To appear in the Proceedings of the International Conference on Distributed Computing Systems (ICDCS), 2003.

Efficiently Distributing Component-Based Applications Across Wide-Area Environments

Deni Llambiri, Alexander Totok, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY, USA
{*llambiri,totok,vijayk*}*@cs.nyu.edu*

Abstract

Distributed deployment of network applications in wide-area environments has proven effective for improving end-user experience. Another trend is the use of component frameworks for building network services. Their component-based nature makes such applications natural candidates for distributed deployment, but it is unclear if the design patterns underlying component frameworks also enable efficient service distribution.

In this paper, we investigate the application design rules and accompanying system-level support essential to a beneficial and efficient service distribution process. Our study targets the widely used Java 2 Enterprise Edition (J2EE) component platform and Java Pet Store, a sample component-based e-commerce application.

Our results present strong experimental evidence that component-based applications can be efficiently distributed in wide-area environments using a small set of generally-applicable design rules for orchestrating interactions and managing component state. We additionally discuss enforcement of these rules, and their automated implementation by container frameworks.

1. Introduction

The Internet currently provides access to a variety of sophisticated network-accessible services such as e-mail, banking, on-line shopping, and entertainment. Given their increased functional and implementation complexity, and the necessity to access distributed sources of data, these services are often realized as distributed applications.

The design and development of such applications exhibits two major trends. First, these applications are increasingly being built on top of commercial-off-the-shelf (COTS) component middleware frameworks. Industry-standard frameworks, exemplified by OMG's CORBA [2], Sun Microsystems' Java 2 Platform Enterprise Edition (J2EE) [27], and Microsoft's .NET [20], permit assembly of services from reusable components, relying upon container environments to provide commonly required support for naming, communication, security, clustering, persistence, and distributed transactions. In addition to providing an integrated environment for component execution, which significantly reduces the time to design, implement, and deploy applications, such frameworks incorporate "best practices" designs. The latter provides developers with *design patterns*, suggesting a standardized structure upon which distributed component-based systems should be based.

The second trend manifests itself in the fact that, generally speaking, application data and data processing in network services is being brought *closer to the clients*. This is being done in order to cope, on the network level, with the inherently bursty and unpredictable nature of traffic in wide-area environments, and, on the application level, with high-volume, widely varying, disparate client workloads. Examples of this approach vary from conventional *caching of static content*, to web content delivery using *content-distribution networks*, to systems such as Akamai's EdgeSuite [7] and IBM's WebSphere [28], which offload part of the data processing from web servers to *edge* servers.

In this work, we combine these two natural trends and explore the question of whether component-based applications can benefit from a distributed, edge deployment in wide-area environments. Distributed deployment brings several advantages. Cacheable components can be positioned in edge nodes, effectively bringing the service closer to clients, and thus improving not only client perceived latency, but also overall service availability since client requests can utilize several entry points into the service. Furthermore, specific "hot" components can be replicated and/or redeployed on-demand in new physical nodes in response to higher client loads or congested network links. However, despite these advantages, component-based applications are typically deployed only in a centralized fashion in high-performance local area networks. In the rare

cases when these applications are distributed in wide-area environments, the systems tend to be highly customized and handcrafted. One explanation for this situation is that there are no guidelines for how component-based applications should be engineered to enable efficient service distribution in heterogeneous and high-latency network settings.

In this paper, we address this issue by investigating the application design rules and accompanying system-level support required for a beneficial and efficient service distribution process. This study targets the Java 2 Platform Enterprise Edition (J2EE) component platform and Java Pet Store, a sample best-practices application that covers most aspects of the platform. We deploy Java Pet Store in a fixed, simulated wide-area environment, apply various design patterns and optimizations in an incremental fashion, and after each step measure the performance of the application and draw conclusions about the impact of the changes. Our approach focuses on *application-level design patterns* and *optimizations* and is orthogonal to other efforts that have looked at improving container-level mechanisms such as RMI performance [14].

While the overall performance of a network-accessible service usually depends on its component distribution and combined client load, response times observed by clients also significantly depend on *client behavior*, as the execution of different types of requests involve different sets of service components. To model this behavior, we introduce the notion of a *service usage pattern*, a frequently executed scenario of service invocation, which reflects typical client behavior. Considering different service usage patterns, first, helps to identify *which* groups of clients benefit most from certain service distribution and replication strategies, and second, provides an application deployer with knowledge of *how* applications should be deployed to satisfy the needs of certain client groups.

Our results present strong experimental evidence that component-based applications can be efficiently distributed in wide-area environments using a small set of generally-applicable design rules for orchestrating interactions and managing component state. Moreover, we argue that the burden of implementing some of the suggested functionality could be shifted from application programmers to container providers. Application deployers need only declaratively express the desired component behavior via (extended) deployment descriptors; the needed system and application level components could be automatically configured, instantiated and linked by container infrastructures.

Project Context The work presented in this paper is part of a bigger research effort, the *Mutable Services* project, which focuses on the construction of a *flexible service distribution infrastructure* for component-based applications for serving different groups of clients along different *access paths*. Each access path represents a different deployment of the service's components to underlying physical nodes, enabling the associated client group to receive differentiated service. By creating and controlling resources allocated to a particular path, the infrastructure permits a service to adapt to a broad range of "unfriendly" system conditions, including network congestion, bandwidth mismatches and high latency between client and server locations, as well as node and link failures.

