RC22376 (W0203-057) March 18, 2002
Computer Science

IBM Research Report

Edge-Server Caching of Enterprise JavaBeans: Architecture and Implementation

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Edge-Server Caching of Enterprise JavaBeans: Architecture and Implementation

Avraham Leff, James T. Rayfield
avraham@us.ibm.com, jtray@us.ibm.com
914-784-{6381, 7559}
Fax: 914-784-6040
IBM T.J.Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

1 Abstract

We discuss key issues involved in the caching of Enterprise JavaBeans (EJBs) on an Edge-Server, and then present the architecture and algorithms that we have used to build an edge-server that solves these issues. The ability to cache EJBs is important because current edge-servers permit only caching of data such as static HTML pages. Applications that access stateful data must therefore access the back-end server frequently, so that performance is affected by network latency and the existence of system “hot spots”. With cached EJBs, edge-servers can run transactional applications that access dynamically changing data using a well known application programming interface. The behavior of the EJB runtime is functionally indistinguishable from that of a standard J2EE server, and thus the application behavior is unchanged. We also show that, with relatively little effort, non-cached applications can be transparently transformed into edge-server applications.

2 Introduction

2.1 Location Independence and Dependence

Current component technologies (e.g., [2], [8], [3]) offer the benefits of *location independence* to application developers, at least as far as the programming APIs are concerned. Developers can completely ignore issues about where the application and its components will run. At runtime, the component middleware transparently instantiates the required components. Unfortunately, since the instantiated components are bound to only one computer (or “server group”), this location independence is actually an illusion from a performance perspective. Typically, a J2EE application is deployed such that the client interacts with the application through a web-browser; the browser interacts with a servlet [9]; and the servlet invokes methods on Enterprise JavaBeans. Thus, when a client on machine$_A$ interacts with the servlet on machine$_B$, the servlet will ultimately invoke methods on components executing on machine$_C$. When those methods finish, the return values are shipped back to machine$_B$, and the application's method finally returns a result to the client on machine$_A$. (Even if the servlet and EJBs are co-located, the client must interact with the server every time that it interacts with the application.) Figure 1 depicts the runtime situation under J2EE.

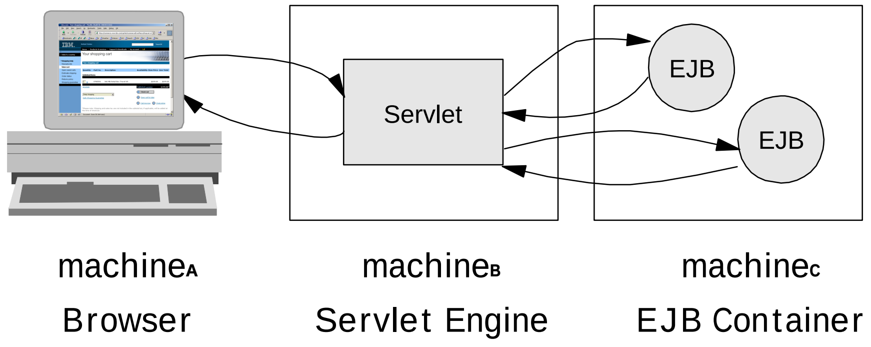

FIGURE 1. J2EE Web-Application Architecture

As a consequence of runtime location *dependence*, each time that a web-client interacts with the application, a client/server interaction must be performed, since a component's Home and instances do not (and *cannot*) reside on the client.

2.2 Implications of Runtime Location Dependence

The fact that current technologies involve runtime location dependence has important implications for application performance and system scalability. First, because the application's code and data cannot be moved closer to one another, application performance suffers because of the network latency incurred when one machine sends method requests or returns data to another. Second, because components are bound to a given server, if many clients access those components, the server becomes a system “hot-spot” since the components cannot be offloaded to less utilized servers. As a result, even technologies that perform dynamic data assembly at the “edge”, assemble items such as HTML page fragments, rather than cached data components with state backed by persistent storage [1].

