Efficiently Distributing Component-based Applications Across Wide-Area Environments

Technical Report: TR2002-832

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

Distribution and replication of network-accessible applications has been shown to be an effective approach for delivering improved Quality of Service (QoS) to end users. An orthogonal trend seen in current-day network services is the use of component-based frameworks. Even though such component-based applications are natural candidates for distributed deployment, it is unclear if the design patterns underlying component frameworks also enable efficient service distribution in wide-area environments.

In this paper, we investigate application design rules and their 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 two sample component-based applications: Java Pet Store and RUBiS.

Our results present strong experimental evidence that component-based applications can be efficiently distributed in wide-area environments, significantly improving QoS delivered to end users as compared to a centralized solution. Although current design patterns underlying component frameworks are not always suitable, we identify a small set of design rules for orchestrating interactions and managing component state that together enable efficient distribution. Futhermore, we show how enforcement of the identified design rules and automation of pattern implementation can be supported by container frameworks.

Keywords

Middleware, Component Frameworks, Internet Technologies, Wide Area Deployment, Application Design Patterns, State Replication and Caching, E-commerce Applications

1 Introduction

The role of the Internet has undergone a transition from simply being a data repository to one providing access to a variety of sophisticated network-accessible services such as e-mail, banking, on-line shopping, entertainment, and serving as a data exchange transport for applications utilizing the emerging Web Services platform. With the rapid growth of functional and implementation complexity of such services, and driven by their increasing necessity to access distributed sources of data, network-accessible services are often realized as distributed applications.

Two major trends can be observed in common ways of designing and deploying network-accessible services.

The first trend is the increasing popularity of commercial-off-the-shelf (COTS) component middleware as a platform for building distributed network-accessible applications. Current-day industry standard component frameworks, exemplified by OMG's CORBA Component Model [6], Sun Microsystems' Java 2 Platform Enterprise Edition (J2EE) [38] and Microsoft's .NET [28] frameworks, 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. The advantage of using component architectures is two-fold. First, component frameworks provide an integrated environment for component execution, as a result significantly reducing the time it takes to design, implement, and deploy systems. Second, a component model incorporates "best practices" designs, providing developers with *design patterns*, suggesting a standardized structure upon which distributed component-based systems should be based.

The second trend in the design and utilization of network-accessible services is, generally speaking, bringing application data and data processing *closer to the clients*. This is being done in order to cope, on the network level, with inherently bursty, unpredictable nature of Internet traffic, especially in wide-area environments, and, on the application level, with high-volume, widely varying, disparate client workloads. Examples of this approach vary from old-fashioned edge *caching of static content*, to web content delivery using content-distribution networks [1], to systems such as Akamai's EdgeSuite [13] and IBM's WebSphere [39], which offload part of the data processing from web servers to *edge* servers.

In this work, we try to 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. Though nominally suitable for deployment in distributed environments, 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.

The advantages of distributing and replicating components across wide area environments are several. 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. Although component frameworks offer mechanisms to enable distributed deployment of components, significant challenges need to be addressed before wide-area deployments of general applications becomes commonplace. The most basic challenge is: how should component-based applications be engineered to enable efficient service distribution in heterogonous and high-latency network settings?

In this paper, we answer this question 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 two sample applications that cover most aspects of the platform: Sun's Java Pet Store and Rice University's RUBiS. We deploy Java Pet Store and RUBiS 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 [20].

While the overall performance of a network-accessible service usually depends on its component distribution and combined client load, request response times observed by clients also significantly depend on *client behavior*, as different types of users tend to make different types of requests and, as a consequence, different sets of service components are involved in a request's execution. In this paper, we introduce the notion of a *service usage pattern*, a frequently executed scenario of service invocation, which reflects typical client behaviour. Considering different service usage patterns, first, helps to identify, which groups of clients benefit most from certain service distribution and replication, and second, provides an application deployer with the knowledge of *how* applications should be distributed and/or replicated, in order to be *adapted* to 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 would need only declaratively express the desired component behavior via (extended) deployment descriptors; and 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, that focuses on the construction of a *flexible service distribution infrastructure* for component-based applications, which permits different groups of clients to access the service via different *access paths*. Each of the access paths represents a different deployment of the service's components to underlying physical nodes, enabling the associated client group to receive differentiated service. Thus, the mutable services 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 in component software technology, Java 2 Enterprise Edition component platform, and the Java Pet Store and RUBiS sample applications. 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 test applications, 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

2.1 Component Software and Component Frameworks

Component software Traditional software development can broadly be divided into two camps. At one extreme, a project is developed entirely from scratch, with the help of only programming tools and libraries. At the other extreme, everything is "outsourced," in other words, standard software is bought and parameterized to provide a solution that is "close enough" to what is needed.

The concept of *component software* represents the middle path, where an entire application is *assembled* from individual atomic components, developed by third parties. Although each component is a standardized product, with all the advantages that brings, the process of component assembly allows the opportunity for significant customization, thus avoiding the drawbacks of using standard monolitic software applications. In addition, some individual components can be custom-made to suit specific requirements or to foster strategic advantages.

Component frameworks In its early days most of the empasis in the development of component software was on the construction of individual components and on the basic "wiring" support of components, leading to specifications such as Java RMI and COM/DCOM. It was highly unlikely that components developed independently under such conditions would be able to cooperate usefully.

Inception of *component frameworks* was the most important step that lifted component software off the ground. A component framework is a software system that supports components conforming to certain standards and allows instances of these components to be "plugged" into the component framework. The component framework establishes environmental conditions for the components and regulates the interactions between them. This is usually done through *containers*, component holders, which often also provide commonly required support for naming, security, transactions, and persistence. Component frameworks provide an integrated environment for component execution, as a result significantly reduce the effort it takes to design, implement, deploy, and maintain applications. Current day industry component framework standards are represented by Object Management Group's CORBA Component Model [6], Sun Microsystems' Java 2 Platform Enterprise Edition (J2EE) [38] and Microsoft's .NET [28], with J2EE being currently the most popular and widely used component framework in the enterprise arena.

