Developing Edge Computing Applications

2Q2003

Authors

Gennaro (Jerry) Cuomo
Brian K. Martin
Keith B. Smith
Steve Ims
Helen Rehn
Marc Haberkorn
Jay Parikh
Contents

1. Preface ........................................................................................................................ 3
2. Introduction ................................................................................................................... 4
   Intended Audience ......................................................................................................... 4
   The Edge Processing Value Proposition ....................................................................... 4
3. Edge Processing Architecture ....................................................................................... 6
   Evolution of Application migration into the Network ..................................................... 6
   Edge Computing Roles .................................................................................................. 7
   Edge Computing Environment ....................................................................................... 9
4. Developing Edge Computing Applications ................................................................. 11
   Application Components ............................................................................................... 11
   Deployment to Akamai® ................................................................................................. 30
5. Edge Computing Results from the Lab ...................................................................... 31
   Topology ......................................................................................................................... 31
   Test Methodology ......................................................................................................... 31
   Results and Analysis ...................................................................................................... 32
1. Preface

Okay, I'll admit it. I'm a "midnight snacker". However, it is quite a task to get up out of bed, walk downstairs, through the hall and living room, and finally into the kitchen. Then, if I don't trip over one of my children's toys -- which can introduce significant delays towards my final destination, I arrive at the refrigerator, open it and get my snack. Being a "performance guy", I had an idea on how to streamline this process. I purchased a small refrigerator and placed it at the edge of my bed. In that fridge, I off-loaded a small stash of snacks from my main fridge to the new one. Now snacking at night is a breeze. I calculated that when it took me 5-10 minutes to complete the snacking process, it now takes me 1-2 minutes. The best part is I no longer risk injury along those long journeys down the stairs in the dark.

In a sense, this scenario outlines the essence of what the Edge Computing Project is trying to accomplish. However, instead of off-loading snacks, edge computing is focused on off-loading parts of your enterprise application into a value added network, like the one provided by Akamai®. By following the off-load design patterns outlined by the Edge Computing Project, the performance, scale and resiliency of your enterprise application will improve significantly.

-- Gennaro (Jerry) Cuomo, IBM Lead – The Edge Computing Project
2. Introduction

Intended Audience

This document is intended to be read by IBM internal communities, business partners and customers who are connected, in some way, to enterprises who are constructing large scale e-business applications. The contents of this paper should be shared with those looking forward to applying on demand technology to dramatically improve application performance, scale, resilience, and reduce total cost of ownership.

The Edge Processing Value Proposition

An Edge Processing Scenario

An enterprise wants to run a Web-based marketing promotion around a specific event; for example, a digital camera company wants to conduct an on-line photo contest to promote the sale of their cameras and photo finishing services. The promotion’s duration is for six weeks. The customer already has the existing content (e.g., HTML, images, etc.) for the dynamic Web site, but wants to develop some new Servlets and Java™ Server Pages (JSPs) for dynamically assembling the content based on user profile information. The customer faces two choices:

1. Lease or acquire, and configure the systems necessary to support the promotion and the Web traffic surge it creates.

2. Contract with a service provider, for example Akamai, to host the application and static content that it assembles and renders.

This paper focuses on the second choice, which utilizes Akamai EdgeComputing™ powered by WebSphere®.

Edge Processing’s Business Value

From this example, the business value of Edge Processing can be described as follows:

- **TCO**: The customer does not need to acquire, configure, and test systems that it needs only for a short period of time.

- **Responsiveness**: The customer does not need to plan for acquiring, configuring and testing systems in their timeline for reacting to a business opportunity.

- **Resilience**: Unplanned spikes in the workload do not degrade system performance, and denial of service attacks are mitigated

- **Performance and Scale**: Significantly reduced end user response time. Many end user requests are handled without a call from the “edge” over the Internet to the enterprise.
IBM and Akamai recently announced Akamai EdgeComputing powered by WebSphere (See - http://www.akamai.com/en/resources/pdf/EdgeComputing_Akamai.pdf). Akamai's EdgeComputing service provides a hosting environment for WebSphere-based J2EE applications. Utilizing this EdgeComputing platform (i.e., “the edge”) is one of the keys to achieving the business value of edge processing. However, building and deploying applications that leverage this new environment requires a slightly different approach to design and deployment. An efficient J2EE application running on the edge will need to follow best practices and be configured and deployed with policies which optimize their behavior when running in this widely distributed environment.

The Edge Computing Project is an on demand technology project focused on dramatically improving application performance, scale, resilience and reduced total cost of ownership. In particular, The Edge Computing Project is pursuing best practices and platform enhancements targeted at building enterprise applications that are split (or off-loaded) between an enterprise platform (e.g. consisting of WebSphere, DB2, Tivoli and Lotus) and Akamai’s EdgeComputing Platform powered by WebSphere. The edge computing sample application was developed as a living example of a split edge-enabled application. This paper is based on results from The Edge Computing project.

The application code discussed in this paper can be found (after November 6, 2003) at http://www.alphaworks.ibm.com/tech/edgecomputing.
3. EdgeComputing Architecture

Evolution of Application migration into the Network

Over the past several years, static content, dynamic page assembly, and application code have progressively migrated into the network. This section outlines this evolution.

Static Caching

Akamai made its name by caching static objects (GIFs, PDFs, other download objects) closer to the end-user, by partnering with over 1,000 internet providers and carriers around the world. Akamai servers installed at these providers, cached static Web content. Akamai’s EdgeSuite service drastically reduces load on the enterprises’ Web servers.

Dynamic Page Assembly

As Web content started to become more dynamic and personalized, Akamai teamed with companies like IBM and introduced sophisticated caching technology like Edge Side Includes (ESI), which allowed reusable fragments to be assembled closer to the users.

Edge-able Applications

With a full EdgeComputing platform, including WebSphere and Cloudscape, feature-rich applications can be built to run at the Edge, thereby offloading the Enterprise Computing platform.
EdgeComputing Roles

To better understand the computing model as it is presented by Edge Computing, it is important to understand the roles of the major participants within this environment.

Platform provider - IBM

The EdgeComputing architecture assumes two platforms. The first is the enterprise platform that is the full feature WebSphere software platform and all of its extensions. The second is the EdgeComputing platform. IBM provides this second platform to Akamai, the service provider, in the form of an embeddable WebSphere Application Server (WAS), which includes a subset of the J2EE programming model. Future versions are intended to include a run time version of Cloudscape and a lightweight Portal run time.

It is important to note that all code that runs on these platforms conforms to the J2EE standard. Standard and special deployment policies ensure the edge server optimally interacts with the enterprise server. Hence, it is really the deployment model that is changing, not the programming model. Certain restrictions apply on the EdgeComputing Platform as imposed by the service provider.

Service Provider - Akamai

Akamai’s globally distributed network of servers can now execute WebSphere applications. Akamai’s platform is able to understand traffic conditions on the Internet at any moment in time, and to choose the optimal location from which to process user requests. The Enterprise computing platform no longer has to process every single end user request, but instead processes as little as 1-5% of the overall demand on the application. (This factor will certainly vary depending on the data access patterns of the application.) Akamai absorbs the majority of the burden of processing the J2EE logic or ESI assembly, and can scale on demand to reliably and securely deliver Web applications without the entire burden or worry of capacity planning. In order to accomplish this increased off-load of the Enterprise platform, static content and applications are pushed out and stored/executed close to the end user on the Akamai server.

