addition, this approach enables use of other system calls (e.g., "chroot") to limit the application server process to a subset of the filesystem, outside of which it will not be able to read or write.

[0073] One or ordinary skill in the art will appreciate that the JESAPI interface can be designed such as described above to support application servers unchanged. Alternatively, a given application server vendor may modify given application server functionality as appropriate to enable the application server to run on the CDN server provider's edge server platform, in which case certain changes to the servlet container may be necessary for it to be run on the edge server. Thus, for example, where possible, a new subclass of an existing servlet container component should be created (as needed) and then modified to interface to the edge server manager. In either case, preferably the edge server manager interfaces client requests to and from the edge server itself.

[0074] Some additional aspects of the edge-enabled application framework are now described below, and several examples are also provided.

[0075] Customer Configuration

[0076] When an edge server receives a request from a client, preferably it first matches the request with an appropriate customer configuration file. The configuration file may be delivered to the edge servers via any convenient mechanism, such as a CDN metadata transmission system as illustrated in **FIG. 1**. Of course, any convenient technique for providing the customer configuration data to the edge servers can be used. If the customer configuration associates Java processing with the request, the Java processor is engaged as has been described.

[0077] Web Container

[0078] As noted above, if the WAR file is already in the edge server, the Java processor uses the applicable servlet or JSP page (for Web tier processing) to generate a response for incoming requests. A standard deployment descriptor preferably is used to properly map the requests to a servlet. If the Java application is not currently on the edge server, it is retrieved from the origin site or from some other source. Because the retrieval process may cause a significant latency, the application may be retrieved asynchronously, while the initial request is tunneled to the origin server simultaneously. The output of the processed request is returned to the user. The executed servlet preferably remains in memory ready for the next user.

[0079] Network and Resource Management

[0080] Servlets preferably are managed to make sure no process consumes an undue amount of resources. Proper resource monitoring and load balancing assures that no application affects another one running at the same time. The Java application may make requests for content through the network. The requests preferably are made through HTTP and HTTPS protocols. Remote Invocation of other Java resources is also preferably done through HTTP.

[0081] The Role of the Origin Site

[0082] The origin server may remain an integral part to the edge application, especially when just the Web tier is deployed on the edge network. In addition, because some servlets rely on access to centralized resources, not all requests can be processed by the edge server. In such case, the origin site is responsible for fulfilling the non-edgeable requests, as well as answering any remote calls that might be made by the edge-deployed application.

[0083] The following are the typical responsibilites of the origin site in such circumstances: respond to RMI requests from the edge tier, respond to HTTP requests from static and dynamic content, set Host Response Headers (HRH) for controlling edge server behavior as necessary, serve WAR files when requested by the edge servers, and respond to JDBC requests from the edge tier.

[0084] Edge-to-Origin Communication

[0085] The communication between the servlet on the edge server and the origin site preferably occurs through HTTP or HTTPS protocols as follows: Remote Method Invocation (RMI) communication is tunneled through HTTP; SOAP messages are exchanged over HTTP or HTTPS; JDBC is tunneled over HTTP/HTTPS; responses to relational database queries are encoded in XML (allowing the edge server to cache the results, re-use them with the

future requests, and minimizing the inter-tier latency); Servlet control methods (e.g., RequestDispatcher.include( ) and RequestDispatcher.forward( )) are preferably supported regardless of whether the communication is edge-to-origin, or origin-to-edge communication; and custom communication solutions are supported provided messages are transported over HTTP or HTTPS. **FIG. 14** is illustrative of these techniques.

[**0086**] To ensure that the application is scalable and benefits from being on the edge, the amount of bytes sent and the number of calls between edge and origin should be minimized. This can be accomplished, for example, through caching of the data on the edge, and through the use of a data-access facade (instead of making multiple calls to a database, in which case an edgeable servlet is used to call a non-edgeable servlet to make the database calls on its behalf).