2.3 Naive Approach

Why can't the application running on machine$_B$ be moved closer to the client so that it executes on machine$_A$? Or, why can't the application's components residing on machine$_C$ be moved to machine$_B$ where the application executes? Consider what will happen if systems naively “copy” or “move” applications from one machine to another. In one approach, only the application's *code* is copied while the application's data remain bound to a specific server. This does not solve the problems discussed above: because data must be accessed remotely, network latency is incurred (reducing performance); similarly, the data server's utilization increases (limiting scalability). In another approach, the application's code *and* data are copied. In many cases, the data simply won't fit on the second server. But, even if the data can be replicated to the server, the application's transactional coherence will break since data are now replicated dynamically throughout the system. Furthermore, security concerns often preclude replication of sensitive data.

2.4 Our Approach: Edge-Server EJB Caching

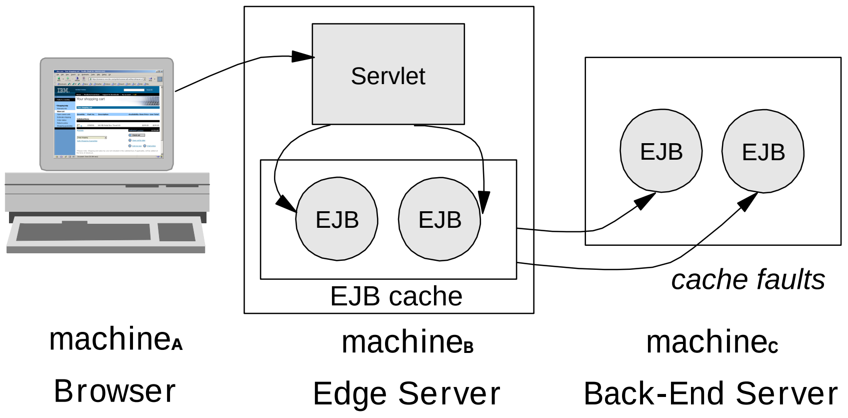

FIGURE 2. Edge-Server EJB Caching Architecture

Edge-Server EJB caching attempts to improve application performance and system scalability by intermediating *edge-server* computers between web-clients and the *back-end server* that hosts the data. The idea is that persistent, “master”, versions of a J2EE EJB component reside on a back-end server, while transient versions of the component are cached, on demand, on edge-servers. Performance is improved (because network latency is reduced) and scalability is increased (because data-serving requests are offloaded from the back-end server to the edge-servers). Figure 2 depicts an application deployed to an edge-server runtime configured for EJB caching.

In this paper we give a detailed discussion of our edge-server caching architecture and current implementation.

3 Requirements

The key requirement faced by our edge-server caching architecture is that little effort be involved in “edge-ifying” an existing application.

- We do not invent a new application component model, but instead use the EJB model of session and entity beans.

- Although the runtime of an edge-server caching system differs from standard J2EE systems, the application developer does not write new code to access the runtime. Instead, tooling takes standard EJBs as input and produces output classes that use the input class's business logic with additional code to access the runtime.
- It is crucial that the edge-server version of the application support an identical, or similar, transactional model to that of the EJB specification. Application developers and clients expect that the system provide both concurrency and transactional isolation; the edge-server runtime must therefore provide both.

