Application Specific Data Replication for Edge Services*

Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng
Laboratory for Advanced Systems Research
Department of Computer Sciences
The University of Texas at Austin
lgao, dahlin, nayate, zjiandan
@cs.utexas.edu

Arun Iyengar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
aruni@watson.ibm.com

Abstract

The emerging edge services architecture promises to improve the availability and performance of web services by replicating servers at geographically distributed sites. A key challenge in such systems is data replication and consistency so that edge server code can manipulate shared data without incurring the availability and performance penalties that would be incurred by accessing a traditional centralized database. This paper explores using a distributed object architecture to build an edge service system for an e-commerce application, an online bookstore represented by the TPC-W benchmark. We take advantage of application specific semantics to design distributed objects to manage a specific subset of shared information using simple and effective consistency models. Our experimental results show that by slightly relaxing consistency within individual distributed objects, we can build an edge service system that is highly available and efficient. For example, in one experiment we find that our object-based edge server system provides a factor of five improvement in response time over a traditional centralized cluster architecture and a factor of nine improvement over an edge service system that distributes code but retains a centralized database.

1. INTRODUCTION

The emerging edge services architecture for providing web services processes client requests at a collection of edge servers distributed across the network and near end users [1, 5, 8, 35, 38]. This approach minimizes communication across the wide area network during the processing of client requests in order to improve service availability and latency.

Improving availability and latency is crucial for business-critical e-commerce servers. Although some server vendors advertise "four-9's" (99.99%) or "five-9's" (99.999%) of availability, when network failures are considered, end-to-end service availability is often as low as two-9's (99%), meaning that an average web client cannot contact an average web server for about 15 minutes a day [15, 28, 44]. Furthermore, despite that Internet web server response times have been improved, human factors studies suggest that human productivity improves more than linearly as computer systems response times fall in the sub-second range [21].

Many systems for business logic (code) distribution and execution at edge servers have been built [2, 5, 8, 35, 38], but the core challenge, dynamic data distribution and consistency, still remains. The data on which edge servers operate must be consistently replicated for the edge servers to correctly deliver the services. Although web-scale replication is well understood for traditional caching, where all updates are made at a central server, replication and consistency for edge servers that can both read and write data is more challenging. Brewer [7] suggests that there is a fundamental *CAP dilemma* for data replication in large scale systems: systems cannot simultaneously achieve both high **C**onsistency and high **A**vailability if are subject to network **P**artitions. As a result, distributed code is used for caching and content assembly [1, 8, 35] but seldom used for replication of web services with a rich mix of reads and writes.

The goal of this project is to build an edge service replication architecture using application-specific distributed objects [36] for e-commerce applications. Standard e-commerce implementations allow business logic (e.g. servlets, Enterprise Java Beans, or CGI programs) to access the central databases directly. In the case where business logic is distributed, accesses to the central database become costly remote operations. Our edge service architecture replicates both business logic and data on edge servers. It encapsulates the shared data and manages the distributed state behind application-specific distributed object abstractions. As illustrated in Figure 1, we deploy business logic, distributed objects, a database, and a messaging layer on a set of distributed servers that are accessed by clients via standard HTTP front ends. The distributed objects interposes between business logic and the local database to control data access. They also communicate with other instances of the distributed objects through the persistent messaging layer [14, 22, 30] to manage data replication and consistency.

In this paper, we demonstrate that the edge services architecture is feasible for the TPC-W benchmark [34], which represents an online bookstore with functionalities including browsing, shopping, and ordering. To further explore data replication issues, our study also uses a variation of the TPC-W benchmark, called the distributed bookstore, that includes additional consistency constraints.

Our experiment suggests that although strong consistency and high availability are difficult to achieve for a completely general large-scale system using a generic database interface, the semantics of the specific shared objects needed by the distributed bookstore are relatively straightforward to provide. We identify five simple distributed objects to manage the consistency of different subsets of the distributed bookstore's shared data. The *catalog* object is used to maintain catalog information in our system. It exploits the fact that catalog updates take place at one place and are read at many others. We use the *order* object to collect finalized orders at

*This work was supported in part by the Texas Advanced Research Program, Cisco, and an IBM University Partnership Award. Dahlin was also supported by a Sloan Research Fellowship.