[**0087**] The present invention delivers the ability to run Java-based web applications at the edges of the Internet, near the end user, providing several benefits. The web application will be served by as many servers as necessary to maximize the performance. New servers are allocated automatically based on increased traffic, without capital expenditure by an enterprise. Offloading applications from the origin to a distributed network can eliminate single points of failure. In addition, monitoring of edge servers, built-in redundancies and the ability to map users instantly to the optimal servers allows the CDN service provider to bypass network congestions and overcome hardware failures. Offloading application processing from a single origin to to numerous servers at the edge can result in significant performance gains. By mapping each user to an optimal or preferred server, the CDN service provider avoids Internet bottlenecks and can dramatically reduce latency. The ability to allocate servers on demand means applications will never lack processing power or bandwidth. By reducing the number of application servers needed to run at the origin site, the CDN service provider reduces complexity associated with hardware and software maintenance and management.

[**0088**] There is no limitation as to the particular type of application component that may be implemented and deployed as an edge-enabled CDN application. In addition to the examples set forth above, representative applications include, without limitation, product configurators, dealer locators, contest engines, content transcoders, content generators, search aggregators, financial calculators, registration engines, and a myriad of others.

[**0089**] One of ordinary skill will recognize that many variants are within the scope of the present invention. Thus, for example, a particular edge server may execute a first type of application server instance (e.g., Tomcat servlet container) as well as a second, different type of application server instance (e.g., IBM WebSphere Application Server). As already described, multiple instances of a particular application server will typically be used on a given edge server to facilitate use of that server by multiple service provider customers. Of course, other Web containers besides Apache Tomcat can be used to implement the Web tier, and other Enterprise containers besides IBM WebSphere Application Server can be used to implement the Enterprise container. There is no requirement that a particular application have components that execute on both the edge and the origin; indeed, a given application may execute in a standalone manner completely as an edge-enabled application. There also is no requirement that the application components be packaged as WAR or EAR files, as any convenient mechanism may be used to deploy the application components to the edge. There is no requirement that application components be loaded only in response to client requests at a particular edge server. Indeed, in many cases it will be desirable to pre-deploy an application or an application component based on some prediction of expected future need for that application or component, or for purposes of fault tolerance. Thus, a given application or component thereof may be delivered to a particular edge server and initialized and started irrespective of whether an end user request has been received at the server. Also, there is no requirement that application components be fully or partially J2EE-compliant, or even that the subject matter be implemented entirely in Java. Indeed, the present invention is also extensible beyond Java and J2EE. In particular, the inventive concepts may be practiced in any platform-independent application server programming environment (e.g., Microsoft .NET, Mod Perl executing in Apache, Zope, or the like) capable of being deployed in a distributed computing environment such as a content delivery network.

[**0090**] The CDN service provider may provide the ability to test and debug the application within an enterprise firewall. A test server may be a CDN edge server simulator that can be used during application development and testing to validate the execution of the application on the platform's runtime environment.

[**0091**] To deploy a prepared edgeable application, the content provider preferably publishes the application (e.g., using FTP) to a CDN staging network. The staging network preferably is a set of staging servers, which may be the CDN edge servers or some other set. This creates a staging environment in which the application can be tested by the enterprise's quality assurance personnel. When tests prove satisfactory, the application is made live, preferably through a secure web interface. **FIG. 15** illustrates this process, which takes advantage of a service provider portal. Customers also may upload, deploy and provision applications programmatically. Deployment to the edge preferably occurs automatically. Edge-enabled applications or their components may also be deployed from a CDN content storage network or some other third party server. As already noted, application components also may be pre-fetched to a particular edge server or server region to reduce start-up latency. In a general case, however, an edge application component has not been pre-deployed and an end user has been mapped to a particular edge server. If the end-user request then matches the configuration parameters created during the setup phase, the edge server to which the end user has been mapped will attempt to load the associated Java application. If the Java application is not in cache, it is retrieved from the staging network, or the content storage network, or some other server. Preferably, the application continues to reside within the servlet container for the next request. Unused applications preferably are removed from the Web container but may still be stored in cache. Preferably, if an application has been invalidated, or if the application has not been accessed for an extended period of time, it is removed from the disk cache. To protect the Web application from unauthorized access, preferably only the

edge servers are allowed to retrieve the application from the staging network or the other content storage network.