The architecture of component-based systems is often significantly more demanding than that of traditional monolitic integrated solutions. To make development of efficient component-based applications a feasible task, a component model incorporates "best practices" designs, and establishes them as a "norm of life" through repeated reuse, by providing developers with *design patterns*, suggesting a standardized structure upon which distributed component-based systems should be based.

2.2 J2EE, Java Pet Store, and RUBiS

Java Pet Store [34] is a best-practices sample application from the Java Enterprise BluePrints program maintained by Sun Microsystems. It represents a typical e-commerce application. Customers can browse through a catalog of products, select items of interest and place them in a shopping cart. Upon indicating readiness to buy what is in the shopping cart, the application displays a bill detailing prices and quantities. The customer can also create a permanent account with the on-line shop, which includes billing and shipping information. This paper refers to version 1.1.2 of Java Pet Store.

RUBiS (Rice University Bidding System) [35] is an auction site prototype modeled after the popular e-commerce web portal eBay.com [11]. RUBiS implements the core functionality of an auction web site: selling, browsing and bidding on items. Visitors can search through a catalog of items devided into several categories and belonging to different geographical regions. They can bid on items of interest, as well as put comments for other users. Users may also choose to sell an item, registering it and specifying several parameters, such as action duration, and initial, reserve and buy-now prices. All non-browsing activities require creation of a permanent account with the web site and logging in.

Java Pet Store aims at covering as much of the J2EE component platform as possible in a relatively small application. Its main focus is on design patterns and industry best practices that promote code and design reuse, extensibility, modularity, ease of maintenance, isolation of development tasks by skill sets, and decoupling of code-bases with differing rates of change. RUBiS, on the contrary, uses a small subset of J2EE technology, but it does so to efficiently emulate the behavior of an existing e-commerce web site. Therefore, together Java Pet Store and RUBiS represent a big class of component-based applications already widely employed in the industry. So we believe that conclusions drawn in this paper are relevant not only to Pet Store and RUBiS, but to a wide class of general purpose component-based applications.

Java 2 Enterprise Edition Java Pet Store and RUBiS demonstrate the use of the most common types of J2EE components, with a focus on Enterprise JavaBeans (EJB) [15]. Generally speaking, J2EE components 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 largely classified into two categories: those that hold session state on a per-client basis, thus effectively acting

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

as an extension of the client's run-time environment on the server-side, and components that represent shared state that typically corresponds 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, or their alternatives: direct Java Database Connectivity (JDBC) or stored procedures, Java Data Objects (JDO) [23], and proprietary Object/Relational mapping tools. 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 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 the *Model-View-Controller* (MVC) architecture [36], 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 state across three tiers - Web, EJB, and Data. The Web tier uses servlet `HTTPSession` and `ServletContext` objects as well as JavaBeans accessed from JSP pages. In addition, the web tier directly manages database connections. In the EJB tier, state is maintained using stateful session beans and entity beans, which implement business use cases, such as browsing the catalog, manipulating the shopping cart, changing the inventory, and others. 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 Java Pet Store is completely contained in the Web tier, and is built on top of a reusable framework for web applications.

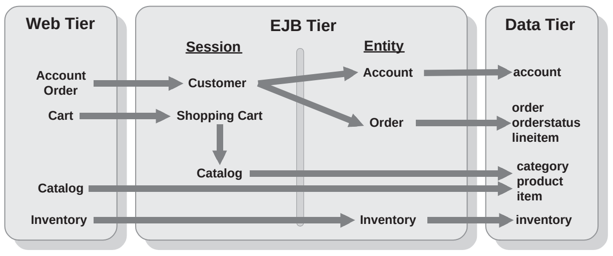

Figure 1. Pet Store component architecture.

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 the Java Pet Store application, 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. Other middle tier components such as `ProfileMgr`, `Mailer`, and `AdminClientController` were not used during our tests. All entity beans in Pet Store 1.1.2 are implemented using Bean Managed Persistence (BMP) since at the time, Container-Managed Persistence (CMP) was not standardized yet, and could have resulted in code that was not portable across different application servers. Later versions of Pet Store (starting from 1.3) conform to the EJB 2.0 specification and take advantage of both CMP and local interfaces.

RUBiS Originally RUBiS was developed at Rice University as part of a study investigating the combined effect of application implementation methods, container design, and efficiency of communication layers on the performance scalability of J2EE applications in a LAN setting [7]. Several implementations of the application were made, ranging from a servlets-only implementation, to one utilizing session and entity beans. Even in the most elaborate version (a.k.a. *Session Façade* configuration), which we took as a baseline for our experiments, RUBiS is significantly lighter weight than Java Pet Store.

The application design of this RUBiS version is rather streamlined: for each type of web page there is a separate servlet which, if necessary to generate the response HTML page, invokes business method(s) on associated stateless session bean(s), that in turn access related entity EJBs to get application shared data, stored in the database. Therefore the RUBiS component architecture is almost "linear": each servlet has reference(s) to dedicated stateless session bean(s) only (almost always just one), which have references to related entity beans only. The application does not keep per-client session state, so it neither keeps any (HTTP session) data in the Web tier, nor does it use stateful session beans in the EJB tier. The application keeps shared persistent data in the database and caches it in the entity beans.

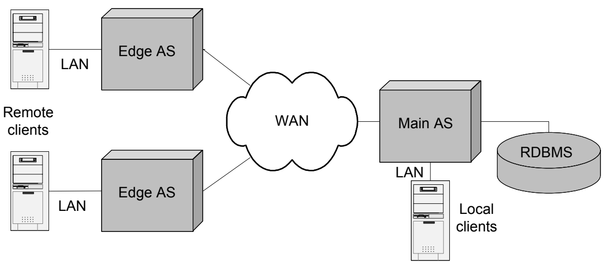

Figure 2. Network configuration.

As a consequence of such a design, there is no notion of "logged in" user-driven sessions in RUBiS, i.e., where the client logs in and performs an arbitrary sequence of activities that do not require further authentication, and logs out at the end of a session. In RUBiS, user authentication is required before each essential non-browsing client activity and covers only one such activity. Passing session and authentication information between related HTTP requests is implemented through hidden URL parameters.