Copyright is held by the author/owner(s).
WWW2003, May 20–24, 2003, Budapest, Hungary.
AC1-58113-680-3/03/0005.

multiple locations and process them at the backend server. It takes advantage of the fact that many nodes write orders, but only one needs to read them as well as the fact that the requirement on update sequencing across nodes is loose. The *profile* object represents the user profile information in our system. It takes advantage of low concurrency of access to each record on multiple nodes, as well as field-specific reconciliation rules [32]. The *inventory* and *best-seller-list* objects are used for tracking a bookstore's inventory and best seller lists. In the case of inventory, the object exploits the fact that edge servers care about whether the inventory is zero but do not need to know the actual value. In case of the best seller list, the object takes advantage of the fact that a few purchases of a non-popular book do not necessarily put this book on the best seller list because the purchases of popular books can greatly exceed those of non-popular ones.

Encapsulating database access behind object specific interfaces yields many advantages. First, client requests are locally satisfied by distributed objects, which asynchronously manage the local database consistency. Thus, edge servers are able to continue to operate, even in the case when network partitions occur. Furthermore, because requests are satisfied locally at edge servers, the response time is better than that of the centralized system because we minimize trips to store or retrieve data at the central database. Second, each distributed object can make use of object specific strategies to replicate data and to enforce exactly the consistency semantics it requires. Third, distributed objects restrict data access to a narrower interface than a general database interface, which allows for simplifying assumptions in the objects' consistency protocols.

We construct and evaluate a prototype system based on Apache web servers, Tomcat Servlet engines, a JORAM implementation of the Java Message Service, and a DB2 database, and we find that it has excellent availability, consistency, and performance. Under this implementation, our edge servers approximate the ideal system, in which high speed and reliable links connect end users to service front-ends and connect service front-ends to backend databases. For instance, our system continues to process requests with the same throughput and response time before, during, and after a 50-second network partition that separates edge servers and the backend server. And the response time of our system is nearly 5 times better than that of the traditional centralized system, in which end users connect to web servers via slow WAN links.

Qualitatively, we find the application-specific consistency rules easy to build and understand. We speculate that this approach may be useful for engineering systems for two reasons. First, once developed, distributed objects encapsulate the complexity of data replication and provide simple interfaces for engineers to use to build edge services without worrying about the intricacies of consistency protocols. Second, when experts construct the distributed objects, the restricted interface makes it easier to build distributed objects with the ability to handle consistency than to write reconciliation rules [32] for generic database interfaces.

This paper's main contribution is to demonstrate that object-based data replication makes it easy to build a distributed e-commerce web service and thereby dramatically improve both availability and performance. Although we focus on TPC-W and the more demanding distributed bookstore benchmark in this study, we speculate that similar techniques might also apply to a broader range of applications. Some consistency optimizations we exploit are similar to some proposed in previous work [25, 26, 32, 36, 42], but our emphasis is on how to integrate these ideas and effectively apply them to make an important class of applications work.

In the rest of the paper, we first present the background information on the TPC benchmark W. Then, in Section 3, we discuss the design of our distributed bookstore application with the focus on the four distributed objects that enable data replication for the edge services. In Section 4, we conduct experiments with the TPC-W benchmark workload, primarily targeting system availability, performance, and consistency. We discuss other similar work in Section 5 and summarize our work in Section 6.

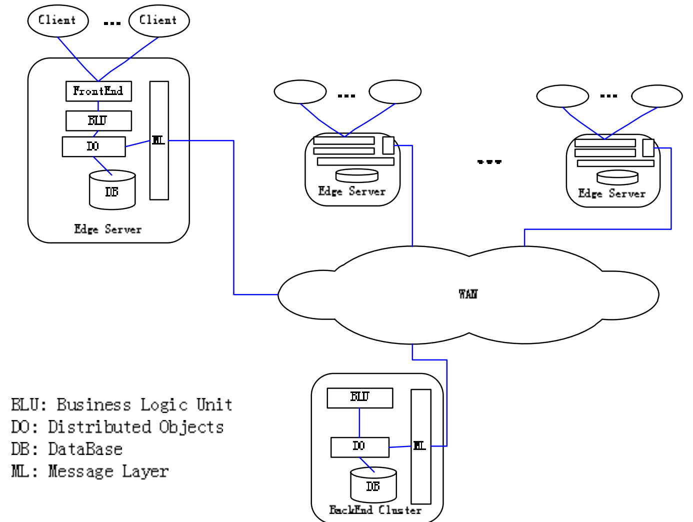


Figure 1: The edge services architecture diagram.

