Alternative Edge-Server Architectures for Enterprise JavaBeans Applications

Avraham Leff and James T. Rayfield

IBM T. J. Watson Research Center, P. O. Box 704 Yorktown Heights, NY 10598
{avraham,jtray}@us.ibm.com

Abstract. Edge-server architectures are widely used to improve web-application performance for non-transactional data. However, their use with transactional data is complicated by the need to maintain a common database that is shared among different edge-servers. In this paper we examine the performance characteristics of alternative edge-server architectures for transactional Enterprise JavaBeans (EJBs) applications. In one architecture, a remote database is shared among a number of edge-servers; in another, edge-servers maintain cached copies of transactionally-consistent EJBs. Importantly, the caching function is transparent to applications that use it.

We have built a prototype system in which edge-servers are enhanced with an EJB caching capability. The prototype enables a realistic comparison of these architectural alternatives. We use a benchmark EJB application to drive a performance analysis of the architectures. We also compare these edge-server architectures to a classic clustered datacenter architecture.

1 Introduction

1.1 Edge-Server Architectures

Edge-server architectures [16][6][2][5] are widely used to improve web-application performance by moving web-content from back-end servers to the edge of the network (e.g., internet service providers). By caching data at the "edge", edge-servers increase throughput (by offloading traffic from back-end servers), and reduce application latency (by moving data closer to the client).

Unfortunately, the data replication and update algorithms used in current edge-server architectures are severely limited. First, updates take place at a single central server (the master copy of the database). Updates to shared data cannot be made at the edge servers. Second, updates are pushed (or pulled) from the central server to the edge servers in a non-transactional fashion. Thus different edge servers will receive the updates at different times (i.e. the data as seen across all edge servers is not consistent).

For many web applications, these are reasonable compromises which are made in order to achieve high performance and scalability. In a typical ecommerce

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 195–211, 2004.

application, it is not critical to have all the edge-cached catalogs updated transactionally. Similarly, it is not necessary to allow applications running at the edge to update the shared catalog database.

However, for transactional data (e.g. bank accounts), this is not sufficient. Bank accounts must show the same balance at every edge server, and update (e.g. debit) operations must happen in an ACID[11] fashion. The simple approach is to centralize transactional data, and not to replicate or cache it. Unfortunately this brings back the latency and bandwidth problems that were addressed by edge servers for non-transactional data.

This paper explores whether the benefits of edge-server technology can be extended to applications requiring the use of transactional data. Specifically, we examine whether *Enterprise JavaBeans*[4] (EJBs) applications can be deployed to edge-server architectures. EJBs are an example of a transactional component model; and, while this paper is focused on EJB technology, it applies more generally to any framework for distributed enterprise components[13].

1.2 Enterprise JavaBeans Component Model

EJBs are a component model for enterprise applications. (We refer here to the *entity bean* flavor of EJBs, in contrast to the *session bean* flavor.) Like CORBA[3] and RMI[17], EJBs are a distributed component model, and, as such, encapsulate both "data" (the component's state) and "code" (business logic in the component's methods). In addition, EJBs automatically supply common requirements of enterprise applications such as persistence, concurrency, transactional integrity, and security. Bean developers focus on the business logic of their application; when deployed to an EJB *container*, the components are embedded in an infrastructure that automatically supplies the above requirements. For example, a deployer might specify that an *Employee* entity bean's state is backed by persistent storage in the *HR* relational database, specifically in its *Employees* table. EJBs use declarative transaction management on a per-method basis, so that the `incrementSalary` method might be declared to require a transactional scope.

1.3 Edge-Servers and EJBs

While edge-servers currently cache both dynamic and static web-content, the cached data are not transactional. Static data are especially easy to cache because they are infrequently updated. Even when dynamic data are cached, updates typically do not need to be propagated atomically throughout the web cluster, since no transactional model is provided. Web-servers can therefore use algorithms which are expensive for write operations, and which do not provide a traditional *ACID*[11] transaction model[12]. Such applications and environments differ greatly from that of EJBs in which writes are frequent and strong transactional guarantees must be provided.

Specifically, edge-server caching of EJBs faces the following challenges:

- EJB caching must deal with read/write data as well as read-only data.
- As a stronger requirement than read/write capability, EJB caching must provide transactional consistency among the cached replicas.
- "Cache-enabling" existing applications and J2EE application servers must involve little effort. Customers should not be forced to modify existing applications in order to improve performance. Customers will also not want to maintain two programming models: one, for non-cache-enabled applications, and one for cache-enabled applications. Specifically, an EJB caching framework should have the following features:
  - It should not inject a new application component model, but instead use the EJB model of session and entity beans.
  - Although the runtime of cache-enabled application servers differs from standard J2EE application servers, the application developer should not be forced to write new code to access the runtime. Instead, tooling should take standard EJBs as input and produces cache-enabled EJB implementations with the same Java interface as output.
  - The cache-enabled version of the EJBs should support the same transactional model as described in the EJB specification: i.e., it must provide concurrency and transactional isolation.