The rest of the paper is organized as follows. Section 2 provides some background on the Java 2 Enterprise Edition component platform and Java Pet Store application. Section 3 describes our testing methodology. Section 4 introduces design patterns and optimizations one at a time and details how they were applied to the application, and outlines the impact of the changes by analyzing resulting performance. Section 5 describes how the identified design rules can be incorporated in component models and frameworks, and discusses how some of the proposed functionality can be automated by container environments. Related work is discussed in Section 6 and we conclude in Section 7.

2. Background

Java Pet Store [24] is a best-practices sample J2EE application from Sun Microsystems' Java BluePrints program. It represents a typical e-commerce application – an on-line store – and covers the most important features of the J2EE component platform in a relatively small application. This paper refers to Java Pet Store version 1.1.2.

Java 2 Enterprise Edition The J2EE platform supports the development of network applications, whose functionality is decomposed across three tiers - Web, Enterprise JavaBean (Business Logic) [8], and Data. J2EE components across these tiers fall into two main categories: stateless and stateful. Stateless components are exemplified by (synchronous) stateless session beans and (asynchronous) message-driven beans, and typically provide generic application-wide services. Since they do not contain any state, the replication of stateless components is rather straightforward. Stateful components can be classified into two categories: those that hold session state on a per-client basis, thus effectively acting as an extension of the client's run-time environment on the server-side, and components that represent shared state corresponding to the domain layer of the application. In the J2EE realm, the first category is exemplified by web components - servlets and JavaServer Pages (JSP) - that hold HTTP session information, and stateful session beans that offer improved scalability and transactional awareness, whereas the second category consists of entity EJBs. Generally speaking, session-oriented stateful components tend to be in-memory objects, whereas shared stateful components are transactional, persistent entities that are typically co-located with database servers. Since stateful session components are not shared

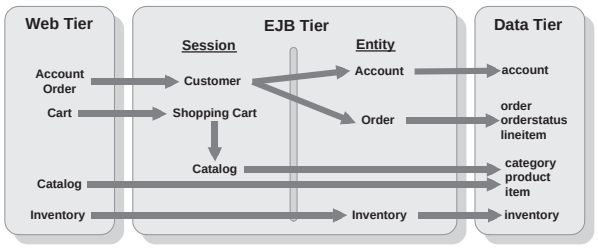

Figure 1. Pet Store component architecture.

they can be deployed in edge servers for better locality. We discuss our approach for replicating shared stateful components later in the paper.

Java Pet Store The fundamental design pattern used in Java Pet Store is well-known *Model-View-Controller* (MVC) architecture [26], which decouples the application's data structure, business logic, data presentation, and user interaction.

The *Model* represents the structure of the data in the application, as well as application-specific operations on those data. Java Pet Store stores application data across all three tiers - Web, EJB, and Data. The web tier uses servlet `HTTPSession` and `ServletContext` objects as well as JavaBeans accessed from JSP pages, to store client session state. In addition, the web tier directly manages JDBC connections to the database. In the EJB tier, state is maintained using stateful session beans and entity beans. The application also maintains persistent product, inventory, account, and order data in a relational database.

The *View* consists of objects that deal with presentation aspects of the application. The implementation of the view in Pet Store is completely contained in the Web tier, and is built on top of a reusable framework for web applications.

The *Controller* translates user actions and inputs into method calls on the Model, and selects the appropriate View based on user preferences and Model state. In Pet Store, the Controller is split between the Web and EJB tiers.

The main relationships among the most accessed Java Pet Store components are shown in Figure 1, and the most relevant EJBs to our experiments are listed in Table 1.

3. Methodology

We deployed Java Pet Store in a fixed, simulated wide-area environment, and applied various design patterns and optimizations in an incremental fashion. After each step, we measured its performance and drew conclusions on the impact of the changes.

3.1. Network topology

Our network topology aims at capturing a simple scaled-down wide-area distributed deployment of Java Pet Store. The system consists of two application servers (JBoss 2.4.4 [15]) and a single database server (Oracle 8.1.7 Enterprise Edition), each running on a dedicated 1GHz dual-processor Pentium III workstation. A wide-area network (WAN) separates the two application servers. One of the application servers is located in the same LAN as the database server, hence acting as the **main server** of the system. The other application server acts as an **edge server**. In addition, there are two client machines, one for each application server. The network topology was emulated by connecting all of the above nodes using a software router built with the Click modular router infrastructure [6]; traffic shaping components were used to simulate 100 ms latency each way in the WAN link.

Table 1. EJBs in Java Pet Store.

EJB Name	Description
Stateless Session Beans	
`Catalog`	Handles read-only queries to product database
`Customer`	Serves as a façade to Order and Account
Stateful Session Beans	
`ShoppingCart`	Maintains list of items to be bought by customer
`Sh.Cl.Contr.`	Manages model objects and processes events
Entity Beans	
`Inventory`	Records availability information for each item
`SignOn`	Keeps userid/password information
`Order`	Keeps order information
`Account`	Keeps account information

3.2. Service usage patterns

While designing this study, we observed that overall response times observed by clients depended not only on the application distribution and combined client load, but also on the *request type* since the execution of different requests involves different sets of service components. To model this behavior, we divide all clients into two *service usage patterns*: *Browser* and *Buyer*.