2. TPC-W BACKGROUND

TPC Benchmark W (TPC-W) is an industry-standard transactional web benchmark that models an online bookstore [34]. It is intended to apply to any industry that markets and sells products or services over the Internet. It defines both the workload exercising a breadth of system components associated with the e-commerce environment and the logic of the business oriented transactional web server. The benchmark defines activities including multiple concurrent online browsing sessions, dynamic page generation from a database, contention of database accesses and updates, the simultaneous execution of multiple transaction types, and transaction integrity (ACID properties). These are core demands in many e-commerce applications, but the weights of these activities may be different across applications.

The benchmark defines three *scenarios* (workload mixes): browsing, shopping, and ordering. The browsing scenario consists of a mix of 95% browsing interactions, such as searches and product detail displays, and 5% ordering interactions. The ordering scenario consists of 50% shopping interactions and 50% ordering interactions. For scalability measurements, the benchmark also defines the size of data entries in the backend database, which affects the performance of some interactions such as search. The defined data entries include the number of books in the database and the number of initial registered customers, as well as the number of book photos of different sizes.

The primary metric of the TPC-W benchmark is *WIPS*, which refers to the average number of Web Interactions Per Second completed. It is used for measuring the system throughput. Another metric is the Web Interaction Response Time, (*WIRT*), which is used for measuring the latency of the system.

3. SYSTEM DESIGN

3.1 Overall architecture

As Figure 1 indicates, our edge services architecture consists of

a backend cluster and a collection of edge servers distributed across the network. The common components on both edge and backend servers are business logic, a messaging layer, a database, and the distributed objects. Edge servers have an additional component, the HTTP front-end, through which clients access the service.

The edge services model works as follows. Clients use HTTP to access services through edge servers that are located near them. A number of suitable mechanisms for directing clients to nearby servers exist [1, 6, 16, 35, 41], and these mechanisms are orthogonal to our design. The HTTP front-end passes user requests to business logic units for processing and forwards replies from the business logic units (e.g. servlets, cgi, or ASP) to the end users. The business logic processes client requests on the edge server, and it stores and retrieves shared data using the interface provided by distributed objects. Each distributed object stores and retrieves data in the local database and also communicates with remote instances of the object in order to maintain the required globally consistent view of the distributed state. This distributed object architecture is similar to that proposed in the Globe system [36]. Distributed objects use JDBC to operate on the local database and the messaging layer to communicate with instances on other servers.

The messaging layer uses persistent message queues [14, 22, 30] for reliable message delivery and an event-based model for message handling at the receivers. To ensure exactly once reliable delivery even in the presence of partitions and machine crashes, the local messaging layer instance stores messages on the local disk before attempting to send them. Upon the arrival of each message at its destination, the messaging layer instance of the destination invokes the message handler to pass this message to the corresponding distributed object instance. The messaging layer provides transactional send/receive for multiple messages.

We choose IBM DB2 for the database in our distributed TPC-W system. On each edge server, we use the Apache Web Server as the HTTP front-end and Tomcat servlet engine to host business logic servlets. We use a third party implementation of Java Message Service (JMS), called JORAM [23], for the messaging layer. In some of our experiments, we find that the relatively untuned JORAM implementation limits performance. Therefore, as a rough guide to the performance that a more tuned messaging layer might deliver, we also implement a *quick messaging layer* that provides the same interface as JORAM but without the guaranteed correct behavior across long network partitions or node failures. We report performance results for both systems. We modify the TPC-W database schema and business logic for the TPC-W online bookstore from the University of Wisconsin [33] to fit in our object-based edge service architecture. We add 5 distributed objects on both the backend and edge servers to manage the shared information, namely the *catalog*, *order*, *profile*, *inventory*, and *best-seller-list*.