While EJB caching has been successfully applied [14] to improve throughput in low-latency, clustered, environments, it does not necessarily follow that caching will be useful for high-latency (several hundred milliseconds per interaction) edge-server environments. The key issue is that transactional consistency requires that the EJB state at each of the edge-servers be synchronized with the persistent state of the remote database. This implies that whenever a transaction commits, at least one high-latency round trip must be performed between an edge-server and the remote database. First, the edge-server must transmit its transactional state (e.g., the set of EJBs modified during the transaction) to the remote database. In contrast to non-transactional data, an edge-server cannot independently commit local modifications to the EJBs because it must ensure that these changes do not conflict with the actions of other edge-servers. Second, after receiving the transaction state, the remote database must determine whether the transaction can be committed or whether it must be aborted; it then informs the edge-server of the transaction's outcome. The duration of this round-trip may be sufficiently long as to counter one of the basic motivations for edge-server architectures: namely, to use cached data to reduce application latency

1.4 Relationship to Other Caching Work

We have already explained that the key difference between classic web-caching and edge-server caching of EJBs involves the fundamental requirements of transactional data. EJB-caching more closely resembles distributed client-server database systems. Typically, such systems use one of two approaches: function

(query) shipping or data shipping. In function-shipping systems, operations applied to shared data are propagated to the shared server. In data-shipping systems such as ours, the database clients cache a portion of the database, and operations are executed against the cached data on the client. Data-shipping systems require the use of a transactional cache-consistency algorithm in order to maintain ACID properties among the different client applications. A common approach is to designate one copy of the database as the "master" copy, and use algorithms which synchronize access (and recovery) against this copy.

Many such algorithms have been proposed and studied for distributed client-server database systems[8]. In terms of this taxonomy, we use a detection-based algorithm, with deferred validity checking, and invalidation when notified by the server about an update. Our system is somewhat different, using a component-server model rather than page-based models. In our work, we have extended the transactional consistency algorithm to include predicate-based queries, rather than simply direct access. This forces us to deal with more complex isolation issues such as the "phantom-read" problem.

Most importantly, our work addresses the issue of transparently cache-enabling an existing, high-level component API such as the EJB model.

More recent work in distributed client-server caching attacks performance issues by relaxing the consistency requirements. For example, DBCache[15] uses the federated features of DB2 to maintain a partial copy of a database that is weakly synchronized with the database server. Application queries are then executed against the cached database. DBProxy[1] retains the results of previously executed queries in a cache; this cache is then used to satisfy subsequent queries or subsets of the original query. Both of these approaches improve performance at the cost of replacing the traditional transactional guarantees with "time-based" guarantees: the data are only guaranteed to be up-to-date within some specified time period. In contrast to some of this database caching work, we assume that cached-enabled applications will expect the same transactional model as non-cached-enabled applications (i.e. strict ACID semantics). Furthermore, we examine the caching of transactional data to high-latency environments such as edge-servers.

1.5 Key Contributions

One contribution of this work is to advance the "state of the art" of edge-server architecture by demonstrating that edge-servers can successfully cache transactional, as well as non-transactional, data. We quantify the benefits of this approach by comparing the performance of a benchmark EJB application when deployed to (1) a cache-enabled edge-server architecture and (2) a "vanilla" edge-server architecture that must access a remote EJB application server whenever an application accesses EJBs. Further, we examine which of two alternate EJB caching architectures are best suited to an edge-server environment.

Another contribution of this work is to question whether edge-servers should cache transactional data at all. As explained above, transactional requirements imply that at least one high-latency round-trip be performed whenever a trans-

action commits. This raises a intriguing question: perhaps a classic, clustered (non-edge) data-center architecture is best suited for high-latency transaction environments. Rather than clients running application on edge-servers, clients may be better served by directly accessing a remote EJB application server. This paper quantifies the benefits of the clustered data-center for transactional applications deployed to high-latency environments, and shows that the (cache-enabled) edge-server architecture is a valid architecture even for transactional applications.

1.6 Paper Organization

The rest of this paper is organized as follows. First, we describe the EJB caching in some detail; then, we describe a set of alternative high-latency architectures for EJB (transactional) applications. The remainder of the paper is a performance evaluation of these architectures based on a benchmark EJB application.

2 Caching Framework

2.1 Application Components

Our caching framework [14] substitutes *Single Logical Image* (or *SLI*) Home and bean implementations for the standard JDBC Home and bean implementations used in the non-cache-enabled application. The caching runtime copies the state of the relevant persistent EJBs into *transient* EJBs as necessary, and then transparently delegates to them. The SLI bean introduces no business logic of its own; it simply delegates all method invocations to the transient bean. Because the transient bean implements the same interface as the original, JDBC, bean and differs only in the way it accesses its datastore, the *business logic* of the application is unchanged.