Such a "minimal" design, however, turns RUBiS into a high-performance application and allows it to achieve good scalability, as seen in both the original RUBiS study [7] and our tests (section 4).

3 Methodology

We deployed both test applications in a fixed, simulated wide-area environment, and applied various design patterns and optimizations in an incremental fashion. After each step, we measured 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 the test applications. Even though the testbed configuration may seem not representative of actual WAN conditions, it is sufficient for demonstrating the importance of using identified design patterns and optimizations for efficient wide-area distribution of component-based applications.

The system consists of three application servers (JBoss 2.4.4 [22] bundled with Jetty 3.1.3 [24] web server and JSP container, for the Java Pet Store tests; and JBoss 3.0.3 with Jetty 4.1.0, for the RUBiS tests) and a single database server (Oracle 8.1.7 Enterprise Edition), each running on a dedicated 1GHz dual-processor Pentium III workstation. For the RUBiS tests, we used a MySQL 4.0.12 database running on the same workstation as one of the application servers. Preliminary tests showed that the database never became a performance bottleneck. Putting database on the same machine as the application server does not affect performance of the latter, because database operation rates produced by our tests were much lower than both

Table 2. Java Pet Store 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%

databases are designed to sustain. In all of the tests database CPU consumption never exceeded 5%.

A wide-area network (WAN) separates the three application servers. One of the application servers is located in the same LAN as the database server (runs on the same workstation, in the RUBiS tests), hence acting as the **main server** of the system. Two other application servers act as **edge servers**. In addition, 9 client machines were used to generate client load, three for each application server. Clients machines are collocated with the corresponding server (sitting on the same LAN), emulating client load coming from users "close" to that server. The network topology was emulated by connecting all of the above nodes using a software router built with the Click modular router infrastructure; traffic shaping components were used to simulate 100 ms latency each way in the WAN links, with 100 Mbit/s maximum combined network bandwidth (see Figure 2).

3.2 Service usage patterns

During preliminary testing of changes made to Java Pet Store and RUBiS, we came to the conclusion, that while overall performance of the application depended on its distribution and combined client load, response times observed by clients also significantly depended on *client behavior*, as different types of users tend to access different web pages and, as a consequence, different sets of service components are involved in a request's execution. In our subsequent tests we divide all clients between two different *service usage patterns*: *Browser* and *Buyer* for Java Pet Store, and *Browser* and *Bidder* for RUBiS. Intuitively, *Browser* corresponds to *read-only* activities, while *Buyer/Bidder* is involved in *read-write* sessions.

Java Pet Store Browser This pattern represents a user that merely *browses* the application web site in search of items of interest or with no particular goal. 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. During our tests, we used Java Pet Store browser sessions consisting of 20 requests to the pages described in Table 2. Each session is a logically organized sequence of requests starting with the *Main* page. For example, a request of an *Item* page always goes after a request for a *Product* page, such that the

Table 3. Java Pet Store 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

Table 4. RUBiS Browser session.

Page	Functionality	Requests (%)
Main	Static page; serves as an entry point to the application	2.5%
Browse	Static page; displays several browsing options	2.5%
All Categories	Displays the list of available categories	2.5%
All Regions	Displays the list of regions	2.5%
Region	Displays the list of available categories for a region	2.5%
Category	Displays the list of available items in a category	7.5%
Category & Region	Displays the list of available items in a category and a region	7.5%
Item	Displays details about an item, such as current price and number of bids	42.5%
Bids	Displays the list of bids on an item	15%
User Info	Displays public information about a user, such as e-mail, current rating and the list of user comments	15%

requested item belongs to the previously requested product.

Java Pet Store 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 them out. For the purpose of our tests, we organized Java Pet Store buyer sessions as a sequence of pages emphasizing buyer's essential activities: a buyer signs in, buys an item, and signs out (see Table 3).

RUBiS Browser This pattern represents, as in Java Pet Store, a user that merely *browses* the RUBiS web site and never bids on items. Our tests use RUBiS browser sessions of length 40, made up of individual page requests with the weights shown in Table 4. Each session is a logically organized sequence of requests starting with the *Main* page.

RUBiS Bidder Unlike in Java Pet Store, where there is only one type of *"write"* activity - buying an item, in RUBiS, a user can *bid* on an item and *put* a comment for another user. For our tests, we organized RUBiS bidder sessions as a sequence of pages emphasizing these activities: bidder bids on an item and leaves a comment for the seller of the item (see Table 5).

Table 5. RUBiS Bidder session.

Page	Functionality
Main	Static page; serves as an entry point to the application
Put Bid Auth	Prompts bidder to enter User ID and password to put bid on an item
Put Bid Form	After verifying user credentials, system displays the bidding form
Store Bid	The bid is accepted and stored in the database
Put Comment Auth	Prompts user to authenticate him/herself, to proceed with writing a comment
Put Comment Form	After verifying user credentials, system displays the form for writing a comment
Store Comment	The comment is accepted and stored in the database

Considering different *service usage patterns* is essential for accurate and meaningful performance measurement of a distributed component-based application for several reasons. First, it helps to identify, which groups of clients benefit most from certain service distribution and replication. Second, it provides application deployers with the knowledge of *how* applications should be distributed and/or replicated, in order to be *adapted* to the needs of certain client groups.

3.3 Client simulation

Clients were divided between *Browsers* and *Buyers/Bidders*. Each client ran a session consisting of a sequence of web page requests with delays inserted after each request to simulate user think time. We used *soft delays* between successive page requests in a session, i.e. instead of waiting a predefined `DELAY` time interval after receiving response from the previous request, the client waits for only `DELAY - response_time`. So effectively `DELAY` becomes the time interval between *sending* requests, which allowed us to simulate 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/bidders, but rather on the combined load coming from all clients. In all of our tests, we use a combined client load of 30 web page requests per second, coming from a mixture of 80% browsers and 20% buyers/bidders, equally divided between all client machines (10 HTTP requests per second coming from each of the three client groups). Each test lasted for approximately one hour, preceded by several minutes of system "warm-up," if needed.