Browser This pattern represents a user that merely *browses* the application web site in search of items of interest. This user neither logs in, nor buys any products. A typical scenario for a browser would consist of a (relatively long) sequence of accesses to pages that present product-related information. Our tests use browser sessions of length 20, made up of individual page requests with the weights shown in Table 2. Each session is a logically organized sequence of requests starting with the *Main* page.

Buyer This pattern represents the behavior of a client who already knows what to buy. A buyer logs in, finds item(s) of interest, probably accessing a few product-related pages, puts desired items into the shopping cart, and checks out. For our tests, we organized buyer sessions as a sequence of pages emphasizing these activities: a buyer signs in, buys an item, and signs out (see Table 3).

Separating out these patterns is a core aspect of this

Table 2. Browser session.

Page	Functionality	Requests (%)
Main	Serves as an entry point to the application	5%
Category	Displays list of products associated with a particular category	15%
Product	Displays list of items associated with a particular product	30%
Item	Displays details about an item, including description, price, and the quantity in stock	45%
Search	Displays list of products, whose names match the specified search keyword(s)	5%

study, and was motivated by our efforts to accurately and meaningfully analyze our performance measurements. In general, knowledge of such usage patterns can not only help identify *which* groups of clients benefit most from certain service distribution and replication strategies, but also guide *how* the application should be distributed to satisfy the demands of a certain client group. The application configurations and measurements described in Section 4 embody both these points.

3.3. Client simulation

Client sessions were created by inserting a *soft delay* after each request to simulate user think time. Soft delays ensure a steady client load independent of response times.

Preliminary testing indicated that client response times did not depend on the relative ratio of browsers and buyers, but rather on the combined load coming from all clients. In all of our tests, we use a combined client load of 20 web page requests per second, coming from a mixture of 80% browsers and 20% buyers, equally divided between **Remote** and **Local** client machines. Each test lasted for one hour, preceded by 30 minutes of system "warm-up."

Some of the static content of Java Pet Store consists of 96 images totaling 318 Kbytes. During our tests, we did not send HTTP requests for these images, because in real-life environments web browsers and proxies tend to successfully cache such content.

3.4. Code modifications

Java Pet Store was not designed as a J2EE performance benchmark, so we made several modifications to define a fair baseline for our experiments. We increased the size of the database to allow testing of a greater number of concurrent users without contention for the data. Specifically, we added five artificial categories, 50 products and 300 items. We also removed unnecessary database requests and made other changes analogous to those made in another J2EE performance study [23]. Furthermore, we optimized all entity beans so that `ejbStore()` calls do not access the database at the end of read-only transactions.

This study focuses more on the performance impact of wide-area latencies on client response times, and less on the impact of specific application servers, web servers, and database servers (or combinations thereof). For this reason, we keep a modest load throughout all of our experiments, and do not overstress the servers.

Table 3. Buyer session.

Page	Functionality
Main	Entry point to the application
Signin	Prompts user to enter user ID and password
Verify Signin	System authenticates submitted credentials
Shopping Cart	Upon the user adding an item to the shopping cart, the updated cart content is displayed
Checkout	User initiates checkout process
Place Order	User confirms the order
Billing and Shipping	User confirms billing and shipping information
Commit Order	User commits order; all necessary database updates happen here
Signout	User signs out

4. Distributing Pet Store

Table 4 shows average response times per page for the five Pet Store configurations described below (for web page descriptions refer to Tables 2 and 3). Bold numbers indicate significant changes in performance, as compared to previous experiments.

4.1. Centralized Pet Store

In the first experiment, we ran the centralized undistributed version of Java Pet Store with the modifications noted above. In this configuration, which represents the low end of the distribution spectrum, the main server received all 20 HTTP requests per second and no requests were sent to the edge server. As seen in Table 4, accessing the service across a WAN link incurs approximately an extra 400 ms due to two round trips: one for TCP handshaking and another for the HTTP request (our tests did not use keep-alive HTTP connections).

4.2. Remote façade

The centralized configuration suffers from two major problems. First, the system does not utilize all its resources, since the edge server is not used. Second, HTTP requests from remote clients incur significantly higher response times in comparison to local client requests. Both of these problems can be addressed by migrating part of the application components into the edge server.

To start with, we deployed all of the Java Pet Store web components as well as the `ShoppingCart` and `ShoppingClientController` stateful session beans in both servers. However, we observed that doing so resulted in wide-area HTTP requests being substituted by multiple, more expensive, wide-area inter-component RMI calls. An additional problem results from the fact that the *Category*,

Table 4. Average response times (in ms) for five Pet Store configurations.