Since the EJB specification requires that EJBs cannot be serialized (rather, they are passed by reference), we must provide "value objects" that can be passed between address spaces. We term these value objects *mementos*[10]. Mementos have the same notion of "identity" as EJBs, as they support the `getPrimaryKey` method. Transient EJBs introduce two memento-related methods: `create(Memento)` (on the EJB home) so that they can be created from persistent state; and `Memento getMemento()` (on the Remote interface) so that the caching runtime can update the persistent state from the client's cached state. The memento containing the state at the beginning of a transaction is called the *before-image*; the memento containing the state at the transaction's end is called the *after-image*. The cache-enhanced application server maintains a transient datastore of memento instances.

The EJB container that manages the transient and SLI Homes is a standard container. The SLI and transient beans are fully compliant EJBs with Remote and Home interfaces and a Bean implementation. They differ from the familiar persistent, jdbc, beans only in that they use a transient datastore when loading

and storing bean state. A SLI and associated transient bean share a common identity because `getPrimaryKey` returns the same value; this value is identical to that returned by persistent bean in the original application.

2.2 Populating the Cache

The EJB cache is populated in one of the following ways:

1. Direct application access through the bean's primary key, via an `ejbLoad` or `findByPrimaryKey` invocation.
   In this case, the cache runtime first determines whether the bean is already cached. If a cache miss occurs, the cache runtime fetches the before-image directly from the persistent datastore and caches it for subsequent use.
2. Indirect application access, when the bean is part of the result set returned by a custom finder method invocation.
   Unlike a direct access, the cache runtime must first run the query against the persistent datastore because only that datastore is guaranteed to have the entire (potential) result set available. The result set returned by the persistent datastore is then used to populate the cache. However, in order to guarantee that the application sees its prior updates, the runtime ensures that result set elements that were cached prior to the custom finder invocation are not overlaid with the current persistent state. Finally, with the finder's entire result set available in the cache, the custom finder is run against the transient Home, and that result is returned to the application.
   Other transactions may commit their state to the persistent datastore while a given transaction executes on a cache-enhanced application server. This implies that the algorithm used to implement custom finders can add members to the result set if the application executes the finder multiple times in a single transaction. The isolation model supplied by the framework is therefore slightly less powerful than *serializable* isolation, and corresponds instead to *repeatable-read* isolation[11].
3. Explicit bean creation by the application.
   In this case, the appropriate `create` method is invoked on the SLI home, delegated to the transient Home, and the resulting bean is cached.

2.3 Implementing Transactions

Populating a transient EJB cache is only one part of an EJB caching framework. The system must also provide transactional semantics identical to that provided by a non-cache-enabled runtime to a J2EE application. Because we want to allow inter-transaction caching (i.e., to allow EJBs cached by one transaction to be available to other, concurrent and subsequent, transactions) the system uses *optimistic* rather than *pessimistic* concurrency control[11] (or *detection based* rather than *prevention based*[8]). Under the pessimistic approach, one transaction cannot use data cached on behalf of another transaction because cached data must be locked throughout the period that it's accessed. The long duration

of the lock period implied by inter-transaction caching makes the pessimistic approach much less feasible than the optimistic approach.

In our approach, a common transient store (not EJB-based) is maintained alongside a per-transaction transient store. When a direct-access operation results in a cache miss on the per-transaction store, the common store is checked for a copy of the EJB data before an attempt is made to access the persistent EJB. The disadvantage of this approach is that, since each cache-enabled application server maintains its own common transient store, the "conflict window" (i.e., the period of time in which an application's persistent state can be concurrently modified by some other transaction) widens. Just as we replace the original application's JDBC Homes and beans with their SLI equivalents, we replace the original pessimistic JDBC Resource Manager with an optimistic SLI Resource Manager.

Whenever the cache runtime must access the persistent EJBs (in any of the "populate" scenarios discussed above), it creates a separate (non-nested) short transaction for the duration of the access. This transaction is committed immediately after the access completes so that locks are released quickly by the persistent store. The application-generated transactions are thus decoupled from the datastore transactions used to provide data to the cache and update data from the cache. A single application transaction thus typically brackets multiple persistent datastore transactions. Finally, when the application transaction running on the cache-enabled application server commits, a persistent datastore transaction is run to commit the application's state changes.

The isolation semantics provided to the application are the following. If another transaction, t_2, modifies the persistent datastore's data from the state that existed at the beginning of the application's transaction t_1, t_1 will be aborted. We implement this behavior by comparing the before-image of every bean accessed in the transaction to the current corresponding image in the datastore at commit time. Only if no conflicts exist are t_1's EJBs committed to the datastore. During a successful commit, the transaction's set of after-image mementos are written to the datastore in a single datastore transaction. More subtly, if the application creates an EJB, the system must also verify that no EJB with the same key exists at commit time. Similarly, if the application removes an EJB, the system must also verify that the current-image still exists before deleting it and committing the transaction.