4 Single-Logical-Image EJBs

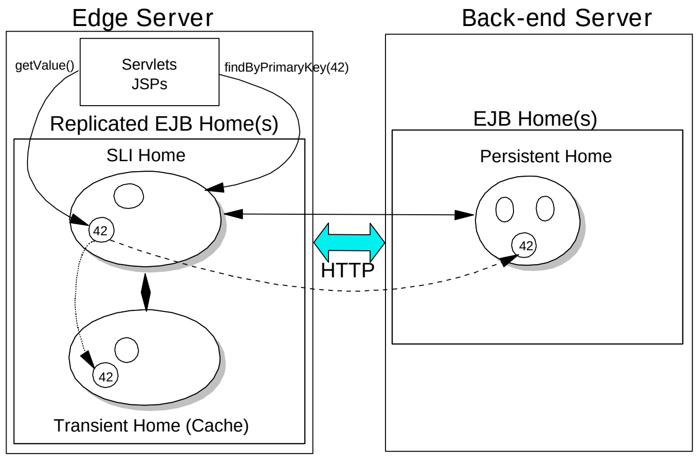

FIGURE 3. Edge-Server EJB Caching: Bean Architecture

A edgeified J2EE application contains three versions of the EJBs used by the non-edgeified application: *persistent*, *transient*, and *single-logical-image* (or *sli*) beans. The persistent beans are simply the beans in the original application: standard EJBs maintained by an EJB container on the back-end server. The transient beans are cached versions of the persistent beans: their state is initialized to that of the corresponding persistent bean, and may then be modified by the application client within a transactional context. Finally, the sli beans “compose” a persistent and transient bean such that the state exchanges between the back-end and edge-servers are coordinated to follow the rules specified by the EJB specification. All three bean types are fully compliant EJBs with Remote and Home interfaces and a Bean implementation. The EJB container that manages the Homes is a standard container. The transient and sli bean types differ from persistent, jdbc, beans only in that they use a transient datastore when loading and storing bean state. In addition, the sli bean delegates all business-logic method calls to the associated transient bean. The three beans share a common identity because `getPrimaryKey` returns the same value for a given bean “triple”. Figure 3 depicts the three bean types.

5 Mementos

In addition to these three bean types, the edge-server caching architecture contains the *memento*[5] artifact. Mementos represent a serializable version of an EJB's state such that:

- A transient bean can be created and initialized with the state of the corresponding persistent bean. Edgeified Homes (and beans) must therefore support an `(ejb)create` method that takes a `Memento` parameter.
- A persistent bean can be updated to reflect the modifications made by an edge-server client to the cached bean. Edgeified beans must therefore support an `updateFromMemento` method.

Mementos must be added to the architecture since the EJB specification explicitly forbids EJBs to be passed “by value”. Mementos have the same notion of “identity” as EJBs, as they support the `getPrimaryKey` method. Conversely, all edgeified beans support the `getMemento` method.

The edge-servers maintain a *TransientStore* that contains a bean's state on a per-transaction basis. For transient beans, the state consists of a memento holding the instance variable state for the current transaction, wrapped with “getter” and “setter” methods. If the EJB existed before the current transaction began, the memento is initialized from the persistent bean on the back-end server. If the EJB was created during the current transaction, the memento is initialized by the `ejbCreate` business logic. For sli beans, the state consists of a memento holding two bean references: one referring to the persistent bean on the back-end server, and one referring to the transient bean on the edge-server. In addition, a sli memento contains a reference to the memento sent by the back-end server, termed the *before-image* memento.

6 Bean Methods

In this section we describe how the methods on a standard EJB are implemented in our edge-server architecture.

6.1 Business Logic

We use the term “business logic” to denote methods that can be implemented only by the bean developer who knows how the EJB is supposed to behave. Business logic obviously includes methods such as “credit” and “debit”. It also includes methods that affect instance variable state such as “getters”, and “setters”. Less obviously, business logic includes methods such as `ejbCreate` and `ejbPostCreate` since they initialize EJB state as a function of the supplied parameters.

Since the architecture requires that edgeified applications behave exactly as the original non-edgeified application, deployment to the edge-server cannot modify business logic in any way. Instead, a sli bean delegates all business logic to its cached transient bean. We discuss later how the transient bean's business logic can be identical to the original, persistent, bean's business logic.

6.2 FindByPrimaryKey

When an edge-server client invokes a `Home.findByPrimaryKey` method, the following steps occur.