Configuration	Client	Browser					Buyer								
	Page	Main	Categ	Prod	Item	Search	Main	S/in	Verif	Cart	Ch/out	Pl.Or.	Bill	Commit	S/out
Centralized Pet Store	**Local**	87	95	94	88	106	98	78	89	120	76	70	70	158	90
(section 4.1)	**Remote**	488	492	492	486	496	489	480	482	658	477	646	482	708	447
Remote façade	**Local**	64	78	80	72	82	61	52	63	85	54	51	54	134	54
(section 4.2)	**Remote**	**72**	387	389	373	384	**60**	**54**	630	407	**61**	**57**	**61**	500	**63**
Stateful component	**Local**	55	82	84	55	77	60	51	65	77	53	50	55	**584**	54
caching (section 4.3)	**Remote**	55	394	390	**57**	393	68	52	629	**80**	50	49	53	**950**	62
Query caching	**Local**	56	50	51	54	87	58	51	61	70	50	50	54	614	52
(section 4.4)	**Remote**	55	**51**	**51**	55	481	61	49	638	69	51	52	53	966	54
Asynchronous updates	**Local**	61	54	53	57	92	61	53	64	75	53	53	56	**195**	56
(section 4.5)	**Remote**	59	51	53	58	459	59	48	632	69	50	50	50	**536**	52

Product, *Item* and *Search* pages present product information to end users, retrieving information from the Product database directly via JDBC. The lifecycle of opening, managing, and properly recycling database connections, as well as traversing query results demands verbose communication with the database server, resulting in overwhelmingly degraded performance when the web tier and database are separated by a high-latency network.

Both these problems result from an application structure that relies on fine-grained invocations of core components (such as entity EJBs and the data sources) from front-end components in the web layer. In addition to the performance disadvantages described above, such as structure contributes to less maintainable, less reusable, and tightly coupled code. Fortunately, both of these concerns can be alleviated using a simple, straightforward design pattern, where the domain model, typically implemented as a collection of possibly related entity beans, is wrapped with a new thin layer of *façade objects* [5, 19]. Clients, which only have direct access to the façade, can delegate execution of use cases in just one network call to the remote façade. The latter in turn can efficiently execute the use case performing possibly multiple local calls against co-located domain objects. Besides reducing the number of remote method invocations, the façade provides a single entry point into the domain model, enabling improved transactional and security control. The pattern does not suggest a singleton façade responsible for the entire application; instead, multiple façade objects should be created to serve collections of related use cases. For the pages used in our experiments, we rewrote the application code so that every page incurs no more than one RMI call to shared components. The only exception is the *Verify Signin* page, which makes two RMI calls, one to create a `Customer` session bean for the customer that logged in, and another for retrieving the customer's profile for future use. Figure 2 illustrates an example of the use of the façade pattern for the `Catalog` bean.

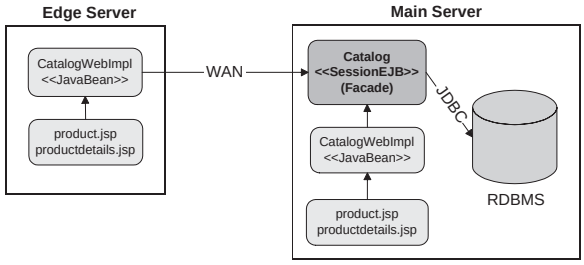

Figure 2. Remote Façade.

Façades also address the remote database access problem by instead directing client requests to an object that is co-located with the database server. In our case, we substituted all direct database accesses from the web layer with calls to the `Catalog` bean that served as a façade.

Average client response times for this application configuration (shown in Table 4) make the following points:

- Many pages (HTTP requests) can be served completely using only session information stored in the edge server. In particular, six out of nine buyer page requests can be served locally.
- RMI can require more than one round trip for a single method invocation, which slightly diminises the benefits of the façade pattern. It has already been pointed out that this is mainly due to ping packets and distributed garbage collection [25].
- The response times of local clients improve because of better load distribution.

4.3. Stateful component caching

The previous configuration improves locality and load distribution by deploying session-oriented stateful components on both servers. However, it does not yield much benefit for pages that trigger invocations on shared stateful components. In the third configuration, we focus on these components, exemplified in Pet Store by entity EJBs.

Our experience suggests that entity beans are excellent at handling heavy, concurrent transactional access, but they can be quite inefficient when used as data caches. However, data locality is critical when it comes to efficient wide-area service partitioning. Fortunately, entity beans can be easily transformed into read-only data caches by minor modifications to their lifecycle definition. As a matter of fact, most application server vendors already support some form of read-only entity beans.

Such support typically consists of invalidating the read-only bean upon updates, forcing it to refresh itself by "pulling" data from the database. This approach works well in a local-area setting, where the communication overhead with the database is negligible, but results in unacceptable performance in the wide-area. To avoid opening and maintaining remote database connections, read-only beans can efficiently refresh their content by querying a remote façade upon the first business method call after the invalidation. Another approach would be to *push the updated state* to read-only beans as a parameter of the invalidation call. This push-based scheme has the major advantage that clients of read-only beans will always have local response times, which is not the case with the pull-based approach. At first sight, it might seem that since the push-based scheme is not demand-driven, it can result in sending superfluous updates. However, the number of RMI calls is the same in both cases, because the invalidation call has to be made anyway. In the push-based scheme, more data potentially is being transferred. Several simple and effective optimizations can be applied, to cope with this problem, such as: transferring only the changes instead of the entire bean's state (i.e., fields that were modified), and compressing large fields for better bandwidth utilization.