2.4 EJB Caching Architectures

Two EJB-caching configurations are discussed in this paper. In the *split-servers* configurations, the cache-enhanced application server requires a *back-end* application server that is one deployment "tier" removed from the client. The logic that handles cache misses and the logic that implements the optimistic concurrency control algorithm reside on the back-end server (see Figure 1). In the *combined servers* configuration, the back-end-server function is merged in the cache-enhanced application server (see Figure 2). This has the advantage of removing cross-address-space communication between the application servers, which im-

proves performance under some scenarios. The disadvantage of the combined-servers approach is that the communication protocol between the cache-enabled application server and the database is whatever the JDBC driver uses to communicate with the database. Such protocols are typically not suitable for internet or Grid[9] usage due to firewall and security issues. In contrast, the back-end server approach introduces a known interface between the application server and back-end server. The protocol used to bridge this gap can be customized appropriately to the environment.

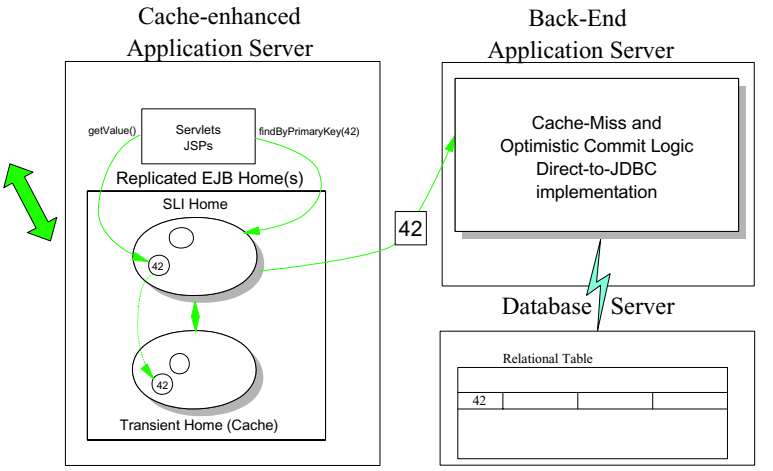

Fig. 1. Split-Server Configuration

3 High-Latency Architectures for EJB Applications

High-latency communication is a principal characteristic of internet environments. In order to better evaluate the benefits of edge-server use of EJB-caching, we characterize three architectures in terms of the location of the high-latency communication path.

1. An architecture in which a remote database is shared by a number of edge-servers. We term this an *ES/RDB* architecture. The edge-servers can be optionally enhanced with an EJB-caching capability. In that case, the ES/RDB configuration corresponds to the "combined-servers" EJB-caching configuration (Figure 2).
   In the ES/RDB architecture, the high-latency communication path lies between the application servers and the database (see Figure 3).
2. An architecture in which cache-enhanced application servers coordinate transactional activity using a common, remote, back-end server. The remote back-end server is closely clustered with a database. We term this an *ES/RBES* architecture.

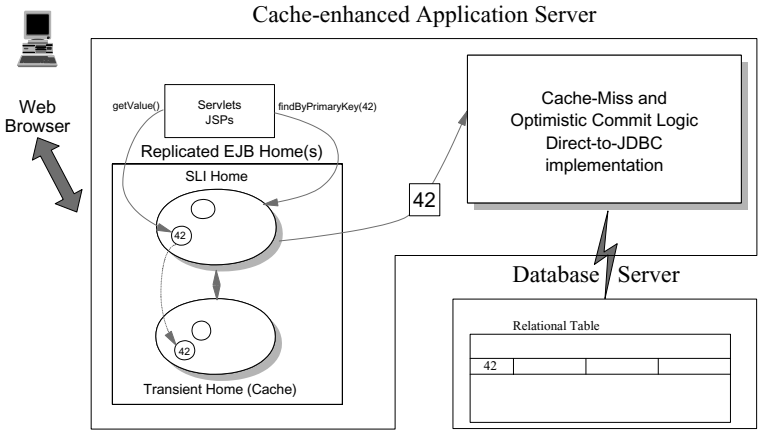

Fig. 2. Combined-Server Configuration

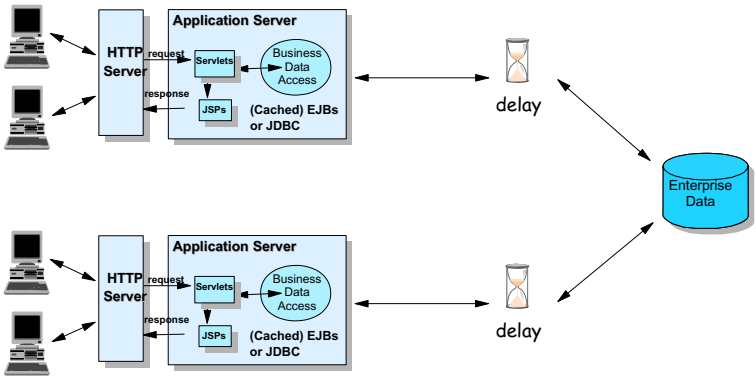

Fig. 3. Edge-Servers Sharing a Remote Database (ES/RDB)

In the ES/RBES architecture, the high-latency communication path lies between the cache-enhanced application servers and the back-end server that provides the cache-miss and transaction commit logic (see Figure 4). This architecture is meaningless to anything but a EJB-caching architecture, and corresponds, specifically, to the "split-servers" configuration (Figure 1).