Some of the application static content consists of 96 images totaling 318 KBytes in Java Pet Store, and 3 images (28 KBytes) in RUBiS. 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 Java Pet Store was not designed as a performance benchmark (ECPerf is the standard J2EE benchmark [12]) so we made several modifications that allowed us to measure the performance of our browser and buyer sessions. We increased the size of the database to allow testing 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 extra database requests and made changes to allow simultaneous database connections for users engaged in catalog browsing. Most of these modifications have been applied in another study [32]. Furthermore, we optimized all entity beans so that `ejbStore()` does not go to the database at the end of read-only transactions, and also removed an excessive database call which was present in `ejbFindByPrimaryKey()` methods. These modifications were aimed at providing a fair baseline for our experiments, and were not intended to turn Java Pet Store into a high performance application.

RUBiS As we already mentioned, RUBiS is a significantly more lighter weight application than Java Pet Store, and is already optimized to act as a performance benchmark. Therefore we made minimal changes to this application, mostly non-critical refactoring towards code unification and simplification:

- Entity beans moved from CMP 1.1 to CMP 2.0.
- Database schema slightly changed to avoid less efficient custom BMP queries. All queries were implemented through CMP-rendered entity bean home finders.
- Removed some unnecessary inter-component references and invocations.
- Some code optimizations were made, e.g., introduced static `String` fields instead of reading text from a file each time HTML header and footer is generated in servlets.

As in Java Pet Store, we increased the size of the database to avoid data contention. We added 400 users from 20 regions, selling 400 items belonging to 20 categories.

In this study, we are not interested on the impact that specific application servers, web servers, and database servers (or combinations thereof) have on overall performance. Our focus is on the impact that wide-area latencies have on client response time. Hence, we keep a modest load throughout all of our experiments, and do not overstress the servers. Throughout the tests, the CPU utilization on the machines running the JBoss/Jetty bundle never exceeded 40%.

4. Distributing Java Pet Store and RUBiS

Tables 6 and 7 show average response times per page for the five Java Pet Store and RUBiS configurations described below (for web page descriptions refer to Tables 2, 3, 4, and 5). Both **remote** group of clients (connecting to the **edge servers**) observed, as one would expect, practically equal response times, within a small error margin. Bold numbers indicate significant changes in performance, as compared to configurations appearing earlier in the table.

4.1 Centralized application

In the first experiment, we ran the centralized undistributed version of Java Pet Store and RUBiS with the modifications noted above. In this configuration, the main server got all 30 HTTP requests per second, whereas the edge servers were not used at all. This configuration represents the low end of the distribution spectrum, where effectively no distribution takes place. As seen in Tables 6 and 7, accessing the service from a WAN link incurs approximately an extra 400 ms, which is due to two round trips: one for TCP handshaking and another for the HTTP request (we did not use keep-alive HTTP connections throughout our tests).

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 being used at all. Second, HTTP requests going to the main server 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.

The second configuration in our experiments was obtained by deploying all web components (JSPs and servlets) and stateful session beans in all three servers. This configuration addresses the problems of the previous centralized configuration by making better use of available resources and also bringing some of the application components closer to remote clients. However, wide-area HTTP requests are now substituted by possibly multiple wide-area inter-component RMI calls.

In addition to contributing to less maintainable, less reusable, and tightly coupled code, repeated fine-grained invocations of core components, such as entity EJBs, from front-end components (web layer) add the overhead of multiple network calls, and reduce concurrency at the server-side, since transactions are effectively taking longer to complete. A superior alternative is to wrap the domain model, typically implemented as a collection of possibly related entity beans, with a new thin layer of *façade objects* [27, 10]. Clients, that have access only to the façade, can delegate the execution of use cases in just one network call to the remote façade, which in turn can perform multiple local calls needed to execute the use case against co-located domain objects. Besides reducing the number of remote method

Table 6. Average response times (in ms) for five Pet Store configurations.

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

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.

We discuss in more detail below, the modifications that were made to the test applications.

Java Pet Store Pet Store uses stateful session beans (`ShoppingCart` and `ShoppingClientController`). Together with web comonents they were deployed in all three servers. In the original Pet Store, *Category*, *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. Generally, such scenarios can be easily avoided by directing client requests to a façade 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. Furthermore, for all the pages used in our experiments, we rewrote the application code so that every page included in the experiment 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.

To further reduce the number of remote method invocations, we used the façade pattern in conjunction with caching of home and remote RMI stubs. Home stubs were always cached to avoid unnecessary trips to the local JNDI tree (*EJBHomeFactory* design pattern [27]). In the case of stateless remote façades, remote stubs were pooled as well

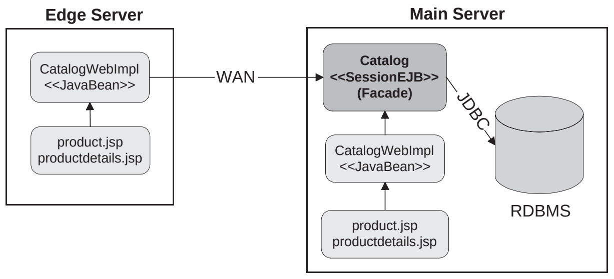
Edge Server
Main Server
CatalogWebImpl <<JavaBean>>
product.jsp productdetails.jsp
WAN
Catalog <<SessionEJB>> (Facade)
JDBC
CatalogWebImpl <<JavaBean>>
product.jsp productdetails.jsp
RDBMS

Figure 3. Remote Façade.

in the client side to avoid the penalty incurred by the RMI call that initially creates the remote stub.

Figure 3 illustrates an example of the use of the façade pattern for Pet Store (for brevity, in the rest of the paper, we will show such examples only for the Java Pet Store application).