1. The Home invokes the `ejbFindByPrimaryKey` method on the sli entity bean.

   a. If the client has previously removed the bean, the bean no longer exists (even if it's still present on the back-end server) and a `FinderException` is thrown.

   b. If it is currently cached (as a transient bean) on the edge-server, the corresponding primary key is returned.

   c. If it's not currently cached on the edge-server, the sli-bean delegates the `findByPrimaryKey` operation to the Home residing on the back-end server. Assuming it exists on the back-end server, the sli bean returns the primary-key to the container.

2. If the bean exists, the Home invokes `bean.ejbLoad()`on the sli bean which:

   a. fetches a memento from the back-end server

   b. uses the memento to create a transient bean on the edge-server

   c. sets the sli-bean's transient and persistent bean references

   d. returns the bean to the client.

The transient bean is now cached on the edge-server, and will be used on subsequent method invocations.

6.3 Custom Finders

In contrast to `findByPrimaryKey`, custom finder methods cannot be executed solely on the client since there is no guarantee that the result set of an arbitrary query is already cached on the client. Conversely, because we cannot overwrite changes made to the cached objects on the edge-server, we cannot simply use the result set returned by query execution on the back-end server. Instead, we use the following algorithm.

1. Execute the query on the back-end server and return a collection of mementos corresponding to the query's result set of EJBs.
2. On the edge-server, iterate over the set of mementos, and construct the edge-server query set:

   a. Get the memento's primary key.

   b. If the corresponding (sli) EJB has been deleted by the edge-server client, do not include the EJB in the edge-server query set.

   c. If the corresponding EJB is already cached on the edge-server, use the edge-server version in the query set, because the back-end server's version of the EJB may be out of date for this transaction.

   d. If the corresponding back-end server EJB is not already cached on the edge-server, create sli and transient EJBs and mementos on the edge-server as if the edge-server application had requested this EJB from the back-end server. The sli EJB is added to the edge-server query set.

   At this point, a valid query set (i.e., with the most recent back-end server state and edge-server state) for this query is now cached on the client.
3. Execute the original query predicate against the transient Home, and save the result set. This step requires the ability to implement the query logic against a transient datastore, without the benefit of an SQL query engine. Subsequent iteration over the result set will convert the transient EJB to a primary key: the sli Home converts (as required by the EJB specification) the key to a sli EJB.

6.4 Bean Creation

When an edge-server invokes a `create` on a sli Home, the Home drives the `ejbCreate` method on the new sli bean, and the following steps occur.

1. The sli bean invokes the corresponding `create` method on the transient Home to create the corresponding transient bean.
2. The sli bean creates a sli memento, setting its identity from the key of the new transient bean.
3. The sli bean's memento sets its transient bean references to the new transient bean.
4. The sli bean stores its memento in the TransientStore.

We discuss how the `create` operation is propagated to the master-copy of the EJB on the back-end server in Section 8.

6.5 Bean Removal

When an edge-server invokes a `remove` on a sli Home or Remote, the Home drives the `ejbRemove` method on the sli bean, and the following steps occur.

1. The sli bean invokes `remove` on the transient Home.
2. The sli bean removes its memento from the TransientStore.

We discuss how the `remove` operation is propagated to the master-copy of the EJB on the back-end server in Section 8.

7 Application Transformation

The use of sli EJBs (Section 4), Mementos (Section 5), and sli bean method algorithms (Section 6) enables us to satisfy one of the requirements for an edge-server caching architecture described in Section 3. The EJB component model is perfectly suitable for use in an edge-server environment, and there is no need to use one component model for standard J2EE deployment, and another, new, component model for edge-server deployment. It may seem, however, that the number of additional beans (transient and sli) and the complexity of bean algorithms, preclude meeting another of the requirements: namely, that the application developer not write additional code to deploy an application on an edge-server. In this section we explain how tooling enables us to meet this requirement as well, so that standard J2EE applications are transparently transformed for edge-server deployment.