3. A classic clustered datacenter architecture, in which clients do not interact with edge-servers but instead communicate directly with remote application servers. We term this a *Clients/RAS* architecture.

   In the Clients/RAS architecture, the high-latency communication path lies between the web-clients and the remote application servers (Figure 5). As explained previously, in a transactional high-latency environment, this is a plausible alternative to an edge-server architecture.

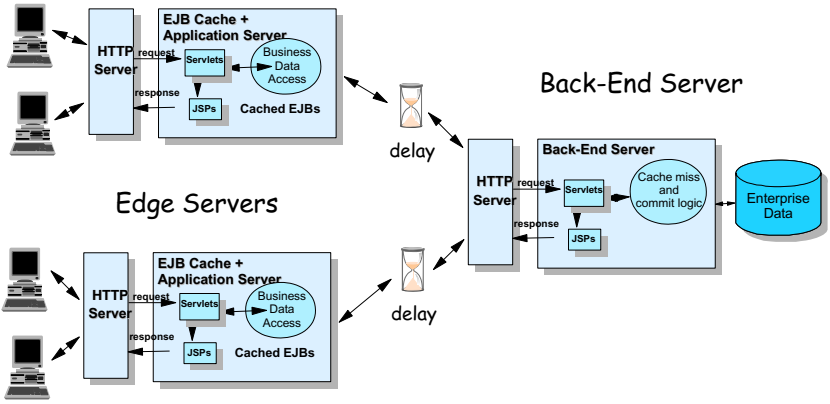

Fig. 4. Edge-Servers Sharing Remote Back-End Application Server (ES/RBES)

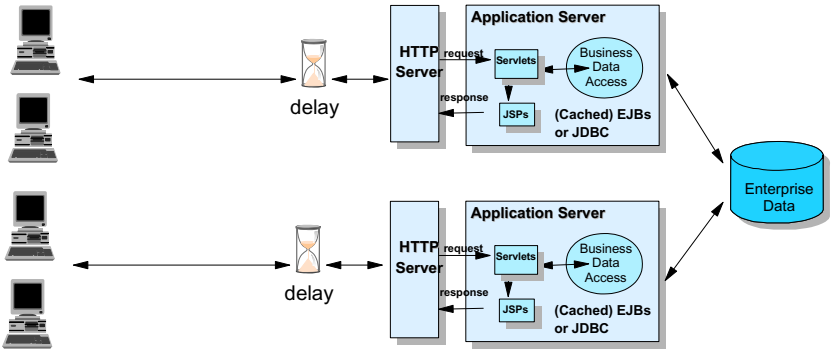

Fig. 5. Clients Accessing Remote Application Servers (Clients/RAS)

4 Performance Evaluation

In this section we evaluate:

1. Whether, and to what degree, Cache-enabled edge-servers improve the performance of EJB applications as compared to "vanilla" edge-servers.
2. Which version of the EJB-caching architecture is best suited for an edge-server environment.
3. Whether edge-servers – even when cache-enabled – are in fact suitable for transactional applications.

We do this by running cache-enabled and non-cache-enabled versions of a sample application in each of the three latency configurations discussed in Section 3. Before describing the test application, we describe the test configuration.

4.1 Test Configuration

System Components. The application server, *delay-proxy* server, back-end server, and database server run on four separate machines. Each is a uniprocessor, Pentium III, 1266MHz Intel machine with 256MB physical memory and 1GB paging space. The machines run RedHat Linux 7.1 (kernel 2.4.2-2), and are connected with a 100 Mbit Ethernet. DB2, version 7.2 provides the persistent datastore. The JVM is IBM's JDK Version 1.3.1; and the J2SDKEE version is 1.2.1. Tomcat, version 4.1.12 is used as the servlet engine. A prototype J2EE container is used for the SLI, persistent, and transient containers.

Delay Proxy. Our machines are deployed in a LAN environment with latency of, at most, several milliseconds. Because the performance evaluation requires that the application be deployed in an environment with latency of tens of milliseconds, we use a proprietary delay proxy to emulate a high-latency communication path. The delay proxy process runs on a dedicated machine. Depending on which communication path has high-latency, all communication between the specified endpoints (e.g., application servers and the database server) is intercepted by the delay proxy listening at a specific port. The proxy reads the incoming data, interposes a specified amount of delay, and only then writes the incoming data to the original destination. The data interception is functionally transparent to both the load generation program and the application. Performance results were generated by varying the delay injected by the proxy and determining the resulting application client latency.