RUBiS RUBiS does not use stateful session beans, so only web components were deployed in the edge servers. RUBiS required fewer code modifications because it already employed `Session Façade` design pattern, as manifested in the name of the RUBiS version, that we took for our experiments. Execution of use cases is delegated by web components to the façade session beans, collocated with the entity beans. We only made sure that there is only one RMI call from the web layer to the EJB layer in every servlet web page generation method. Most changes were done to implement the *EJBHomeFactory* design pattern. Servlets now cached remote stubs of stateless session beans, while the latter cached home stubs of related entity beans, to reduce unnecessary lookups in the JNDI tree of the main server.

Average client response times for this configuration for the two applications are shown in Tables 6 and 7. Several points stand out from the measurements of this configuration:

- Many pages (HTTP requests) can be served com-

Table 7. Average response times (in ms) for five RUBiS configurations.

Configuration	Cl.	Browser										Bidder						
	Page[1]	Main	Browse	All Categ	All Regions	Region	Category	Categ & Reg	Item	Bids	User Info	Main	Put Bid Auth	Put Bid Form	Store Bid	Put Comm Auth	Put Comm Form	Store Comment
Centr. RUBiS	L	14	12	33	26	35	43	21	27	40	43	12	13	32	36	13	25	35
	R	421	414	434	438	434	649	426	430	446	452	419	419	439	437	414	432	432
Remote façade	L	10	11	27	30	34	35	19	24	35	34	10	13	30	30	14	26	30
	R	**4**	**3**	424	407	399	499	265	275	300	379	**4**	**3**	408	284	**3**	284	282
St. comp. caching	L	13	16	29	32	39	38	23	19	30	31	10	15	23	**372**	14	22	**377**
	R	3	3	423	463	435	526	279	**7**	323	404	4	4	450	**680**	4	303	**628**
Query caching	L	9	12	12	15	17	16	12	15	16	16	9	10	15	377	9	16	374
	R	5	4	**7**	**7**	**7**	**6**	**5**	8	**8**	**8**	3	3	**7**	798	3	**6**	729
Async. updates	L	12	12	9	9	11	13	13	14	15	15	10	15	15	**32**	9	10	**34**
	R	4	5	9	7	6	6	4	7	10	10	5	4	9	**421**	4	12	**419**

pletely using only session information stored in the edge server. This is particularly prominent in the case of the Pet Store buyer, whereby six out of nine page requests can be served locally.

- If serving a page request from a remote client requires going to the main server, a wide-area HTTP request in this configuration is substituted by one inter-component RMI call. In this case the façade design pattern does not itself significantly improve request response time, but it makes it minimal, keeping the number of wide-area RMI calls as small as possible.
- RMI can require more than one round trip for a single method invocation. It has already been pointed out that these shortcomings are mainly due to ping packets and distributed garbage collection [5]. Generally speaking, the benefits of the façade pattern are slightly diminished since RMI can incur more than one round trip per method invocation.
- The response times of local clients went down due to better load distribution.

4.3 Stateful component caching

In the previous configuration, all session-oriented stateful components were deployed in both servers, improving locality and load distribution. However, pages that trigger invocations on shared stateful components did not gain much benefit from this approach. In the third configuration, we turned our attention to these shared stateful components, exemplified in J2EE by entity EJBs and relational database sources.

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. As a matter of fact, this is clearly manifested in the lifecycle and transactional management specifications of entity beans, and it simply reflects design choices made by the EJB architects. However, data locality is critical when it comes to efficient wide-area service partitioning. Fortunately, entity beans can be easily transformed into 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 with a timeout invalidation mechanism, and in some cases they also support a programmatic invalidation interface. Also, according to the EJB 2.0. specification [15], plans are under way to include some flavor of read-only entity beans as part of the standard.

Common to all the current approaches for updating read-only beans is that, upon invalidation, the read-only bean refreshes itself with the database using a pull protocol. This approach works well in a local-area setting, where the read-only bean communication overhead with the database is negligible, but as stated earlier, it 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 is being transferred,

[1]**L** – local client; **R** – remote client.

but this is a small price to pay for significantly improving the response time of remote clients. Furthermore, several simple and effective optimizations can be applied, 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. Moreover, in most cases the bandwidth problem is immaterial, since more than half of the data traffic incurred by RMI is due to distributed garbage collection [5].

The above insights can be materialized in a version of the so-called *Read-Mostly Pattern* [25, 40] 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 across the wide-area to the edge 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.

Java Pet Store The following changes were made to Java Pet Store in addition to the last façade configuration:

- Three new read-write entity beans 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.
- Read-only versions of `Category`, `Product`, `Item`, and `Inventory` beans were introduced.
- 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 `Catalog` bean delegates to the newly introduced entity beans.
- The read-only beans and the `Catalog` bean were also deployed on the edge servers. The edge `Catalog` bean also has a reference to the central `Catalog` bean. If a request that comes to the edge `Catalog` bean cannot be served locally by delegating to the read-only beans, it will be dispatched to the central `Catalog` bean, which is co-located with the database. For example, aggregate queries are always delegated to the central `Catalog` bean since they need to be executed in the database server.

RUBiS The following modifications, analogous to those of Java Pet Store, were made to RUBiS:

- Read-only BMP versions of `Item` and `User` beans were introduced.
- A blocking push-based update mechanism was implemented for read-only beans. The updates make use of a remote façade to ensure that each update makes only one RMI call.

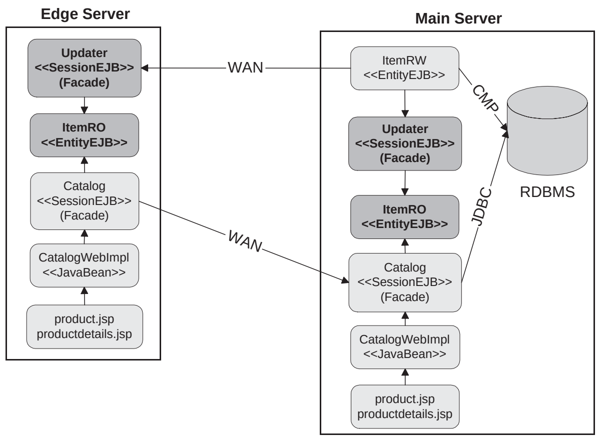
Edge Server
Main Server
Updater <<SessionEJB>> (Facade)
ItemRO <<EntityEJB>>
Catalog <<SessionEJB>> (Facade)
CatalogWebImpl <<JavaBean>>
product.jsp productdetails.jsp
WAN
WAN
ItemRW <<EntityEJB>>
CMP
JDBC
RDBMS

Figure 4. Stateful Component Caching.