Having described our invention, what we claim is as follows:

1. In an apparatus for use in a distributed computing platform and having a processor, an edge server manager, a virtual machine, and an application server, the improvement comprising:

code that interfaces the edge server manager to a set of one or more application server instances, each of which are selectively instantiated on a virtual machine instance to execute a given application component;

code that monitors given resource utilization by each of the application server instances and that, responsive to such monitoring, invokes a given action with respect to any application server instance that over-utilizes a given resource; and

code for preventing a given application component executing in a given application server instance from taking a given action.

2. In the apparatus as described in claim 1 wherein the given action invoked is the termination of the application server instance that over-utilizes the given resource.

3. In the apparatus as described in claim 2 wherein the given action invoked is a rate limiting of the application server instance that over-utilizes the given resource.

4. In the apparatus as described in claim 1 wherein the given resource is a resource selected from a set of resources that includes memory usage, CPU usage, disk usage and network I/O usage.

5. In the apparatus as described in claim 1 wherein the given action prevented by the given application component is selected from a set of actions that includes: restricting the given application component's reading or writing to a file system, restricting the given application component's ability to open sockets, restricting the given application component's ability to start, stop or modify threads, and restricting the given application's ability to read given server code.

6. A method operative in an apparatus having a processor, a virtual machine, and an application server, and a set of application components, comprising:

in response to requests, initiating a set application server instances;

executing the application components on the application server instances to respond to the requests;

during execution of the application components, monitoring utilization of given resources by each of the application server instances; and

responsive to the monitoring, invoking a given action with respect to any application server instance that over-utilizes a given resource.

7. The method as described in claim 6 further including the step of restricting a given application component executing in a given application server instance from taking a given action.

8. The method as described in claim 7 wherein the given action prevented by the given application component is selected from a set of actions that includes: restricting the given application component's reading or writing to a file system, restricting the given application component's ability to open sockets, restricting the given application component's ability to start, stop or modify threads, and restricting the given application's ability to read given server code.

9. The method as described in claim 6 wherein the given action invoked is the termination of the application server instance that over-utilizes the given resource.

10. In the apparatus as described in claim 6 wherein the given action invoked is a rate limiting of the application server instance that over-utilizes the given resource.

11. The method as described in claim 6 wherein the given resource is a resource selected from a set of resources that includes memory usage, CPU usage, disk usage and network I/O usage.

12. A method operative in an apparatus having a processor, a virtual machine, and an application server, and a set of application components, comprising:

in response to requests, initiating a set application server instances;

executing the application components on the application server instances to respond to the requests;

during execution of the application components, restricting a given application component executing in a given application server instance from taking a given action.

13. The method as described in claim 12 further including the steps of:

monitoring utilization of given resources by each of the application server instances; and

responsive to the monitoring, invoking a given action with respect to any application server instance that over-utilizes a given resource.

14. The method as described in claim 13 wherein the given action invoked is the termination of the application server instance that over-utilizes the given resource.

15. In the apparatus as described in claim 13 wherein the given action invoked is a rate limiting of the application server instance that over-utilizes the given resource.

16. The method as described in claim 13 wherein the given resource is a resource selected from a set of resources that includes memory usage, CPU usage, disk usage and network I/O usage.

17. The method as described in claim 12 wherein the given action prevented by the given application component is selected from a set of actions that includes: restricting the given application component's reading or writing to a file system, restricting the given application component's ability to open sockets, restricting the given application component's ability to start, stop or modify threads, and restricting the given application's ability to read given server code.

18. The method as described in claim 12 wherein each application server instance executes in association with a virtual machine instance.

* * * * *