4.2 Test Application

Trade2 is a publicly available application developed by IBM that "models an online brokerage firm providing web-based services such as login, buy, sell, get quote and more". Table 1, extracted from the application's documentation, describes the Trade2 runtime and database usage characteristics. A client interaction with the application involves a random sequence of the "trade actions" listed in the Table, bracketed by a "login" and "logout". The client web-browser sends a trade action request to a servlet; the servlet invokes the appropriate session bean method; the method, in turn, drives methods on or more entity beans. Finally, the result of the "trade action" is constructed in a JSP and returned to the client browser. On average, a single session consists of about 11 individual trade actions. We consider Trade2 to be a sufficiently complex application to make it a suitable J2EE benchmark. We downloaded version 2.531, cache-enabled it, and then evaluated its performance.

4.3 Results Roadmap

To evaluate the effectiveness of a given architecture, we focus on two statistics: the latency of a client/server interaction, and the bandwidth required to service

Table 1. Trade Runtime and Database Usage Characteristics

Trade Action	**Description**	**CMP Bean Operation**	**HTTP Session**	**DB Activity** ($C/R/U/D$)
Login	User sign in, session creation	Update	Create, Update	Registry `R, U` Account `R`
Logout	User sign-off, session destroy	Update	Read, Destroy	Registry `R, U`
Register	Create a new user profile and account	Multi-Bean Create	Create, Update	Account `C, R`, Profile `C`, Registry `C`
Home	Personalized home page including current market conditions	Read	Read	Account `R`
Account	Review current user profile information	Read	Read	Profile `R`
Account Update	"Account" followed by user profile update	Read/Update	Read	Profile `R, U`
Portfolio	View users current security holdings	Read	Read	Holding `R`
Quote	View a current security quote	Read	Read	Quote `R`
Buy	"Quote" followed buy a security purchase	Multi-Bean Read/Update	Read	Quote `R`, Account `R, U` Holding `C, R`
Sell	"Portfolio" followed by the sell of a holding	Multi-Bean Read/Update	Read	Quote `R` Account `R, U` Holding `D, R`

the client's request. These results are presented for the performance of the Trade2 benchmark in the three architectures discussed above.

Within a specific architecture, the effectiveness of EJB caching is evaluated by comparing its performance against two, non-cached-enabled, versions of the application.

- *JDBC*: a pure JDBC [7] implementation, included in Trade2. We include this algorithm because JDBC implementations are commonly understood to provide better performance than "higher-level" implementations such as EJBs.
- *Vanilla EJBs*: an implementation using non-cached EJBs with bean-managed-persistence (BMP), with persistence provided by DB2. This corresponds to the EJB-ALT mode in Trade2.

Results were obtained in a "low-load" situation so as to factor out queuing delay effects: specifically, one virtual client makes repeated requests to the Trade2 running on a single application server. The latency metric represents average latency of a round-trip interaction between the client and the application as a function of the (*one-way*) delay injected by the delay proxy at the specified

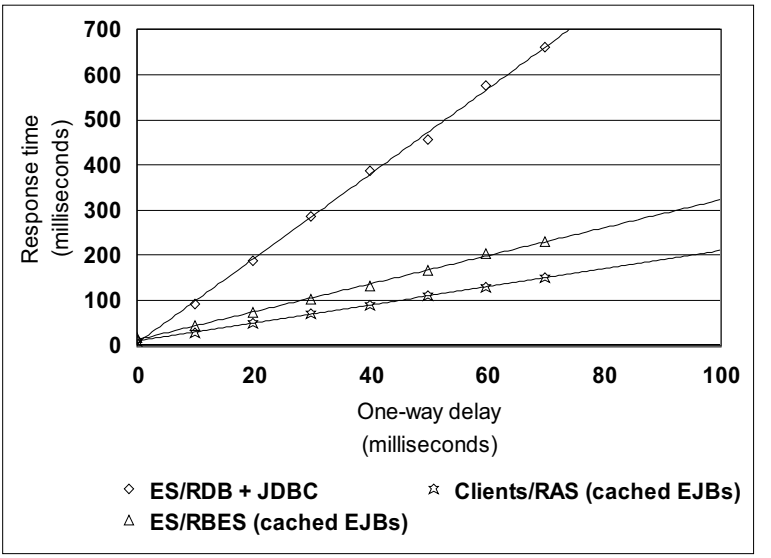

Fig. 6. Comparison of High-Latency Architectures

communication path. The set of possible trade actions are those listed in Table 1. (Both latency and the delay are specified in milliseconds.) In addition to individual data points, we show a linear curve extrapolating the data with an R^2 (quality of fit) of 99%. Client requests are driven by a load generator program on a dedicated machine. Reported latency is the batched (over 20 batches) average of a run consisting of 300 sessions. Each session consists, on average, of about 11 client/server interactions. A warmup period, consisting of 400 sessions, preceded each run.

4.4 Results

Figure 6 shows the latency behavior of the application when deployed to the three architectures; Figure 8 shows the bandwidth required to service client requests for the architectures.

We first observe that the non-edge-server architecture (Clients/RAS, "stars") has lower latency than either of the two edge-server architectures. We also observe that, of the two edge-server configurations, EJB-caching enables the ES/RBES architecture ("triangles") to perform far better than the best algorithm of the ES/RDB ("diamonds") architecture.