Akamai’s tiered distribution scheme employs a two level hierarchy across the 15,000 Akamai edge servers. There is a set of parent servers which serve the remainder of the Akamai edge servers. Therefore, requests to the enterprise come from only the parent servers and not from all edge servers. This scheme decreases the load on the enterprise by providing an additional layer of caching at the parent servers and thus fewer overall requests (and connections) are made to the enterprise.

Furthermore, Akamai’s SureRoute technology determines the fastest route between an Akamai edge server and the enterprise, ensuring optimal performance and guaranteed delivery to every end user, regardless of Internet conditions. SureRoute will route around Internet congestion problems as well as paths that are slow. There are two modes of operation: SureRoute for Failover and SureRoute for Performance. SureRoute for Failover is invoked when the edge server cannot communicate with the origin using the direct Internet route. This provides increased reliability for an enterprise. SureRoute for Performance is
invoked always to ensure that the optimal path is used between the Akamai edge servers and the origin. This mode ensures optimal performance as well as reliability.

**Enterprise IT Organization**

The Enterprise IT organization uses WebSphere Studio Application Developer with EdgeComputing plug-ins to develop a standard WebSphere/J2EE application in two components: an edge component and an enterprise component. The edge component is developed using Struts, Tiles, Servlets, JSPs, JDBC, JAX-RPC and JavaBeans. In order to run the edge component on Akamai’s EdgeComputing service, the IT organization does the following:

- establishes a service level agreement with Akamai for the application
- packages the edge component into a WAR file, which includes the supporting static content (HTML pages and fragments, GIFs/JPGs, etc.)
- deploys the edge component’s WAR file into Akamai’s EdgeComputing platform using a WebSphere Studio Application Developer wizard.

The enterprise component is developed to support the edge application. The enterprise uses WSDL to expose the enterprise application as a set of Web services. The Web services provide access to information from operational systems, e.g. order management, product catalogs, user management, and personalization data. The EdgeComputing applications use the standard Web services client model to access the enterprise information.
EdgeComputing Environment

The EdgeComputing environment operates across three logical tiers; the client platform, the edge computing platform, and the enterprise platform.

Client Platform

The client platform typically executes in a Web browser. The client platform can sometimes act as the ultimate “edge server”. For example, static images can be cached within the browser, avoiding trips to the edge and/or enterprise. Also, client-side scripts, like JavaScript, can run in the browser and perform tasks like form validation without calling upon upstream services. Hence, applications are encouraged to use JavaScript and DHTML to perform client-side scripting and validation.

EdgeComputing Platform

The EdgeComputing platform hosts the edge component of an edge-enabled application. The primary function of the Edge is to execute dynamic presentation logic using cached data. In particular, these are functions that are supported by the WebSphere Application Server J2EE Web container. The cached data can be resident in main memory where it can be accessed as objects, or within a local relational database where it can be queried.

Presentation features

Dynamic and personalized Web presentations are composed of components or fragments. These fragments are assembled to create the final page that is presented to the end user. On the EdgeComputing platform, JSPs, Servlets, JavaBeans, and the Struts/Tiles framework are used to generate and aggregate pages at the Edge. Portal technology is also under consideration for future inclusion.

It is also possible for pages to be generated and aggregated by retrieving previously generated fragments from cache. WebSphere’s Dynamic Caching Service allows fragment caching of JSP and servlet results, and can also generate ESI content that can be processed and cached by a downstream Akamai Edge Server.

Data interaction features

At the Edge, enterprise data is accessed by calling Web services deployed within the Enterprise. Whenever possible, the EdgeComputing platform can be an optimal place to cache data. For read-mostly operations, the Web services Client Cache, which is a function of the Dynamic Caching Service, can be used to cache the results of these Web services calls.

Cloudscape, a pure Java relational database, is also available at the Edge to host read-only data. Cloudscape version 5.x supports a “database in a jar” feature, which allows a database instance to be packaged in a Jar file and deployed along with an application. This can be a very effective way to cache product catalog or customization information. An enhancement to Cloudscape currently in the works, called the Cloudscape Query Cache, will allow JDBC-based applications to cache query results within the Edge. Subsequent queries can be executed against the locally cached data, while allowing updates to flow back to the Enterprise platform.

Enterprise Platform

The Enterprise Platform plays the role of the “mother-ship”. It is ultimately responsible for keeping the master view of the business data.
**Web Services**

As described in the EdgeComputing Platform section, Web services are used as the umbilical cord between the Edge and Enterprise. Hence, server-side business components will need to be wrapped as Web services if they are to be accessed by the edge. For example, the following are the types of services that were exposed when creating the EdgeComputing sample application:

- **Database** - Access to data-objects, which are ultimately implemented using a persistence technology such as JDBC or EJBs.

- **Personalization and Security** – Access to user registry and user personalization services. This class of service can provide access to the WebSphere Custom User Registry, JNDI lookups to an LDAP server, or the user management functions of WebSphere Portal Server.

- **Monitoring and Statistics** – Calls to aggregate application statistics from the edge. For example, applications can use WebSphere’s Performance Monitoring Interface (PMI).

- **Business process and rules** – Calls to trigger the execution of workflows and business rules.

It is important to note that in the Akamai network, edge servers come and go depending on load; therefore, there may be many more edge servers than enterprise servers under heavy-load conditions. Therefore, in order to optimize read-oriented Web services interactions within the Enterprise, Akamai edge server caching and/or the WebSphere Dynamic Caching Service can be used to cache responses to inbound Web services requests.

**Data Replication**

The Enterprise Platform is responsible for replicating data that is committed via Web services back to the edge platform. For example, the edge computing photo contest application allows users to submit photos to a photo information database and content repository that resides within the enterprise platform. However, rather than having all inquiries for photo metadata go back to the enterprise platform, this data can be replicated out to the EdgeComputing Platform.

Further research on replication technology that supports this configuration is being undertaken.
4. Developing EdgeComputing Applications

This section describes the programming model used to develop EdgeComputing applications. An alternative to edge computing for enhancing the performance of Web applications is Edge Side Includes (ESI). Please see Appendix A for more information about ESI. Application components are described along with APIs used to build them. The tools and deployment environment are also a key aspect of developing EdgeComputing applications. This section also describes how WSAD is used to deploy code to Akamai.

Application Components

Presentation Components

As previously discussed, the presentation component of an EdgeComputing application is an important aspect of the architecture. Typically, the presentation components of a J2EE application are developed using a standard Model-View-Controller (MVC) architecture. In this model, a controller servlet typically executes some business logic to generate a Model which is passed to a JSP to create the View. In order to improve developer productivity, a Web application framework can also be utilized. The most common model for using the EdgeComputing environment is to deploy the presentation components of your application into the Edge and utilize a distributed and cached access to the enterprise data via Web services.