- The read-only beans and `SB_ViewBidHistory`, `SB_ViewItem`, and `SB_ViewUserInfo` façade stateless session beans were also deployed on the edge servers.

Figure 4 shows a partial snapshot of the new component graph for Java Pet Store. Due to space limitations, the figure illustrates the read-mostly pattern only for `Item` EJB. Average response times for this configuration are shown in Tables 6 and 7. Several conclusions can be drawn from the measurements of this configuration:

- Zero staleness for browsers comes at a performance price for buyers/bidders, since they have to block while the updates are being pushed across the wide-area to the edge servers. More specifically, in Java Pet Store, the *Commit* page of the buyer session updates the `Inventory` bean and hence the response time for this page is significantly higher than in the previous configuration for both local and remote buyers. The same effect is seen for the *Store Bid* and *Store Comment* pages of the RUBiS bidder session.
- Even though the Pet Store buyer response time for the *Commit* page is higher, the overall average is not affected so much since the buyer's *Shopping Cart* page can be served locally due to the newly introduced read-only beans. In contrast, the RUBiS bidder average response time increased, because the bidder does not benefit from read-only beans, but needs to block on the *Store Bid* and *Store Comment* pages.
- The *Item* page of both Pet Store and RUBiS browser sessions 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.
- The response time for the Pet Store and RUBiS *Item* page is slightly improved for the local browser due to read-only bean caching versus database access.

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. The root of the problem is the well-known incompatibility (impedance mismatch) between object-oriented languages and SQL. In Java Pet Store and RUBiS, as in most web-based e-commerce applications, aggregate queries constitute a large part of application data retrievals, and hence *caching of query results* in edge servers can further reduce the number of remote method invocations to components that are co-located with centralized database servers. The benefits of caching query results in a local scale are less important because modern database servers are typically equipped with sophisticated query caching mechanisms, and possess all the information needed to make optimal caching decisions.

A general problem with caching query results is determining which queries are affected by changes that occur to the database. This is a well-researched problem [9, 37] and we do not make any contribution 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. The identification of invalidating operations can be left to application developers, and it should be possibly specified via deployment descriptors. A straightforward implementation 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. This scheme has the following benefits over the pull-based approach: (1) query readers are not penalized, because they never trigger requests to the remote database; (2) updates are typically small (usually involving single rows), hence making it easier to propagate only partial information [9] instead of resending the entire query result, effectively reducing bandwidth consumption.

Java Pet Store We cache the results of two queries in the Java Pet Store application: 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, and hence caching them in the edge server avoids remote method invocations to the main server. The query result cache was incorporated in the `Catalog` bean. This incorporation conforms with the EJB specification, because query cache is *soft* state, which is allowed to be kept in stateless session beans. 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 read-only.

RUBiS We implemented caching of all queries involved in the processing of all requests in our browser and bidders sessions. The query result caches were naturally incorporated in those stateless session beans that make corresponding finder method invocations (queries) on entity bean home interfaces. A push-based query update mechanism was implemented, and it makes use of the *remote façade* design pattern, namely updates to read-only beans and query caches are made in one bulk RMI call from the main server.

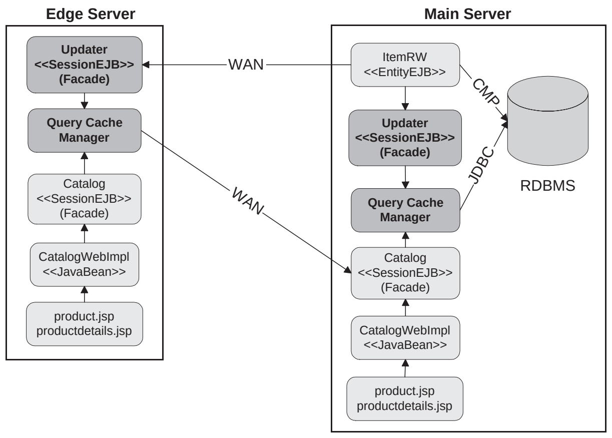
Edge Server
Main Server
Updater <<SessionEJB>> (Facade)
Query Cache Manager
Catalog <<SessionEJB>> (Facade)
CatalogWebImpl <<JavaBean>>
product.jsp productdetails.jsp
WAN
WAN
ItemRW <<EntityEJB>>
CMP
JDBC
RDBMS

Figure 5. Query Caching.

Figure 5 shows relevant components deployed on the main and edge servers, for the Java Pet Store application. Average response times for this configuration are shown in Tables 6 and 7. The following observations can be made from the measurements of this configuration:

- As expected, query result caching lowers both Pet Store and RUBiS remote browser response times. This is especially seen in the triumphal performance of RUBiS remote browser, now indistinguishable from the local browser. Also query caching has a local affect, since it reduces required database accesses.
- The Java Pet Store *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.
- Pet Store buyer's and RUBiS bidder's performance does not improve because they still block on updates.

4.5 Asynchronous updates

Achieving zero staleness for browsers penalizes the buyer/bidder, who blocks while the update is propagated across the wide-area to the edge read-only beans. This approach also suffers from severe scalability issues, since the response time for write operations is proportional to the number of individual fine-grained updates triggered by a single façade call. As a matter of fact, this is the case with the Pet Store buyer's *Commit Order* page, which causes writes to the `Inventory` EJB for each item in the user's shopping cart. This negative effect is not noticeable in our test results, because we never put more than one item in the shopping cart.