One way to understand these results is to examine *latency sensitivity* (Table 2), defined as the increase in the latency of a single client interaction for each unit increase in communication delay.

We see that the non-edge-server architecture is the least sensitive to increases in latency: every increase in *one-way* latency causes a *two-fold* increase in round-trip latency. This is because once the request is received by the application server,

Table 2. Algorithm Sensitivity to Communication Latency

ES/RDB		**ES/RBES**		**Clients/RAS**	
Algorithm	*Sensitivity*	*Algorithm*	*Sensitivity*	*Algorithm*	*Sensitivity*
Cached EJBs	13.0	Cached EJBs	3.1	Cached EJBs	2.0
JDBC	9.4	JDBC	N/A	JDBC	2.0
Vanilla EJBs	23.6	Vanilla EJBs	N/A	Vanilla EJBs	2.0

latency does not affect processing of the request in any way. In contrast, even the best performing algorithm of the ES/RDB architecture is much more affected by latency (9.4) since it incurs this penalty every time that a database access is performed. Note that multiple database requests are required per client request. Also, the JDBC implementation in the ES/RDB architecture is less affected by latency than either vanilla or cached EJBs (see Figure 7). This is likely because the (hand-crafted) JDBC implementation is better optimized than the tooled EJB implementation. For example, BMP EJBs have difficulty caching the results of a `findByPrimaryKey` operation, even though such results are typically reused immediately.

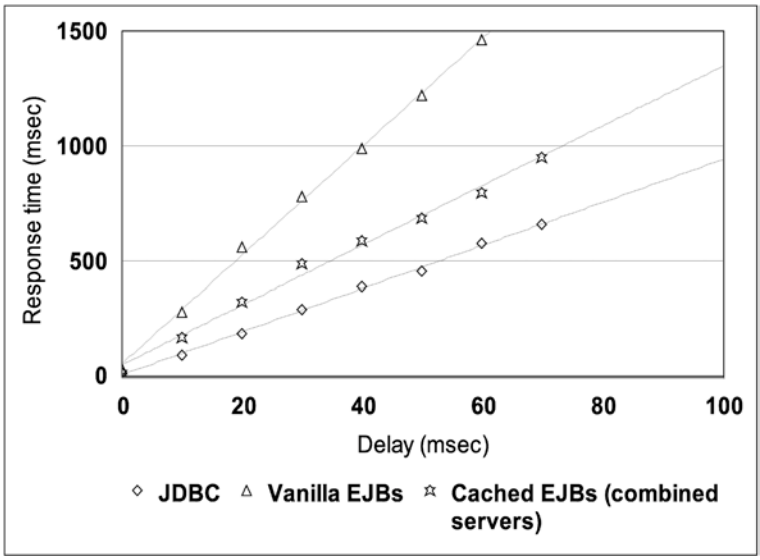

Fig. 7. Edge-Servers Accessing Remote Database

Compared to vanilla EJBs, EJB-caching is quite effective in reducing latency sensitivity. In the ES/RDB architecture (Figure 7), sensitivity is reduced from 23.6 to 13.0; in the ES/RBES architecture, sensitivity is reduced to 3.1. Caching is effective because fewer calls have to be made to access data across the high-latency path. Why is caching more effective in the ES/RBES architecture than

in ES/RDB? The reason has to do with the way that the combined-servers (ES/RDB using cached EJBs) and split-server (ES/RBES) architectures commit a transaction. The combined-servers configuration requires multiple database server accesses, one per memento image. Assuming no cache misses, the split-server configuration requires only a single access to the back-end server. This access is done at commit time in order to transmit the set of memento images involved in the transaction. Of course, the back-end server will, in turn, perform multiple accesses to the database server. However, these occur over a low-latency path. In contrast, the combined-servers configuration has large delays between the cache-enhanced application server and the database server. In consequence, the extra round-trips incurred when a transaction commits dominates the extra address-space crossing required by the split-server configuration.