Figure 1: A Typical EdgeComputing Application
**Struts**

Apache Struts is a supported, open source framework for building Web applications. The Struts framework provides the invisible underpinnings every professional Web application needs to survive. Struts helps you create an extensible development environment for your application, based on published standards and proven design patterns. The core of Struts is a flexible control layer based on standard technologies such as Java Servlets, JavaBeans, Resource Bundles, and Extensible Markup Language (XML).

Struts encourages application architectures based on the Model 2 approach, a variation of the classic MVC design paradigm. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. For the Model, Struts can interact with any standard data access technology, including Enterprise Java Beans (EJB) components, and JDBC. For the View, Struts works well with JavaServer Pages (JSP) files, XSLT, or other presentation systems. See [http://jakarta.apache.org/struts](http://jakarta.apache.org/struts) for more information about Struts and how it can be used as an application development framework.

![Figure 2: Web Application Using Struts](image)
**Tiles**

Typically, Web sites will want to use a common look and feel to layout Web pages within their Web applications. In order to reduce repetition and promote reuse, usually some sort of templating or layout mechanism is utilized. The Apache Tiles framework is one such mechanism. Tiles enables the Web application developer to create screen definitions which can be parameterized and inherited. These screen definitions can then be the targets of a Struts action or included directly via a Tiles tag. Screens are assembled from *tiles* which can be thought of as regions of content. The Tiles framework makes use of layout tiles to define common, reusable page and menu layouts. Layout tiles and content tiles can be nested recursively to create new aggregate content tiles. The combination of screen definitions, layout tiles, and content tiles provides a powerful framework to layout and reuse Web application presentation components.

![Figure 3: Example Page Formed From Tiles](image-url)
**Personalization**

Web Applications are becoming increasingly more dynamic and personalized. As an example, most commerce Web sites now provide targeted content based on the user’s previous purchases, interests, and/or browsing habits. In order to provide this kind of rich, personalized experience in an EdgeComputing application, personalization data must be available on the edge. Depending on quantity, type, and diversity of personalization data required by an application, several different methods can be employed to create personalized pages at the edge.

One method is to use Web services to retrieve the personalization data from the Enterprise and cache the data on the Edge Server. Since a given user will have affinity to a particular edge server, the trips to the back end to retrieve personalization data will only occur when:

- the user first contacts the edge server
- the user is switched to a different edge server due to changing network conditions
- the user's personalization data is flushed from the cache due to inactivity or capacity.

As an alternative to using a personalization cache, the user's personalization data can be stored in an HTTP Session at the edge. This method will eliminate trips to the enterprise when a user switches servers at the edge.

Another method to store personalization data is to use browser cookies. This method requires that the personalization data be rather small and discrete so that it can easily fit in a
cookie to be transmitted to and from the client browser. Another advantage of the cookie approach (if applicable) is that presentation caching can be applied to the personalized fragments using the cookie value as a component of the cache key.

Presentation in EdgeComputing

The EdgeComputing Sample Application uses Struts, Tiles, Servlets and JSPs as the primary presentation component technology. Future versions of this sample will also utilize Portlets. Portlets are not currently used because they are not supported in the Akamai EdgeComputing service.

A screen shot of the edge computing photo contest page shows some of the major screen elements:

The Apache Struts framework is used as the “glue” that binds the edge computing sample application together. All page requests to the edge computing application are implemented as Struts Actions. Sample actions for the edge computing application include Submit Photo, Register, Login, and View Gallery. The Action classes also make use of the Struts validation framework in order to validate form data both on the browser (using Javascript) and on the server. After the Struts action class is invoked and has processed the necessary business logic for a given page view, control is forwarded to a Tiles screen definition for presentation.

Struts Tiles is used to aggregate fragments of content onto a page. All of the application’s screen definitions are held in a single edgecomputing-defs.xml control file. Inheritance is used within the screen definitions to maintain a common look and feel and promote easy customization. All edge computing photo contest screens extend the base screen “edgecomputing.base”
For example, the main menu is defined by the following elements in the edgecomputing-
defs.xml. The definition edgecomputing.base defines the standard layout for an edge computing sample application page as including a header, menu, footer, body, and advertisements. As can be seen in the sample below, definition attributes can have several different values. They can be constant strings, names of other definitions, or a concrete JSP that is used to render content. The screen definition also has a JSP associated with it via the path attribute which is used to render the screen definition. This particular JSP can commonly be thought of as a layout manager for the page. The second kind of JSPs utilized by the edge computing application is content JSPs. These JSPs are used to render a particular fragment of content. Typically, a Struts action will place data into the request (e.g. a collection of image objects to be displayed) and then the content JSP is responsible for rendering this data into the presentation format. Struts includes several helpful tag libraries (logic, bean, html) which allow you to convert request attributes and other data in presentation format. For example, the struts tag logic:iterate provides a tag-based method for iterating through a collection of data. Wherever possible, embedded Java code within a JSP (also known as scriptlets) has been avoided. However, in some cases, Java code is called upon directly to perform tasks that cannot easily be done using tags. The powerful composition capability of the Tiles library makes it easy to define and reuse screen layouts to build a consistent look and feel for a Web site.

```xml
<definition name="edgecomputing.base" path="/layouts/edgecomputingLayout.jsp">
  <put name="title" value="EdgeComputing" />
  <put name="header" value="edgecomputing.header.standard" />
  <put name="menu" value="edgecomputing.menu.standard" />
  <put name="footer" value="/fraglets/footer.jsp" />
  <put name="advert" value="/adverts/vAdvert.jsp" />
</definition>

<definition name="edgecomputing.welcome" extends="edgecomputing.base">
  <put name="title" value="EdgeComputing - Welcome" />
  <put name="menu" value="edgecomputing.menu.welcome" />
  <put name="body" value="edgecomputing.welcome.body" />
</definition>
```
The screen depicted in Figure 6 contains a visual decomposition of the edge computing sample homepage into tiles.

**Figure 5 - Tiles decomposition of the EdgeComputing Home Page**
Business Logic Components

This section discusses best practices for developing and deploying business logic components for an EdgeComputing application. From a performance perspective, the preferred method of accessing business logic and its data depends to a large degree upon the data access pattern and update frequency of the data, thus different programming and deployment models are recommended. The diagram below provides an overview of the best practices (on bottom) which are recommended based upon these factors (on top).

![Diagram showing Business Logic Best Practices Overview]

If the data access pattern is read-only and the update frequency is low, the best practice is to use a **Local DB on the Edge**. If the data access pattern is read-write and the update frequency is relatively low, the use of **Web Services from edge to enterprise** is the best practice. As the update frequency of the data increases such that traffic increases between the edge servers and enterprise, the benefit of edge processing decreases and the use of an **Enterprise Only** application architecture is preferable. These application architectures are discussed in more detail in the following sections.

**Local DB on the Edge**

Consider the case in which the data access pattern is read-only and the data is seldom updated. For example, catalog data may be read millions of times daily while it is updated once per day or even less frequently. In this case, Cloudscape 5.1.10 supports three modes for read-only databases as described below.