7.1 Approach

The approach used in the tooling is based on these observations:

- The EJB specification already distinguishes the bean-provider role from the container-provider role. This allows the edge-server runtime function to be provided through container code without modifying the beans.
- The EJB specification further distinguishes an EJB's Remote interface as seen by the client from the EJB's bean implementation as supplied by the bean provider. On the edge-server, we can therefore replace the standard, jdbc, bean implementation with a sli bean implementation without modifying the client (application) view of the EJBs.
- A standard EJB is already required to provide the component's business logic in the Bean implementation. The sli bean algorithms (Section 6) do not modify this existing business logic, and simply delegate

business logic to the standard bean. The sli bean algorithms are application-independent, and this logic can therefore be represented in a code-generation tool for all EJBs.

7.2 Code Generation

7.2.1 Input

The code generation tool is supplied the following input:

- A standard, jdbc, EJB, including the Home, Remote, Bean, and Key classes. Finder logic is provided in the form used in standard approaches to container-managed persistence, typically some sort of finder-helper class. We require that the Bean class follow the EJB Version 2 guidelines that forbid the use of instance variables, and instead require that instance variables be accessed through abstract *getter* and *setter* methods.
- An implementation, in transient space, for every finder method. Just as developers must supply finder logic in jdbc terms (typically, via an SQL statement), edge-server deployment requires that developers supply finder logic for the cached transient EJBs in “transient” terms. Unfortunately, in the absence of an Object Query Language [7] implementation for transient datastores, this logic must be specially supplied for edge-server deployment.

7.2.2 Output

Given these input classes, the code generation tool emits the following classes:

- A memento class, used by the transient and jdbc beans.
- A sli memento class, used by the sli bean.
- A transient EJB including the Home, Remote, Bean, and Key classes.
- A sli EJB including the Home, Remote, Bean, Key, and Finder classes.

8 Commit Processing

By enabling an edge-server to cache EJBs, J2EE applications can run, unmodified, on the edge-server, potentially improving application performance and system scalability. At some point, the application “commits” (e.g., the user clicks “buy” in a shopping cart application), and the cached state must be transactionally committed at the back-end server. Although the edge-server has transactionally isolated the client from other edge-server clients by storing mementos on a per-transaction basis, the persistent master-copy resides only on the back-end server and has (possibly) been concurrently modified by other clients. In this section we describe the algorithms used to transactionally commit an edge-server application.

The key idea is that the edge-server maintains *before-image* and *after-image* mementos on a per-transaction, per-EJB, basis. The before-image represents the state of the EJB at the time that it was cached on the edge-server. The after-image represents the state of the EJB at the time that the transaction commits. This memento pair is sent to the back-end server which determines whether the transaction can be committed or must be aborted.

Referencing or Updating an EJB. The back-end server must first determine that the EJB has not been modified (and committed) by some other transaction. This is done by comparing the edge-server's before-image against the persistent EJB's current memento. If they differ, the transaction is aborted.

Assuming that the EJB has not been modified by another transaction, the back-end server compares the edge-server's memento pair. If they differ, the persistent EJB uses `updateFromMemento` to have its state reflect the changes made at the edge-server.

Creating an EJB. By determining that the before-image is “null”, the back-end server detects that the memento tuple corresponds to an EJB *created* on the edge-server. It therefore uses the `Home.create(memento)` to create a corresponding persistent EJB. If another transaction created an EJB with the same primary key during the edge-server transaction, there is a conflict and the edge-server transaction must be aborted.

Removing an EJB. By determining that the after-image is “null”, the back-end server detects that the memento tuple corresponds to an EJB *removed* on the edge-server. It therefore uses the `Home.remove` to remove the corresponding persistent EJB. If another transaction removed the EJB during the edge-server transaction, there is a conflict and the edge-server transaction must be aborted.