The above insights can be materialized in a version of so-called *Read-Mostly Pattern* [17] where transactional operations are sent to the read-write version of the bean, which is typically co-located with the data source; non-transactional read operations are handled locally by the read-only cache. In addition, upon write operations, the read-write components push the updates to the read-only beans. In this configuration we strive for zero staleness: read-write entity beans block while the update is pushed to the read-only beans, hence a read operation that arrives after a previous write has committed, will always read the correct value.

The following changes were made to Java Pet Store:

- Three new entity beans (read-write and read-only versions) were introduced: `Category`, `Product`, and `Item`. These beans implement functionality that was previously handled by the `Catalog` bean, which accessed the product database directly via JDBC.
- A blocking push-based update mechanism was implemented between read-write beans and their read-only counterparts. The updates make use of a remote façade so that each update incurs only one RMI call.
- The read-only beans and the `Catalog` bean were also deployed on the edge server. The edge `Catalog` bean also has a reference to the central `Catalog` bean, to delegate requests that cannot be served by read-only beans, such as aggregate queries.

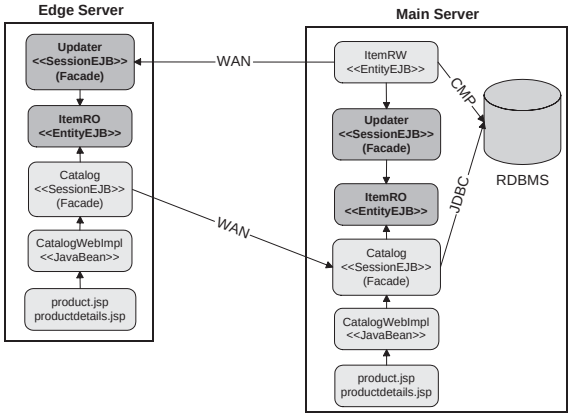

Figure 3. Stateful Component Caching.

Figure 3 shows a partial snapshot of the new component graph. Due to space limitations, the figure illustrates the read-mostly pattern only for `Item` EJB. Average response times for this configuration (see Table 4) support the following conclusions:

- Zero staleness for browsers penalizes buyers, since they have to block while the updates are being pushed across the wide-area to the edge servers. Specifically, the *Commit* page of the buyer session updates the `Inventory` bean, leading to higher response times for this page (as compared to previous configurations) for both local and remote buyers.
- Although the *Commit* page sees a higher response time, the average buyer response time is not affected as much because the *Shopping Cart* page is now served locally by the newly introduced read-only beans.
- The *Item* page of the browser session makes full use of read-only entity beans and so has local response time, but the other pages still need to go to the main server to execute aggregate SQL queries.

4.4. Query caching

Entity bean instances typically correspond to rows in a database table, implying that aggregate queries can only be executed by a relational database system. In Java Pet Store, as in most web-based e-commerce applications, aggregate queries constitute a fair part of application data retrievals, and hence *caching of query results* in edge servers can further reduce the number of remote invocations to centralized database servers.

A general problem with caching query results is determining which queries are affected by changes that occur to

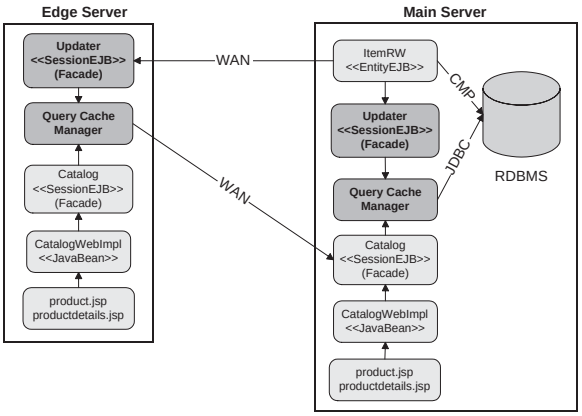

Figure 4. Query Caching.

the database. This is a well-researched problem [18] and we do not make any contributions to this field, nor try to incorporate any advanced query caching techniques in our experiments. Our focus is on the benefits of caching aggregate SQL query results at edge servers to avoid expensive trips to remote data centers. Such an optimization can be implemented in several ways. One way would be to use a demand-driven, pull-based update mechanism, whereby upon receiving the first read request after invalidation, the query cache manager gets the latest updates by re-executing the query in the remote database. Alternatively, a push-based protocol can be used that eagerly sends updates to the query cache manager. Unlike the pull-based approach, this scheme (1) does not penalize query readers because they never trigger requests to the remote database; and (2) involves small updates (typically single rows), making it easier to propagate only partial information [18] instead of resending the entire query result, effectively reducing bandwidth consumption.

We cache the results of two queries in Java Pet Store: the set of products for a given category, and the set of items belonging to a given product. These queries are heavily used by the *Category* and *Product* pages of the browser session. The query result cache was incorporated in the `Catalog` bean. For simplicity, we implemented the pull-based update mechanism for caching query results. However, the impact of invalidations is not visible in our test results, because the catalog of Java Pet Store is effectively read-only.