1. Database files in local file system

   All the files from a Cloudscape database are put into a JAR archive in the local file system. Cloudscape can then access the database (read-only) using a specialized form of the database name. The JDBC URL is of the form:

   `jdbc:db2j:jar:(pathToArchive)databasePathWithinArchive`

   The `databaseName` property in the Cloudscape data source is of the form:

   `jar:(pathToArchive)databasePathWithinArchive`

   For example:

   `jar:(c:/local/dbs/sales.jar)/London/salesdb`

   Although this method is supported by Cloudscape, it is not be feasible in the Akamai network since it requires management of a portion of the application which is not packaged with the application.
2. Database files in application

As in the previous method, the database is contained in a JAR file. However, instead of separately placing the JAR file in the file system, the JAR is made available on the class path of the application. Cloudscape then accesses the database (read-only) using a specialized form of the database name. The JDBC URL is of the form:

```
jdbc:db2j:classpath:databasePathWithinArchive
```

The `databaseName` property in the Cloudscape data source is of the form:

```
classpath:databasePathWithinArchive
```

For example:

```
classpath:/London/salesdb
```

The database JAR is packaged with the application in an EAR file, placing it in the lib folder of the WAR file (where the application’s code JAR files are placed). The application and data are then a single entity. The application’s data source then has a fixed configuration corresponding to the location of the database within the JAR file.

Akamai then:

- distributes the JDBC application and data as an EAR file to its edge servers
- provides the configuration to point to the JAR file for the application’s data source.

3. Database files served from enterprise

A read-only copy of a database’s files is served via URL from the application server in the enterprise. Cloudscape can then access the database (read-only) using a specialized form of the database name. The JDBC URL is of the form:

```
jdbc:db2j:http_url_to_data
```

The `databaseName` property in the Cloudscape data source is of the form:

```
http_url_to_data
```

For example:

```
https://London.edgecomputing.com/published/salesdb
```

Cloudscape continues to execute as a normal database (but read only). Data pages are maintained (only) in the page cache, in memory. When a page is needed that is not in the buffer cache, an HTTP request is made to access that page. The requests are of the form:

```
http_url_to_data/path_to_required_file
```

For example:

```
https://london.edgecomputing.com/published/salesdb/seg0/c1a32.dat
```

The Web server can serve files in either of two ways:
1. As regular, complete files:
   In this case Cloudscape will skip the data until it reaches the page it requires
   and then copy that page into the page cache. This can be inefficient. For
   example, in order to read the 10,000th page; the previous 9,999 pages are
   transmitted across the network, since in Cloudscape the skip happens at the
   client side. However, this is simple to set up since it is standard file
   browsing.

2. As individual pages, via a servlet:
   In its HTTP requests, Cloudscape provides HTTP headers that indicate the
   offset and length required from the file. The servlet then opens the file,
   positions to the required offset, reads the required length, and returns the
   contents in the HTTP reply. The servlet also provides the ability for logical
   database names, e.g. mapping published/salesdb to
   e:/databases/London/2001salesdb

For Cloudscape, the setup of these two modes is identical. It always sends the
HTTP headers and if the reply contains the matching headers, Cloudscape
assumes that the page mode is being used; otherwise, the complete file mode is
assumed. Thus the configuration is limited to the central site. Note that the
database files being served by the Web server cannot be opened by a
Cloudscape database at the same time.

The distribution of the application then matches the class path model, where the
application and the data source configuration are fixed.

Relational database access has been turned into HTTP data requests of static
data. Akamai provides the ability to distribute such data around its network
transparently and with ease. Thus a read-only copy of a database from an
enterprise site can now be cached around the Akamai network. In addition, the
database may exist at various Akamai servers in a partial state. For example, if
only 10% of the database is being accessed frequently, then only that 10% would
be cached at the Akamai servers. Furthermore, the portion of the database which
is cached may differ based upon access patterns in various geographies.

How can the database content be updated on the edge once it has been cached
by Cloudscape? Consider this sequence of events for a JSP/Servlet application
request:

1. start servlet request
2. find data source
3. get connection
4. execute query
5. return results.

There are two opportunities for indirection at steps 2 and 3 which could allow
database content to be effectively updated on the edge.

- At step 2, Akamai edge server can replace/update the data source
  registered at jdbc/londonsales from
  http://london.edgecomputing.com/salesdb20030313
  to a newer read-only copy
  http://london.edgecomputing.com/salesdb20030314. Since these are
different databases as seen by Cloudscape there would be a
requirement for Akamai to shutdown the older version. Or for
Cloudscape to automatically shutdown a database when it becomes idle.
• At step 3, the data source and thus the HTTP URL remain the same, say http://london.edgecomputing.com/salesdb, but when new content is required, Akamai changes the mapping of this fixed URL, as before, from http://london.edgecomputing.com/salesdb20030313 to a newer read-only copy http://london.edgecomputing.com/salesdb20030314. Since this is the same database from Cloudscape’s view, no shutdown is required, but once such a change occurs, Cloudscape must discard everything in its cache. This feature is expected to be provided in a future version of WebSphere.

Given the ability to update read-only databases on the EdgeComputing platform using one of the methods just described, a solution for read-only database caching at the edge is shown below.
The enterprise is shown above the heavy dotted line. DB2 Everyplace Synchronization Server is used to collect changes from the back-end DB2. A Cloudscape client (to DB2e Sync Server) database is used to format the data for use by the edge Cloudscape databases. The copy of the database files to the WebSphere Application Server data server could be smart (only changed files) or dumb (complete copy). The copy of the files would need to interact with Akamai so that requests were sent to the new copy. This does mean every M minutes, the edge Cloudscape databases would start from scratch, much like a total cache invalidation of a set of data. We could skip DB2 Everyplace Sync Server, and just select the complete DB2 database into Cloudscape every N minutes, but this could be heavy load on back-end database. Note that this is a model in which the edge client can tolerate out of date data, governed by the values of M and N.
This section has discussed several methods of accessing a database locally from the edge. Each of the methods described above assumes that the programming model for accessing a database is JDBC. In the future, WebSphere Data Objects (WDO) will be considered.

**Web Services From Edge To Enterprise**

If the data access pattern is not read-only, the recommended model also changes. For example, if the update frequency of a database is once every five minutes, it would be undesirable to use the local database method, which would require updating the application (along with the database) every five minutes. The process of updating an application in Akamai's network may itself take a few minutes.

Therefore, the business logic and data should not reside on the edge server. Instead of a single-tier model, a two-tier model is preferable in which the enterprise server hosts the business logic and data, and access from the edge is performed through Web service calls. Client-side Web service caching is utilized to decrease the edge-to-enterprise communication and improve overall performance, response time, and scalability.

In the edge computing photo contest application, Web services were used extensively to access business logic components. The "user-interface" part of the application is deployed in the Akamai network. JAX-RPC is used to call remote Web services over SOAP/HTTP. The remote Web services are deployed on an Enterprise Server. The Web services are implemented as simple JavaBeans that use JDBC to transact with DB2. Web services client caching is not currently being used (but is expected to be included in a future version of WebSphere). Instead, client-side caching is implemented as a dynamic proxy using the `java.lang.reflect.Proxy` class. The business logic components implemented as Web services are:

- **User management** – provides user registration, authentication, and user profile services;
- **Content management** – provides content management services such as querying and updating of photos;
- **Statistics** – provides statistics associated with the state of the application (e.g. active servers, active users, etc.)