9 Transactional Behavior

9.1 Concurrency Control

The commit algorithm described in Section 8 is an *optimistic* concurrency control (CC) algorithm, in contrast to the *pessimistic* (or locking) concurrency control algorithm typically used on the back-end server [6]. Rather than have the back-end server lock an EJB when it is cached on the edge-server, we allow other transactions to concurrently access the EJB. The edge-server transaction commits its changes to the back-end server using a pessimistic transaction on the back-end server. That transaction, as described above, commits only if it validates the optimistic assumption that its EJBs were not concurrently modified by other transactions.

We have chosen an optimistic algorithm because we speculate that it will have better performance than a pessimistic algorithm in a typical EJB caching configuration. Our reasoning is that if all the concurrency control operations must be propagated to the master (back-end) server, this will incur a significant performance penalty. Although the edge-server must still execute the first phase of a query on the back-end server, optimistic CC still requires fewer interactions than pessimistic because (1) lock-upgrade operations on the edge-server do not need to be propagated to the back-end server, and (2) EJBs can be cached between transactions without accessing the back-end server.

9.2 Degree of Isolation

Transactions are commonly assumed to provide *repeatable-read*, or *degree-3* isolation. Repeatable-read isolation guarantees that, within a single transaction, an application will not see data changes caused by concurrent transactions. The system must therefore detect any changes that would affect query-result sets, even if these changes are not harmful within the context of the application. For optimistic CC, this appears to require that all query result-sets be preserved and verified at transaction commit time. Although transactional caching of data has been studied in client/server environments [4], implementing repeatable-read is too expensive in an edge-server environment.

Instead, our system uses *cursor-stability* or *degree-2* isolation. Here, the system defines a set of cursors, which are used by the application to dynamically indicate which records are in use by the application. Only records which are referenced by a cursor (or modified by the application) must remain unchanged when other transactions commit. Queries executed more than once may return different result-sets, depending on the actions of other concurrent transactions. Because less isolation is required, cursor-stability typically offers much more concurrency than repeatable-read, and much lower overhead for distributed systems. Such isolation guarantees are acceptable for most applications: as noted in [6] most non-distributed systems provide, and most production applications use, cursor-stability isolation.

By providing cursor-stability isolation, application developers are assured that the application will behave in a well-understood way. For most purposes, existing applications will behave as in the non-edge-server environment. This isolation model provides a good compromise between ease-of-use and performance.

10 Project Status

We have implemented the ideas described in this paper in a prototype edge-server. The prototype allows EJBs originally deployed on standard J2EE containers to be redeployed on an edge-server enabled to do EJB caching from a back-end server. The project's ongoing work is related to analysis of application performance when deployed to such edge-servers. Insights gained from this analysis is driving optimizations to the caching and transaction algorithms.

11 References

[1] *A Distributed Infrastructure for e-Business*. http://www.akamai.com/en/html/services/white_paper_library.html. 2002.

[2] *OMG Specifications and Process*. http://www.omg.org/gettingstarted/. 2002.

[3] *Enterprise JavaBeans Specifications*. http://java.sun.com/products/ejb/docs.html. 2002.

[4] Franklin, M. J., Carey, M. J., and Livny, M. *Transactional Client-Server Cache Consistency: Alternatives and Performance.* ACM Transactions on Database Systems. Vol. 22, No. 3. September 1997. pp. 315-363.

[5] Gamma, E. et al, *Design Patterns*. Addison Wesley Longman, Inc. 1995.

[6] Gray. J., Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. 1993.

[7] *Object Data Management Group: Standard Overview*. http://www.odmg.org/standard/standardoverview.htm. 2002.

[8] *Java Remote Method Invocation (RMI)*. http://java.sun.com/docs/books/tutorial/rmi/. 2002.

[9] *JAVA SERVLET TECHNOLOGY IMPLEMENTATIONS AND SPECIFICATIONS*. http://java.sun.com/products/servlet/download.html#specs. 2001.