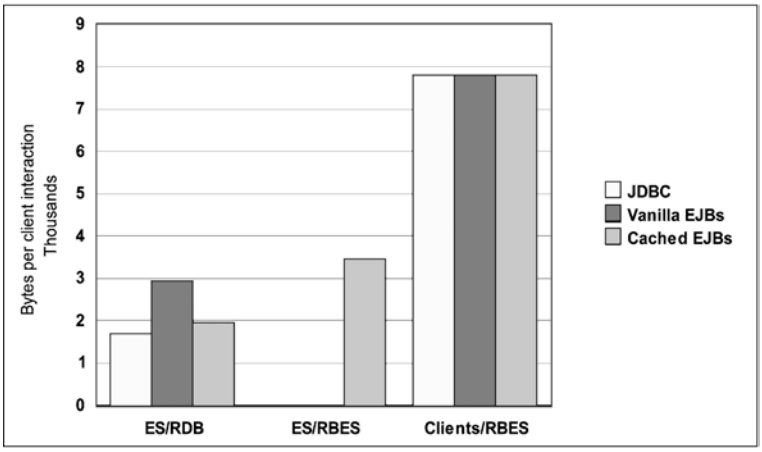

Fig. 8. Bandwidth

Why do the edge-server architectures, even using EJB caching, have greater latency than the remote data-center architecture? With the transactional caching algorithms that we have examined, at least one call to the database or back-end server is required for each commit operation. In Trade2, each client request involves at least one commit operation, because all client requests require access to some transactional data. Therefore, each client request involves at least one round-trip call to the back-end server, and possibly more calls to handle cache misses. These transactional caching algorithms cannot yield an edge-server with lower latency than a non-edge-server configuration. Although we use optimistic concurrency control (Section 2), the use of pessimistic concurrency-control algorithms will not improve the situation. Pessimistic concurrency control requires at least as many calls to acquire and release locks at the back-end server.

We do not claim that non-edge-servers will always supply lower-latency than edge-servers for transactional applications. Other applications may not require

access to transactional data on every request. For example, workflow techniques could batch the commit of multiple client requests as a single transaction.

Although latency performance suggests that the non-edge-server architecture is best suited for transactional applications, Figure 8 shows the weakness of this architecture. We see that every client/server round-trip transmits more than 7000 bytes to the back-end server, while the edge-server architectures transmit far fewer bytes. ES/RBES transmits 3000 bytes and ES/RDB transmits 2000 bytes. These differences relate to one of the basic motivations for edge-servers: to reduce the amount of bandwidth that must be provisioned for the back-end server. In the Clients/RAS architecture, the presentation portion (HTML, images, JavaScript) of an application must all be transmitted on connections to the back-end server. (Because Trade2 does not contain images or static HTML, we expect that other applications would show an even greater "bandwidth effect".) In contrast, the edge-server architectures transmit this data on smaller, local pipes, between the clients and the edge-servers. Much smaller amounts of traffic needs to be transmitted to the shared site (back-end server or database).

As shown by Figure 6, using EJB caching on the ES/RBES architecture provides latency that is almost as good as Clients/RAS – while using much less bandwidth. We consider this configuration to be a superior compromise to optimize these two goals.

5 Summary

In this paper we examined the effectiveness of edge-server architectures for transactional applications. We showed that, in order to maintain transactional consistency, such applications require more interaction between edge-servers and the back-end server than non-transactional web-data. While this causes a non-edge-server architecture to have superior latency behavior than edge-server architectures, we showed that EJB-caching allows edge-servers to provide almost the same latency performance as the non-edge-server architecture, while providing much better bandwidth behavior.

Acknowledgements

We would like to thank Vikaram Desai, Jiwu Tao, and Michael Young (IBM Pittsburgh Lab) for their help in architecting and implementing an earlier version of the ejb caching framework.

References

1. K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On Space Management in a Dynamic Edge Data Cache . Fifth International Workshop on the Web and Databases (WebDB 2002). 2002.
http://www.db.ucsd.edu/webdb2002/papers/42.pdf

2. A Distributed Infrastructure for e-Business. http://www.akamai.com/en/html/services/white_paper_library.html. 2002.
3. OMG Specifications and Process. http://www.omg.org/gettingstarted, 2002.
4. Enterprise JavaBeans Specifications. http://java.sun.com/products/ejb/docs.html, 2002.
5. Edge Side Includes (ESI). http://www.esi.org/index.html, 2002.
6. WebSphere Edge Server. http://www-3.ibm.com/software/webservers/edgeserver/, 2002.
7. JDBC Data Access API http://java.sun.com/products/jdbc/, 2002.
8. M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. ACM Transactions on Database Systems. Vol. 22, No. 3. September 1997. 315-363.
9. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications. 15(3). 2001. www.globus.org/research/papers/anatomy.pdf. 200-222.
10. E. Gamma et al. Design Patterns. Addison Wesley Longman, Inc. 1995.
11. J. Gray. A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. 1993.
12. J. Gwertzman and M. I. Seltzer. World Wide Web Cache Consistency. USENIX Annual Technical Conference. 1996. 141-152.
13. A. Leff, P. Prokopek, J. T. Rayfield, and I. Silva-Lepe. Enterprise JavaBeans and Microsoft Transaction Server: Frameworks for Distributed Enterprise Components. Advances in Computers, Academic Press. Vol. 54. 2001. 99-152.
14. A. Leff and J. T. Rayfield. Improving Application Throughput with Enterprise JavaBeans Caching. May 2003. 23rd International Conference on Distributed Computing Systems.
15. Q. et al Luo. Middle-tier Database Caching for e-Business. Proc. ACM SIGMOD International Conference on Management of Data, 2002.
16. M. Rabinovich. O. Spatscheck. Web Caching and Replication. Addison Wesley Professional, 2002.
17. Java Remote Method Invocation (RMI). http://java.sun.com/docs/books/tutorial/rmi/, 2002.