For example, the following sample sections show portions of the WSDL pertaining to the content management services.

The first WSDL portion describes the composition of a "Content" object.

```xml
<complexType name="Content">
  <sequence>
    <element name="url" nillable="true" type="xsd:string"/>
    <element name="id" type="xsd:int"/>
    <element name="width" type="xsd:int"/>
    <element name="creator" nillable="true" type="xsd:string"/>
    <element name="verified" type="xsd:boolean"/>
    <element name="rating" type="xsd:double"/>
    <element name="creationDate" nillable="true" type="xsd:dateTime"/>
    <element name="title" nillable="true" type="xsd:string"/>
    <element name="description" nillable="true" type="xsd:string"/>
    <element name="height" type="xsd:int"/>
  </sequence>
</complexType>
```

The next WSDL portion describes the message used to cast a vote for a specified piece of content.

```xml
<wsdl:message name="voteForContentRequest">
  <wsdl:part name="content" type="intf:Content"/>
  <wsdl:part name="rating" type="xsd:int"/>
</wsdl:message>
```
Web services and passing of Java objects

Exposing certain interfaces as Web services can present interesting challenges. One issue pertains to the manner in which Java objects are passed over a Web services call. In particular, the WSDL for a Web service defines the precise object types which flow between a Web service client and server. Since the WSDL is statically generated from an interface definition, the object types which aremarshaled are the declared types on the methods of the interface, not the actual instance types. For example, suppose that the **MyWebService** interface is used to define a Web service.

```java
public interface MyWebService {
    Node[] getChildrenOfRoot();
}
```

```java
public abstract class Node {...}
public class Leaf extends Node {...}
public class Tree extends Node {...}
```

Although the **getChildrenOfRoot** method is declared to return an array of Node objects, the client expects the actual instances to be either **Leaf** or **Tree** objects. Web services, however, marshals the declared type, not the actual instance types; thus, an error would occur when the Web services client attempts to create an instance of Node.

A similar problem occurs if the declared return type or parameter is a Java interface. For example, consider the following code fragment.

```java
public interface MyWebService {
    Node[] getChildrenOfRoot();
}
```

```java
public interface class Node {...}
public class Leaf implements Node {...}
public class Tree implements Node {...}
```

Since the **getChildrenOfRoot** method is now declared to return an array of Java interfaces, an error will occur when the Web services client attempts to create an instance of the **Node** interface.

Various methods of solving this problem are described below.

- **Wrapper method**

  In the wrapper method, a class is created which wraps an instance of each possible class which can be passed as an argument or return value from a Web services call. The wrapper class is then used in the Web service interface definition and thus in the WSDL. For example, **MyWebService** is changed as follows:

```java
public interface MyWebService {
    NodeWrapper[] getChildrenOfRoot();
}
```

```java
public abstract class Node {...}
public class Leaf extends Node {...}
public class Tree extends Node {...}
public class NodeWrapper {
```

...
private Leaf leaf = null;
private Tree tree = null;

// Null constructor is required by Web services
public NodeWrapper() {
}

// Constructor to wrapper Leaf
public NodeWrapper(Leaf leaf) {
    this.leaf = leaf;
}

// Constructor to wrapper Tree
public NodeWrapper(Tree tree) {
    this.tree = tree;
}

// Retrieve actual instance value
public Node getNode() {
    if (tree != null) return tree;
    else return leaf;
}
}

Notice that NodeWrapper[] has replaced Node[] as the return type of the
getChildrenOfRoot method. NodeWrapper contains an instance of Leaf and Tree,
one of which is presumed to be null. After the Web service call completes, the edge
application calls getNode on the NodeWrapper to retrieve the actual Tree or Leaf
instance and then casts it appropriately.

- Java serialization method

A second method of overcoming the problem described above is the serialization
method. In this method, the argument and/or return type are declared as a byte array.
The actual argument or return instance is serialized and therefore passed as a byte
array. One disadvantage to this approach is that the Web service is now Java-specific;
however, this is typically not a problem in an EdgeComputing application where the only
Web service client is an application server running on the edge.

The Java serialization method is used in a service provided by the Edge Computing
Framework (ECF). In order to easily allow any object with a defining interface to be
deployed as a Web service, ECF provides a generic RMI Web service which allows an
arbitrary interface and instance to be registered on the enterprise and accessed from the
dge. This service can be used by developers to split an application between edge and
enterprise with no interface changes, no knowledge of Web services, and a minimum
amount of effort.

For example, the sample application’s statistics service uses this service in order to
communicate between the edge and enterprise. The statistics service is defined by the
StatMgr interface while the StatMgrFactory class is used to return an instance of the
service. The enterprise application registers this service using the following code
fragment:

```java
WsService service =
    new WsService(StatMgr.class, StatMgrFactory.getInstance());
WsServerFactory.getInstance().register (service);
```

The edge application accesses this Web service using the following single line of code:

```java
StatMgr statMgr = (StatMgr) WsClient.getProxy (StatMgr.class);
```
When WsClient.getProxy is called, a dynamic proxy is generated using java.lang.reflect.Proxy. This dynamic proxy delegates to an InvocationHandler to marshall and demarshall the parameters and results of the method invocation into the generic java serialization format. On both the server and client sides of the interfaces, the service interface’s classloader is used to demarshall parameters.

Additional register and getProxy methods are also available which allow a service to be registered and accessed using an arbitrary name. Therefore, a single interface can be registered by specifying different names, thus allowing multiple concurrent instances of a service.

WsService, WsServerFactory, and WsClient are all defined in the package: com.ibm.websphere.ecf.webservices.

- XML serialization method

Another more portable option to the Java byte serialization presented earlier is to use the same overall system presented in “Java Serialization method” but instead of Java serialization, use an XML serializer to serialize request and response data for the WebServices call. This method still requires custom stubs, but the data is much easier to parse and generate in other language environments than are serialized Java objects.

Web services and Java exceptions

Another issue which may be encountered when creating a Web service pertains to Java exceptions. If an exception is thrown by one of the methods implementing the Web service, a Web services fault code is passed back to the client rather than the actual Java exception. Therefore, the exception which is thrown by the Web service’s client often does not contain adequate information to determine the original exception, thus making it very difficult to determine the exact cause of a problem and impossible to take some action based upon the type of exception which originally occurred.

For example, suppose the following interface is to be exposed as a Web service:

```java
public interface UserService {
    public java.util.Properties login(String uid, String pwd)
        throws InvalidUidException, InvalidPwdException;
}
```

The login method authenticates a user given a user ID (uid) and password (pwd). If the user ID and password are valid, a set of properties is returned containing user profile information. If the user ID is invalid, an InvalidUidException exception is thrown. If the password is invalid, an InvalidPwdException exception is thrown. If the user ID is invalid, an InvalidUidException exception is thrown.

If the UserService interface is used when generating the WSDL for the Web service, there is no way for a Web services client to distinguish between various login errors: InvalidUidException, InvalidPwdException, or another Web services exception.