Figure 4 shows the realization of this optimization. Average response times for this configuration (refer to Table 4) support the following observations:

- As expected, query result caching lowers the remote browser's response times, but also has a local affect, since it reduces required database accesses.
- The *Search* page performs a keyword query, which is not cached, and hence it still incurs the cost of the remote call to the database façade.
- Buyer's performance does not improve because buyer still blocks on updates.

4.5. Asynchronous updates

Achieving zero staleness for browsers penalizes the buyer, who blocks while the update is propagated across the wide-area to the edge read-only beans. This approach suffers from severe scalability issues, since the response time for write operations is proportional to the number of edge servers times the number of individual fine-grained updates triggered by a single façade call.

Pushing updates in an *asynchronous* fashion eliminates this performance bottleneck. Upon transaction commit, updates are asynchronously pushed across the wide-area to the edge read-only components. But is the staleness of asynchronous updates acceptable? One could argue that even if the web tier components obtained the data from the transactional read-write version of the bean or the database, the information will likely be stale due to the user think time, and other concurrent server activity. However, there could be a problem if a client initiates a server-side update based on data that it has read in a previous transaction, since the update may be based on stale data. In such cases, where a use case can span multiple transactions, it is the responsibility of the application developer to ensure that the data used to update the server is not stale. In a sense, in most real-life scenarios the staleness of shared presentation data is unavoidable, and the asynchronous updates design optimization takes advantage of this fact to significantly improve response times.

The only change required to realize this pattern is to substitute the synchronous update façade with an asynchronous message-driven bean (MDB) façade that propagates updates to both read-only beans and query caches. The read-write beans publish their updates in a local topic, where multiple edge cache updaters are subscribed. This approach completely avoids the blocking problem and its scalability is limited only by the messaging middleware.

Figure 5 shows a partial snapshot of the component graph. Average response times for this configuration (shown in Table 4) make the following points:

- The most noticeable impact of asynchronous updates as compared to the previous configuration is improved buyer response time.
- The average response time for the local buyer remains slightly higher than that of the local browser since the buyer session makes less use of query caching and read-only beans.
- The remote buyer still incurs wide-area latencies in two of the nine pages since it requires access to shared components residing in the main server.

4.6. Summary

Figure 6 summarizes the results obtained from our tests. The last configuration achieves the best overall performance and scalability by accumulating all improvements. The

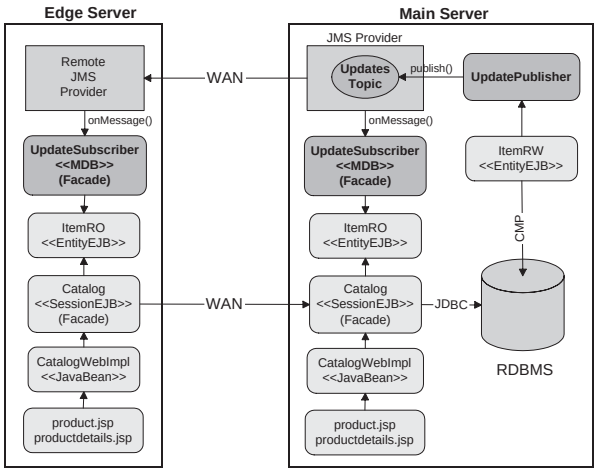

Figure 5. Asynchronous Updates.

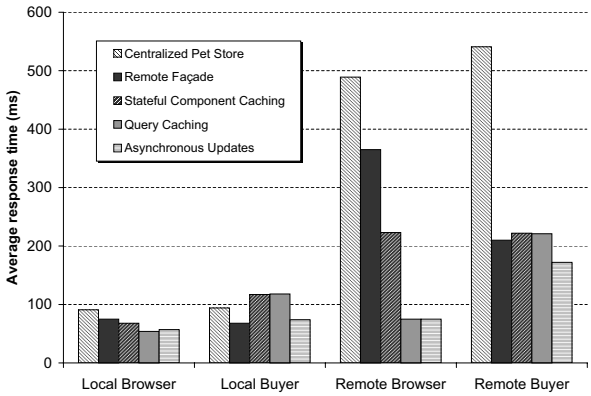

Figure 6. Session average response times.

façade pattern avoids unnecessary remote method invocations and implicitly defines the optimal application partitioning granularity. *Read-only entity beans* and *query caches* deployed in edge servers absorb the load generated by remote clients and save expensive trips to centralized data centers. *Asynchronous propagation of updates* achieves scalability and guarantees that updaters are not penalized by blocking on write operations.

The overall effect of applied design patterns and optimizations is two-fold. First and foremost, remote clients are almost completely insulated from wide-area effects. In the few cases when remote clients incur wide-area inter-component RMI calls, the communication overhead is as small as possible due to the façade pattern. Secondly, both local and remote clients experience improved performance due to aggressive caching of stateful components. Both these effects validate the current trend towards distributed deployment of network-accessible applications.

5. Framework Support for Design Rules

One of the major advantages of component-based development is the incorporation of "best-practices" design patterns as part of the component model, which *"forces"* the adaptation of proven and effective design techniques. In this section, we make several recommendations for incorporating the design rules that we applied to Java Pet Store and automating their implementation.