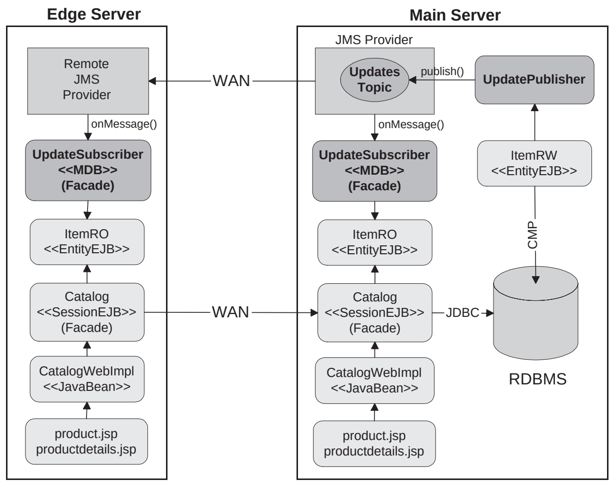

Figure 6. Asynchronous Updates.

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? Read-only beans and aggregate SQL query results typically contain data that is consumed by the web tier and displayed to the user in a tabular format. Even if the web tier components obtained this data from the transactional read-write version of the bean or the database, the information will likely be stale due to the incurred communication overhead, 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 (the "version number" design pattern can be used to handle such scenarios [27]). In a sense, 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 from the last configuration was 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 6 shows a partial snapshot of the Java Pet Store component graph. Average response times for this configuration are shown in Tables 6 and 7. Some remarks about the numbers follow:

- The most noticeable impact of asynchronous updates as compared to the previous configuration is improved Pet Store buyer and RUBiS bidder response times.
- The remote buyer/bidder still incurs wide-area latencies in some of the pages since it requires read-write access to shared components residing in the main server.

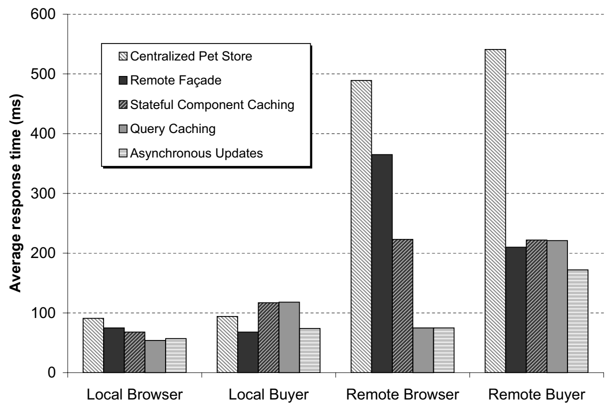

Figure 7. Java Pet Store session average response times.

4.6 Summary

Figures 7 and 8 summarize the results obtained from our tests. The last configuration achieves the best overall performance and scalability by accumulating all improvements. The *façade* pattern avoids unnecessary remote method invocations and implicitly defines the optimal application partitioning granularity. The use of this pattern is required if communicating components are separated by a wide-area network, regardless of the nature of user requests served by these components. *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 Design Rules Enforcement and Pattern Implementation Automation

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

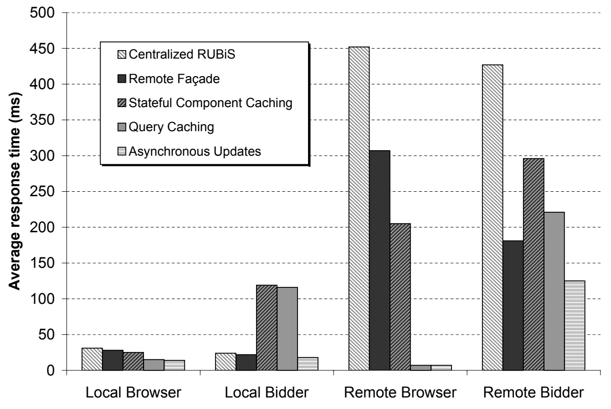

Figure 8. RUBiS session average response times.

adaptation of proven and effective design techniques. In this section we make several recommendations for incorporating the design rules and automating implementation of the design patterns that we applied to Java Pet Store and RUBiS.

Design Rules Enforcement 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 have only local interfaces (as in EJB 2.0) so that 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, which is a practice that leads to expensive unnecessary remote calls.

Pattern Implementation Automation 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. The system infrastructure for this purpose should consist of (1) *extended deployment descriptors* specification and (2) *general and flexible container environments* supporting this specification and implementing its functionality.

Let us revisit the example of read-only entity beans optimization (read-mostly design pattern, section 4.3). The extended deployment descriptor of an entity bean should specify whether the bean is deployed in *read-write* or *read-only* mode. In the latter case, the deployment descriptor should identify the updater read-write bean and the method of update (synchronous vs. asynchronous). Any application-specific relaxed consistency parameters [41] should also go here. The container infrastructure in turn should transparently link the read-write entity bean containers with the corresponding read-only containers to enable propagation of updates. As a result of this automation, developers are freed from implementing tricky update mechanisms that require the deployment of additional auxiliary 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 by container infrastructures. Currently, EJB containers do not support query caching, as that is typically left to database servers. Even though it is natural to let the database server transparently handle query caching, this approach does not improve data locality across wide-area environments. The problem is exacerbated due to the so-called "`n+1` database calls problem" [27], which reflects the fact that with BMP and certain CMP implementations, executing a single aggregate query that returns `n` rows could require `n+1` database calls. Due to the unacceptable incurred overhead caused by impedance mismatch issues, it is desirable to have separate containers for handling read-only aggregate queries. These containers should handle query result caching and invalidation transparently using application-specific information from extended deployment descriptors. This information should identify the queries to be cached and the invalidation mechanism. Moreover, operations (of possibly other components) that cause query result invalidations/updates should be specified as well.

Container support for query caching in J2EE is simplified with the introduction of *EJB Query Language* (EJB QL) in EJB Version 2.0. EJB QL is an SQL-like query specification language for the finder methods of entity beans with CMP. In the deployment descriptor, an application developer specifies the query structure in an abstract fashion using EJB QL, and the actual SQL code is generated by the EJB container. This opens us the immediate possibility for CMP containers to perform smart query caching, as the data structure of queries and corresponding entity beans is explicitly exposed to the container.