One method of solving this problem is to create a class to wrapper the return value and all possible exceptions. This is similar to the wrapper method described previously. All exceptions must be caught by the Web service implementation and returned in the wrapper class. For example, the UserService interface would be as follows:

```java
public interface UserService {
    public UserServiceLoginWrapper login(String uid, String pwd);
}
```

```java
public class UserServiceLoginWrapper {
    private java.util.Properties props = null;
    private InvalidUidException uidException = null;
    private InvalidPwdException pwdException = null;

    // Null constructor required by Web services
```
public UserServiceLoginWrapper() {}

// Construct normal return value
public UserServiceLoginWrapper(java.util.Properties props) {
    this.props = props;
}

// Construct various exception return values
public UserServiceLoginWrapper(InvalidUidException exc) {
    this.uidException = exc;
}
public UserServiceLoginWrapper(InvalidPwdException exc) {
    this.pwdException = exc;
}

// This method is called on the client-side after the Web services client
// has returned an instance of UserServiceLoginWrapper from login
public java.util.Properties extractReturn() throws InvalidUidException, InvalidPwdException {
    if (uidException != null) {
        throw uidException;
    } else if (pwdException != null) {
        throw pwdException;
    } else {
        return props;
    }
}

The Web services interface no longer throws an application exception. Instead, any
exception is caught, wrappered, and returned in the return value. Once the Web services
client has completed the login method, the extractReturn method is called on the return
value to extract the real return value. If an exception occurred at the server, the same
exception will then be thrown by the extractReturn method at the client.

As was discussed earlier, the Edge Computing Framework (ECF) provides a generic Java
RMI Web service which allows an arbitrary Java interface to be registered on the enterprise
and accessed from the edge. This generic service uses a method similar to that described
above in order to pass Java exceptions back to the client.

Monitoring statistics in an EdgeComputing application

As was mentioned previously, the edge computing photo contest application contains a
statistics service. Because an EdgeComputing application is split between edge and
enterprise, it was necessary to provide a richer statistics infrastructure. The hub of this
infrastructure is the statistics service defined by the following interface. This service is part of
the ECF.

public interface StatMgr {
    public long addCounters(CounterGroup group) throws Exception;
    public CounterGroup getCounters (String name) throws Exception;
}

An edge server periodically sends its statistics to the enterprise as a CounterGroup by calling
the addCounters method. The enterprise returns the length of time that the edge server
should wait before sending it’s next update (i.e. before calling addCounters again). If the
edge server does not send an update within the allotted time (plus a grace interval), the
enterprise removes that edge server from it’s list of active edge servers.

When the number of edge servers is large, the enterprise could be overwhelmed if each edge
server sends updates too often. In order to avoid this scenario, the length of time between
updates as returned by addCounters is a multiple of the current number of active edge servers.
This causes the number of total addCounters calls made to the enterprise to be
relatively constant as the number of active edge servers varies.
In order to retrieve statistics that are associated with the enterprise server itself as well as with all active edge servers, the edge server calls the `getCounters` method. The enterprise server associates a name (e.g. "basic", "admin") with a `CounterGroup`, which is used by the edge server when requesting statistics. The enterprise server finds the appropriate `CounterGroup` based upon the name, updates it's counters, and returns them to the edge server. The user interface logic running on the edge server then displays these statistics.

**Security**

This section discusses security as it relates to an edge-enabled application.

First, consider security at the enterprise. In a future release of WebSphere, Tivoli ID Manager (TIM) and WebSphere Member Manager (WMM) merge to become WebSphere Identity Manager (WIM). WIM will also implement the `com.ibm.websphere.security.UserRegistry` interface, which allows it to be used by the WebSphere run time as its custom user registry. Therefore, WIM may be used both by the enterprise run time as well as by the enterprise application for authentication services. Also in a future release of WebSphere, the WebSphere security run time will allow a Custom User Registry to be set on a per-application basis rather than globally.

Next, consider security at the edge. WIM (or a subset of WIM) will be exposed as a Web service. The Web service client will be used both by the edge server run time as the custom user registry, as well as by the edge application. WS-Security (and/or optionally SSL) may be used to provide authentication of the edge server to the enterprise server, and vice-versa.

The following diagram provides an overview of security for an EdgeComputing application using WIM and Web services to communicate between the edge and the enterprise.

![Security Diagram](image)

**HTTP Sessions**

EdgeComputing supports the Java Servlet Specification notion of HttpSession. Edge applications can still use HttpSession to track user actions and to keep state. There is no need to change your application to use an Akamai specific API or another state persistence mechanism. The Akamai session replication system is integrated behind the scenes in concert with the application server.

It is recommended, however, that the session object itself be kept small (as measured by serialized form). Alternatively, session states can still be maintained on the client, using cookies, URL rewriting, hidden HTML form fields, or other means.

**EdgeSession Guidelines:**
• Each session object cannot be larger than 20KB in serialized form. Serializing session data can require significant storage and use a great deal of overhead, with the potential of severely affecting performance. The overhead required grows with the size of the data objects, so it is recommended not to store large objects in a session.
• HttpSessionListener should not be used.
• Infinite session expiration time-out is not supported.
• Applications must handle their own locking/concurrency issues.
• Objects must be “distributable.” The session manager enforces the requirement that Web applications must adhere to the “distributable” conventions described in the Servlet Specification. A Web application must be marked as “distributable” in its deployment descriptor, and any object to be stored in session must implement the Serializable interface java.io.Serializable; otherwise, an IllegalArgumentException will result.
• Session tracking cannot be preserved across edge and origin requests for the same hostname and context path.

JDBC over HTTP Proxy

Akamai has developed a Type 3 JDBC driver that consists of a component that runs on the edge in the EdgeComputing service, and a server component, a server-based Web application, deployed on the origin in the WebSphere Web container. The edge and origin components communicate via an HTTP tunnel. The use of HTTPS protocol is supported. The Type 3 driver solution requires no holes in the origin firewalls and can be deployed in your existing Web site infrastructure. This driver also benefits and complements other Akamai solutions (for example, SureRoute), because the underlying transport is HTTP. Finally, the JDBC implementation allows for result set caching to be done, both at the origin interceptor as well as at the edge.
Deployment to Akamai

Application Deployment to Akamai’s EdgeComputing Platform

Akamai’s customers deploy Web applications onto the EdgeComputing Platform via a “Publish” Web service. Invocation of this Web service has been seamlessly integrated into WebSphere Studio as an “export wizard.” Thus, WebSphere developers may deploy their edge applications to Akamai directly from WebSphere Studio.

Project Support for Akamai’s EdgeComputing Platform

To ensure security and reliability of the EdgeComputing Platform, Akamai does not allow applications to invoke APIs that may cause deviant behavior (e.g. applications may not create new Threads). Akamai inspects each published Web application to confirm that it does not invoke restricted APIs. Although the inspection protects the EdgeComputing Platform, it may lead to unexpected, last-minute changes for developers.

WebSphere Studio will provide an improved development experience through a new “project type” for edgeable applications. By selecting the new “EdgeComputing Platform” project type, the developer's interface to WebSphere Studio will be tailored for edge applications. Most importantly, this means that WebSphere Studio will provide guidance, at development time, to ensure that applications will execute properly on the EdgeComputing Platform.