Enforcement of Design Rules An underlying theme of all the design rules advocated by this paper is to reduce communication overhead imposed by high-latency networks. The most important pattern available to designers and developers of distributed systems is the façade pattern, which minimizes superfluous remote calls between the edge and core tiers. Current-day systems employ various flavors of the façade pattern, such as: *synchronous* (implemented as session beans), *asynchronous* (implemented as message-driven beans). Based on careful analysis of the application requirements, developers should choose the most appropriate flavor of the façade pattern for the scenario at hand, as long as use cases that span several domain objects or other server-side resources are performed on behalf of the clients in one bulk remote call. Generally, the collection of a façade and its co-located, logically related domain entities constitutes the optimal partitioning granularity effectively serving as a *unit of distribution.*

An effective way to promote and enforce the use of the façade pattern is to define façades as the only components that can be invoked by remote clients. Furthermore, all other components present only local interfaces (as in EJB 2.0), so they can never be invoked remotely. If the component model enforces this recommendation, web (edge) tier components can never access core shared stateful components directly, a practice that leads to expensive and unnecessary remote calls.

Automating Pattern Implementation Whereas the correct implementation of the façade pattern largely remains the responsibility of developers, container environments can and should automate transparent caching of stateful shared components. One way of achieving this, and an approach we are pursuing, is to rely on (1) *extended deployment descriptors*, which specify desired behaviors, and (2) *general and flexible container environments* that implement required functionality.

Let us revisit the example of read-only entity beans optimization (read-mostly design pattern, section 4.3). The extended deployment descriptor in this case would specify whether the bean is deployed in a *read-write* or *read-only* mode, also identifying for the latter case, the updater read-write bean, the method of update (synchronous vs. asynchronous), and any application-specific relaxed consistency parameters [30]. The container infrastructure in turn would transparently link the read-write entity bean containers with the corresponding read-only containers to enable propagation of updates. Such automation frees developers from having to implement tricky update mechanisms

that require deployment of additional auxillary components such as message-driven beans and JMS topics (Section 4.5). Another advantage of this approach is that it allows flexible demand-driven (re)deployment of additional read-only beans in response to changing environment conditions, such as higher client loads.

The caching of query results can also be automated using a similar approach. The extended descriptors in this case would identify the queries to be cached, the invalidation mechanism, as well as operations (of possibly other components) that can cause query result invalidations or updates.

6. Discussion and Related Work

Table 4 shows that even when all of our optimizations are accumulated, transactional operations coming from remote clients still incur wide-area latencies because they have to access the main database server. Highly customized aggregate queries (such as keyword searches) also end up being executed in the database server, since their caching is typically ineffective. Both of these problems can be alleviated by orthogonal techniques that involve database partitioning and replication [29]. However, the main focus of this paper is on *lightweight* techniques for application partitioning and replication. In particular, unlike database replication, stateful component instantiation and (re)deployment can be done on-demand at run-time.

Although this work has focused on one sample application, and our conclusions, at first glance, may seem application specific, they are in fact applicable to a wide class of general purpose component-based applications. Java Pet Store covers most of the J2EE component platform and focuses on common ways of building J2EE component applications, so the vast majority of current-day commercial component-based applications share with it their architectural design and functional organization.

The identified application design rules are relevant for interactive scientific grid-based applications as well. These applications show several of the same characteristics as commercial component-based applications, typically including client-side remote instrumentation and visualization components, server-side data processing components, and back-end distributed repositories storing structured data. Ongoing efforts to integrate grid service frameworks with commercial web services standards, exemplified by the open grid-services architecture (OGSA) initiative [21], indicate strong support for this emerging trend.

Although this paper has focused on the static deployment of component-based distributed applications, our long-term goal is to enable *dynamic demand-driven* deployment of application components in response to changing environment conditions (load shifts, congested links, client behavior, and others). Existing component frameworks such as J2EE [27] and Microsoft .NET [20], and grid-service architectures such as Globus [11] and Legion [1] provide support for seamless interaction among distributed components, but as we have shown, do not offer much guidance on how to construct adaptive applications. Our work addresses this shortcoming by identifying common design rules yielding good wide-area performance for such applications.

The identified design rules themselves are related to previous work in two categories: application-level overlay networks and state replication in wide-area environments.

Application-level overlay networks Systems such as Overcast [13] and RON [4] have demonstrated the utility of application-level overlay networks for coping with the unpredictable characteristics of wide-area networks in the context of continuous media delivery and general traffic routing respectively. Similar benefits have also been achieved for web content delivery using *content-distribution networks*. Systems such as Akamai's EdgeSuite [7] and IBM's WebSphere [28] extend the latter to offload part of the processing from web servers to *edge* servers, relying upon emerging specifications such as ESI [9] and OPES [22]. Our work uses a similar notion of *edge* containers to perform application processing closer to the clients thereby potentially offering performance insulated from the characteristics of wide-area environments. However, in contrast to the application-specific solutions described above, our approach is applicable to a large set of general applications built using standard component frameworks.