Placing containers caching read-only queries in edge servers can reduce the number of remote method invocations whose sole purpose is to reach centralized database servers. Our experiments with Java Pet Store and RUBiS show that even straightforward, non-optimized caching of a few "hot" queries can make a noticeable impact on overall application performance.

6 Discussion and Related Work

As seen from Tables 6 and 7, even in the last Pet Store and RUBiS configurations, which achieve the best overall performance by accumulating all improvements, 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 in Java Pet Store) 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 [2]. 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.

In this paper we took Java Pet Store and RUBiS as examples of component-based applications and showed that they can be efficiently distributed in wide-area environments. Even though we focused only on two sample applications, and our conclusions, at first glance, may seem somewhat application specific, they are, actually, applicable to a wide class of general purpose component-based applications. We believe so, because Java Pet Store aims at covering as much of the J2EE component platform as possible and focuses on presenting design patterns and industry best practices of building J2EE component applications, while RUBiS is modelled after an existing popular e-commerce web site. So the vast majority of current-day component-based applications share in some way, their architectural design and functional organization with Java Pet Store or RUBiS.

Despite this study's focus on commercial applications, the identified application design rules are of equal importance for interactive scientific grid-based applications. Typical applications in the latter camp show several of the same characteristics as commercial component-based applications, consisting of 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 [30], 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 [38] and Microsoft .NET [28], and grid-service architectures such as Globus [17] and Legion [29] 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 three categories: application-level overlay networks, state replication in wide-area environments, and distributed grid steering and remote visualization applications.

Application-level overlay networks Systems such as Overcast [21] and RON [3] 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 [1]. Systems such as Akamai's EdgeSuite [13] and IBM's WebSphere [39] extend the latter to offload part of the processing from web servers to *edge* servers, relying upon emerging specifications such as ESI [16] and OPES [31]. 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 [33], which proposes an anti-entropy protocol for flexible update propagation between weekly consistent storage replicas, and TACT [41], which investigates tradeoffs between consistency, performance and availability of replicated services.

Distributed grid steering and remote visualization applications Our work is also related to efforts supporting distributed grid steering and remote visualization applications, which allow remote clients to interact with grid applications across high-latency, low-bandwidth network connections by introducing application-specific wrappers capable of caching and distilling data produced by the application. In contrast to such systems, examples of which include Active Frames [26], Active Streams [4], and MOSS [14], our work represents a more general application and adaptation model. Adaptation is achieved by partitioning existing application functionality across multiple wide-area containers as opposed to introducing new bridging components.

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 [8], CANS [18], and Partitionable Services [19] — 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

Two major trends can be observed in common ways of designing and deploying current day network-accessible services. The first is an increasing popularity of commercial-off-the-shelf (COTS) component middleware as a platform for building distributed network-accessible applications. The second trend is in bringing application data and data processing closer to the clients, in order to cope, on the network level, with inherently bursty, unpredictable nature of Internet traffic, especially in wide-area environments, and, on the application level, with high-volume, widely varying, disparate client workloads.

In this paper, we have tried to combine these two natural approaches and addressed the question of whether component-based applications can be efficiently distributed and replicated in wide-area environments to improve the quality of service delivered to end users. In particular, we have investigated application design rules and their accompanying system-level support essential to a beneficial and efficient service distribution process. We applied various design patterns and optimizations to the Java Pet Store and RUBiS sample applications in an incremental fashion, showing performance improvements and drawing conclusions after each step. Our test results present strong experimental evidence that component-based applications can be efficiently distributed in wide-area environments. More specifically, applications whose typical user sessions do not require heavy transactional access to centralized data and involve user think time can be engineered so that the cost of remote service accesses is absorbed by edge deployment of stateful session components and shared non-transactional caches.

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, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] Akamai Technologies Inc. `http://www.akamai.com/`.

[2] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical wide-area database replication. Technical Report CNDS 2002-1, Johns Hopkins University, Center for Networking and Distributed Systems, 2002.

[3] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient overlay networks. In *18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[4] F. Bustamante and K. Schwan. Active Streams: An approach to adaptive distributed systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[5] S. Campadello, O. Koskimies, K. Raatikainen, and H. Helin. Wireless Java RMI. In *Proceedings of the 4th International Enterprise Distributed Object Computing Conference (EDOC 2000)*, September 2000.

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

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, November 2002.

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

[9] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, April 2000.

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

[11] eBay Inc. `http://www.ebay.com`.

[12] Sun Microsystems. *ECperf Specification. Version 1.1.* 2001.

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

[14] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10 – 20, August 1998.

[15] Sun Microsystems. *Enterprise JavaBeans Specification. Version 2.0.* 2001.

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

[17] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `http://www.globus.org/research/papers.html`, 2002.

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

[19] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A framework for seamlessly adapting distributed applications to heteregeneous environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, July 2002.

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

[21] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

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

[23] Sun Microsystems. *Java Data Objects Specification. Version 1.0.* 2002.

[24] Jetty HTTP Server and Servlet Container. `http://jetty.mortbay.org`.

[25] S. Kounev and A. Buchmann. Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In *Proceedings of the 28th International Conference on Very Large Databases, (VLDB)*, August 2002.

[26] M. Aeschlimann et al. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833 – 1839, June 1999.

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

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

[29] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. Capacity and capability computing using Legion. *Proceedings of the 2001 International Conference on Computational Science (ICCS)*, 2001.

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

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

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

[33] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, pages 288–301, October 1997.

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

[35] ObjectWeb Consortium. RUBiS: Rice University Bidding System. `http://www.objectweb.org/rubis/`.

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

[37] IBM Research. *Smart Query Project.* `http://www.research.ibm.com/smartnetwork/smartquery.html`.

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

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

[40] BEA Systems. *WebLogic Server Documentation.* `http://edocs.bea.com/wls/docs70/`.

[41] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.