Unit test environment – (simulating split-tier environment)

WebSphere Studio hosts an embedded WebSphere Application Server for integrated development, testing, and debugging of traditional Web and Enterprise applications. Work is underway to leverage the embedded WebSphere Application Server for EdgeComputing-style applications, thus providing WebSphere developers with the same integrated experience for the split-tier environment. This new work involves isolated hosting of the edge and enterprise application components, where the edge component executes in an environment that's configured to emulate the Akamai EdgeComputing Platform.
5. EdgeComputing Results from the Lab

The Edge Computing Project team conducted a set of performance and scale experiments using the Photo Contest application. The goal of the experiments was to determine the scaling and response time characteristics of the Photo Contest application as pieces of the application were off-loaded to the EdgeComputing tier.

**Topology**

There are 3 logical tiers used during the Edge Computing Project performance experiments. The Client tier runs the simulator tool which is used to drive client load. The edge computing tier is running the Edge-able components of the edge computing photo contest, which are predominantly the dynamic presentation components (e.g. Servlets/JSPs). The Enterprise tier is running the photo contest Web services, implementing the User and Content data objects and actions, which are accessible via Web services interfaces.

One of the performance experiments used NIST Net to simulate network delay. NIST Net is a network emulation package that runs on Linux. NIST Net allows a single Linux PC, set up as a router, to emulate a wide variety of network conditions.

**Test Methodology**

**Edge Computing Scenario Servlet**

The edge computing sample application’s Scenario Servlet uses “server-side” logic to drive the EdgeComputing application in a way that represents how an end user might interact with the application. To use the Scenario Servlet all one has to do is repeatedly refresh the corresponding URL. With each click of the refresh button, the Scenario Servlet selects the next logical action that might occur at the site, using a configurable probability. Probabilities of actions can be configured to control read to write ratios. For example, by default the application is configured to browse photos (i.e., read-operations) significantly more than either voting or submitting new images (i.e., write-operations). An HTTP cookie is used to keep track of the progression of sequences for a given interaction session. This “server-side” scripting technique significantly reduces the functional complexity of the load driver tool required. The Edge Computing Project team used an IBM internal tool called Web Performance Toolkit (WPT) to simulate client load to the edge computing sample application’s Scenario Servlet. Driving a constant client load to this Servlet, over a large number of requests (e.g. 50,000) will yield a single, repeatable throughput and response time result. For example, if 50,000 requests are sent between 10 simulated clients – the WPT tool would report an average response time (e.g., 2 seconds per page view) and an average throughput (e.g., 120 page views per second).
Measuring Performance

The performance experiments on the edge computing sample application attempt to illustrate how the application scales as the Edge-able portions of the application are distributed to Edge Servers. The experiments are also conducted with and without network delay conditions.

Results and Analysis

Scaling Factor

The following chart illustrates that near-linear scaling is achieved as new Edge servers are brought on line. The scaling occurs because the Edge servers are able to satisfy the majority of the edge computing photo contest requests without going back to the Enterprise tier. Analysis shows that the Web services (JAX-RPC client) cache, which uses the WebSphere Dynamic Caching Services, was very effective in caching the results of the User Preferences and Content query calls, preventing calls to the Enterprise tier. During these experiments the Edge servers were all running at high CPU utilization, while the Enterprise server was running less than 25% utilized with the four Edge servers driving it.

Response Time

During these experiments, client load was incrementally increased by 100 users. (The clients were all running without think-time.) With each increment, new Edge Servers were brought on-line to handle the spike in traffic. The chart to the right illustrates that the response time remains constant as the number of clients increase. This was the expected behavior, given the Enterprise server was never saturated.

Network Delay

An Edge-able application provides customers with near-constant response time. Other the other hand, the response time of a traditional application is proportional to the client's proximity to the enterprise. The following results simulate varying network delay conditions. With each measurement, requests from the WPT tool pass to Edge servers without simulated delay. If one of those requests need data from the Enterprise server, one of 3 levels of delay are applied on the hop between the Edge and Enterprise servers. The first data point, labeled Local in the corresponding chart, shows less than a 1% impact on response time. In the case
labeled **East-West Coast**, which applies a 75 ms network delay, as would be experienced when accessing the Internet between the East and West coasts, the response time is more than tripled. The last case, labeled **US-Japan**, which applies a 200 ms delay, the response time increases over 7.5 times.
6. Appendix A

Edge Side Includes (ESI)

While caching is an alternative to EdgeComputing for improving the performance of Web content, most Web applications contain highly dynamic and personalized content that cannot benefit much from traditional caching technologies. However, by modeling these pages as collections of reusable “fragments” (similar to Tiles), each with its own caching policy, a complete Web page can be assembled at the edge with the full benefit of caching, but without sacrificing personalization or dynamic content. For example, the Photo Application’s pages are both dynamic and personalized, displaying photos and photo categories unique to each user. There is little point to caching these pages with traditional caching since there is little to no reuse. By breaking the page up into fragments, however, we can separate the content into that which is generic and can be cached (such as the welcome and overview message displayed to every user) and that which is unique to a user (e.g. the list of photos he has submitted). In order to cache a fragment, its content must be a function of the elements available in the request from the browser (e.g. request parameters, cookies, headers, URI). These fragments (whether cached or not) can then be assembled at the edge of the network, in this case the Akamai network (Figure 5). The fragment assembly model improves performance over traditional whole-page caching because there is more reuse of cached content and because the edge of the network absorbs the burden of assembling the fragments.

![Figure 9 - A page assembled from ESI fragments](image-url)
Edge Side Includes (ESI) is a W3C Acknowledged Submission (http://www.w3.org/Submission/2001/09) that supports this fragment assembly model by defining three main elements:


2. Edge Architecture Specification: An extension of the traditional web/caching infrastructure with HTTP surrogates that perform assembly of Web content from fragments.


WebSphere Application Server provides two methods for exploiting ESI functionality. The first is WebSphere APIs for Edge Side Includes (WESI) (http://www.alphaworks.ibm.com/tech/wesi). WESI is a set of JSP tags and Java APIs that provide J2EE developers direct access to ESI functionality. While providing a convenient and easy-to-use interface WESI does not, however, expose complex rules for caching and invalidation. For more advanced functionality, the recommended method is to use WebSphere Application Server V5’s Dynamic Caching facility with Edge Side Include support. Instead of explicit markup of Web content, WebSphere V5’s Dynamic Caching facility depends on an XML configuration file, cachespec.xml, for the definition of fragments, cache policy, and invalidation information. If the application is being used with the Akamai network, automated facilities will soon exist to propagate this information to the Akamai network as well.

In order to compare the functionality, performance and development effort of EdgeComputing applications and those using ESI, we created an equivalent ESI version of the existing EdgeComputing Photo Contest application. Because we needed access to more advanced cache policies, we used WebSphere’s Dynamic Caching facility with ESI support. We found that there were two main types of fragments: those that were the same for every user, and those whose content was a function of one or more elements available on the request from the browser (e.g. cookies, query parameters, request headers, etc.). In some cases, we needed to move information from request attributes or sessions in order to make it available in the request from the browser.