State replication in wide-area environments Our identified design rules rely on efficient replication of application components to improve *data locality* and *responsiveness* for end users. Such replication appears similar, at first look, to the replication of stateful components already performed in current-day enterprise systems such as J2EE application servers (where stateful session EJBs are replicated). However, the latter is primarily done in a local scale for *failover* purposes, the application servers involved in the replication are tightly clustered together, and low-level LAN-specific mechanisms such as IP broadcast, are used to synchronize among the replicas. Such tightly-coupled approaches do not scale to wide-area environments, which requires scalable and efficient mechanisms for inter-component synchronization. In this regard, the design rules explored in our paper are more related to (and can leverage) other work on state replication in wide-area systems, examples of which include Bayou [16], which proposes an anti-entropy protocol for flexible update propagation between weakly consistent storage replicas, and TACT [30], which investigates tradeoffs between consistency, performance and availability of replicated services.

This paper extends ongoing efforts in our research group investigating application-neutral techniques for building adaptable general-purpose component-based distributed applications. Three of our previous systems — Application

Tunability [3], CANS [10], and Partitionable Services [12] — have looked at introducing adaptation functionality at the intra-component level, at the level of data streams flowing between static application components, and at the inter-component level. The approach outlined in this paper falls into the third category above, but differs in attempting to realize adaptation without requiring modification of application components by instead relying upon additional functionality in container environments and general-purpose auxillary system components.

7. Conclusion

This paper has investigated whether component-based applications, which represent a dominant trend for constructing network services, can be efficiently distributed and replicated in wide-area environments. In the context of the J2EE framework, we have proposed application design rules and accompanying system-level support essential to a beneficial and efficient service distribution process. Based on incremental modifications to the Java Pet Store sample application, we demonstrate that component-based applications can be efficiently distributed in wide-area environments if they adhere to a small set of identified design patterns and optimizations.

Finally, we argue that the burden of implementing some of the suggested functionality could be shifted from application programmers to container providers. With this support, application deployers need only declaratively express desired component behavior via generalized (extended) deployment descriptors, and needed system-level and application level components could be automatically instantiated, linked and configured by containers.

Acknowledgments

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, SPAWAR SYSCEN, or the U.S. Government.

References

[1] A. Natrajan et al. Capacity and capability computing using Legion. In *Proc. of the Int. Conf. on Comput. Science*, 2001.

[2] Object Management Group. *CORBA Components Specification. Version 3.0.* 2002.

[3] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4:49–62, 2001.

[4] D. Andersen et al. Resilient overlay networks. In *Proc. of the 18th Symp. on Oper. Syst. Princ. (SOSP)*, 2001.

[5] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] E. Kohler et al. The Click modular router. *ACM Trans. on Comp. Syst.*, 18(3):263–297, August 2000.

[7] Akamai Technologies Inc. Edgesuite services. `http://www.akamai.com/html/en/sv/edgesuite_over.html`.

[8] Sun Microsystems. *Enterprise JavaBeans Spec. Version 2.0.* 2001.

[9] Edge Side Includes (ESI). `http://www.esi.org/`.

[10] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services infrastructure. *3rd USENIX Symp. on Internet Technologies and Systems*, 2001.

[11] I. Foster et al. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `www.globus.org/research/papers.html`.

[12] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A framework for seamlessly adapting distributed applications to heteregeneous environments. In *Proc. Int. Symp. on Hign Perf. Distr. Comp. (HPDC)*, 2002.

[13] J. Jannotti et al. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI 2000*.

[14] J. Maassen et al. Efficient Java RMI for parallel programming. *ACM Trans. Prog. Lang. Syst.*, 2001.

[15] JBoss Open-Source Java Application Server. `http://www.jboss.org`.

[16] K. Petersen et al. Flexible update propagation for weakly consistent replication. In *Proc. of SOSP-16*, 1997.

[17] S. Kounev and A. Buchmann. Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In *Proc. of VLDB-28*, 2002.

[18] L. Degenaro et al. A middleware system which intelligently caches query results. In *Proc. of Middleware 2000*.

[19] F. Marinescu. *EJB Design Patterns*. John Wiley and Sons, New York, 2002.

[20] Microsoft Corporation. Microsoft .NET. `http://www.microsoft.com/net/`.

[21] Open Grid Services Architecture. `http://www.globus.org/ogsa/`.

[22] Open Pluggable Edge Services (OPES). `http://www.ietf-opes.org/`.

[23] Oracle Corporation. *Oracle9iAS J2EE Performance Study Results.* `http://otn.oracle.com/tech/java/oc4j/pdf/java_performance_results.pdf`.

[24] Sun Microsystems. *Java Pet Store Sample Application.* `http://java.sun.com/blueprints/`.

[25] S. Campadello et al. Wireless Java RMI. In *Proc. of the 4th Int. Enterpr. Distrib. Object Comput. Conf. (EDOC)*, 2000.

[26] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with J2EE Platform*. Addison-Wesley, 2001.

[27] Sun Microsystems. Java 2 Enterprise Edition. `http://java.sun.com/j2ee`.

[28] IBM Corp. Websphere platform. `http://www.ibm.com/websphere`.

[29] Y. Amir et al. Practical wide-area database replication. Techn. Report CNDS 2002-1, Johns Hopkins Univ., 2002.

[30] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. of the 4th Symp. on Oper. Syst. Design and Implem. (OSDI)*, 2000.