In order to determine how to break up the existing content into fragments, we performed a "glass-level" analysis on current photo application pages, examining them as a user would. We identified sections of each page that appeared to be the same for each user, regardless of whether they were logged in or not. Other sections were shared by all logged-in users or all users who had not logged-in. Still others were dependent on one or more pieces of information – such as the section at the top of each page that displayed the user’s name once they had logged in. First we concentrated on what that information was (e.g. the user’s name), rather than how it was stored (in session, cookie, query parameter, etc.). Once we had roughly identified “fragments” in this manner, we then looked at the logical structure of the application code to see how well the identified fragments matched the existing division of code. For the most part, the code was already a good fit to the fragments we had identified. The process (Figure 6) is discussed in more detail below.
The first and simplest category of fragments contained those that could be cached only based on their URI -- that is, the content did not depend on any other information from the browser request (e.g. cookies or query parameters). An example of this type of fragment was the welcome message displayed to all visitors of the page. Although this message might vary over time, it was otherwise the same for all users. If we used traditional methods to cache these fragments, we might well end up with more entries in the cache than necessary since there is usually a one-to-one correspondence between the URL and cache entry. In the J2EE programming model, query parameters from the parent page are also passed to any included child fragments as part of the child fragment's URL. So if the parent page had any query parameters, this could result in different cache entries for the included fragment, even if these query parameters had no effect on the fragment's content. In order to avoid this issue, IBM research developed the concept of a cache id, which specifies which information from the request is relevant when creating an entry in the cache. By using cachespec.xml, users can specify cache ids for each piece of content. For example, the entry in cachespec.xml for welcome.jsp is:

```
<cache-entry>
  <class>servlet</class>
  <name>/fraglets/welcome.jsp</name>
  <cache-id>
    <component id="" type="pathinfo">
      <required>false</required>
    </component>
  </cache-id>
  <property name="EdgeCacheable">true</property>
</cache-entry>
```

This specifies that only the path portion of the URL should be used as a cache id (Figure 7).
The remaining fragments were dependent on other information that needed to be considered when creating their cache entries (Figure 8). This information needed to be available to the edge when it made caching decisions. This was not a problem when that information was already contained in the request from the browser (e.g., in the form of query parameters or cookies), but if it was held on the server, in session, for instance, it was not available to the edge. We needed to locate this information and place it in the request made by the browser so that it would be available to the edge. Query parameters, cookies, or request headers were all possible places to contain this information. Once we identified all the needed information, we found that it was all related to user characteristics. Because of this, we chose to set additional cookies with user info in addition to the already existing login cookie. These cookies contained the user’s name, whether they had submitted any content, and what their favorite categories were. None of this information was confidential, but if it had been we could have encrypted the cookie values.
In some cases, it was possible to exploit more reuse of fragments with ESI variables. ESI variables allow personalized information to be added at the time of fragment assembly, therefore increasing cache reuse and improving response time because of fewer cache misses. For instance, the logon fragment displays a user's name once they have logged in. If we had defined its cache id to depend on the value of the cookie containing the user's name, then each user would have had a separate entry in the cache for the logon fragment. Instead, we replaced the code that displayed the user's name with the following markup:

```
<esi:vars>${HTTP_COOKIE{displayName}}</esi:vars>
```

As a result, we only needed to keep one copy of the logon fragment in cache and the name of the user was added only at the time of assembly. We used the same technique on other fragments where the user's name was displayed.

![Diagram](image.png)

**Figure 13** – logon.jsp has only one entry in the Akamai Edge Server Cache, but the user’s name is substituted for ${HTTP_COOKIE{displayName}} when the page is assembled

We were also able to use ESI variables along with some Akamai-specific extensions to ESI in order to support randomization at the edge. This is an example of how advanced behavior, that may seem to require an application server at the edge, can be accomplished using ESI. The Photo Contest application contained two ad fragments, hAdvert.jsp and vAdvert.jsp, each of which displayed one of a set number of pre-determined advertisements at random. The original JSP code defined a two-dimensional array, where each element was an image name and link, and then chose at random from that array. For this code to run at the edge, we first created a list of lists in place of the array, using the Akamai assign tag, an extension to ESI 1.0.

```
<esi:assign name="randomList">
  ['tco_anim2.gif', 'http://www-1.ibm.com/servers/solutions/'],
</esi:assign>
```

The next step was to choose an element from this list at random. First, the random index was identified and assigned using:

```
<esi:assign name="index" value="$rand()%$len($randomList)" />
```
and then the actual element is retrieved from the list and assigned to the variable “element”:

<esi:assign name="element" value="${randomList{$(index)}}" />

Using the ESI assign tag ensured that the element would not be chosen from the list until request time. Once the image and link for the advertisement were chosen, they were then used to build the image tag.

<A href="${esi:vars}${element{1}}"></A>

<IMG border="0" alt="Ads from IBM and Akamai" src="/adverts/${esi:vars}${element{0}}"></A>

When specifying cache rules in cachespec.xml, it is also possible to specify which actions (accessing a servlet, JSP, or command) trigger an invalidation of content from WebSphere Application Server’s Dynamic Cache. For instance, if the logout.jsp page is accessed by user1, then all of user1’s content in the cache can be invalidated automatically. Cache Invalidation for WebSphere Application Server to Akamai EdgeSuite, available soon on alphaWorks, allows these same triggers to invalidate content on the Akamai network as well, automatically. Cache Invalidation for WebSphere Application Server to Akamai EdgeSuite can be installed through the WebSphere admin console and does not require any changes to existing applications or to the Dynamic Cache configuration file. Once it is installed, invalidations flow seamlessly from WebSphere Application Server to the Akamai network.

The purpose of writing an ESI version of the EdgeComputing Photo Contest application was to compare the two in terms of functionality, performance, and development effort. We were able to replicate almost all of the functionality of the EdgeComputing application using ESI with the exception of one feature. In the EdgeComputing application, a random photo was displayed on the main page. If a user were logged in, the random photo was from one of their favorite categories. Because the list of photos was not available at the edge, we were not able to select a random one and still cache the page at the edge. In order for caching with ESI to be effective, the appropriate information needs to be available at the edge. We were, however, able to duplicate all the other functionality of the EdgeComputing application. In terms of performance, we found that ESI generally reduces response time if all fragments on a page could be cached. If more than one fragment on a page cannot be cached, the time it takes to retrieve those fragments from the enterprise in addition to assembly time negates the value of having the other fragments cached at the edge. One case where EdgeComputing can improve performance over ESI is when information received at the edge does not need to be transmitted to the enterprise immediately. For example, when a vote is submitted in the Photo Contest application, it does not need to be counted immediately. With EdgeComputing, these votes (or other information), can be batched and sent at certain intervals rather than with every request. ESI does not provide this kind of control, since any non-cacheable request must immediately flow back to the enterprise. The last category of comparison is development effort. If an application is already using WebSphere’s Dynamic Caching facility, then extending it to use ESI as well should involve minimal development effort. The only change involved would be if the information needed to cache fragments was not available in the request and would need to be exposed.