In the rest of the section, we focus our discussion on the design of the five distributed objects. By targeting consistency requirements for each individual distributed object, we explain how to design simple consistency models to solve the *CAP* dilemma in building a replication framework for edge services at the object level.

3.2 Design Principles

Design trade-offs for our distributed TPC-W system are guided by our project goal of providing high availability and good performance for e-commerce edge services as well as by technology trends. When making trade-offs, we consider the fact that technology trends reduce the cost of computer resources while making human time relatively expensive [12]. Therefore, we are willing to trade hardware resources, such as network bandwidth and disk space, for better system availability and shorter latency for users as well as design simplicity and better consistency for system builders. Our first priorities are availability and latency. They are the most important goals for edge services because both availability and latency directly impact the service quality experienced by end users. The second set of priorities are the consistency and simplicity of the system. Good consistency is a high priority because a key challenge in any relaxed consistency system is reasoning about subtle corner cases [17], and good consistency makes this reasoning more straightforward for system designers. We emphasize simplicity throughout our design so we can easily understand and tune system performance and thereby be more confident of its correctness. Simplicity is also important for making the approach useful in practice. To quantify the simplicity of our distributed bookstore system, we compare the size of source code for both centralized and distributed implementations. The source code of the latest distributed TPC-W bookstore implementation is only three thousand lines more than that of the centralized version, excluding the messaging layer implementation. The third priority is optimizing resource usage such as network bandwidth, processing power, and storage. We seek a simple distributed object architecture that improves availability and response time while keeping throughput and system cost competitive with existing systems.

We have made several design decisions based on these priorities. We focus our attention on moderate scale replication with 2-20 edge server locations rather than large scale replication to hundreds or thousands of edge servers. Recent work has suggested that moderate scale replication provides better availability when consistency constraints are considered [43]. This discover simplifies the design of distributed objects because the replication is not at a large scale. We have bypassed a number of potentially attempting optimization options for each distributed object. Since our main objective is to show the feasibility and the effectiveness of using the distributed object architecture for WAN replication, we place a heavy emphasis on simplicity. Future work may further enhance the benefits of the architecture by systematically optimizing performance.

Our distributed object architecture assumes that edge servers are trusted. This requirement of trust is another argument for focusing on replication to a few (2-20) edge servers and not thousands of replicas. This trust model is not only reasonable in the environment where the service provider owns and manages geographically distributed service replicas, but also appropriate when a service provider out-sources replication to a trusted edge service infrastructure provider or CDN that ensures physical and logical security of edge-server resources. We also assume edge servers and the backend server communicate through secured channels though our current prototype does not encrypt network traffic.

3.3 Distributed objects

Distributed objects may be a simple way to achieve both high availability and good consistency for some large-scale systems in the wide area network. We speculate that we can design the distributed objects using application specific knowledge so that their interfaces and internal structures are restricted, making it easier to implement them and to enforce the consistency required of them. In this section we discuss the design and evaluation of the key distributed objects in the distributed TPC-W system.

3.3.1 The catalog object

The *catalog* object is the abstraction of one-to-many updates. It accepts writes at one place and propagates changes to multiple locations for subsequent reads. In the distributed TPC-W system, we use this object to manage catalog information, which contains book descriptions, book prices, and book photos. Update operations on

catalog data are performed at the backend and propagated to edge servers.

The interface of the *catalog* object includes a write operation that takes a *key-value pair*, and a read operation that takes a *key* and returns the corresponding *value*. The backend server issues updates by invoking the write operation, and edge servers retrieve the updates with the read operation. An update from the backend server must be seen at some future time by all edge servers, who retrieve a set of values corresponding to keys. For correctness, the system must guarantee FIFO consistency [31] (aka PRAM consistency [24]) in which writes by the backend are seen by each edge server in the order they were issued. Enforcing FIFO consistency guarantees that, for example, if the backend server creates an object and then updates a page to refer to that object, the system ensures that an edge server that reads the new page will also see the new object. Note that because only one node issues writes, FIFO consistency is equivalent to sequential consistency [26]. But for this same reason it is much easier to implement than general sequential consistency. Also note that although FIFO consistency provides strong guarantees on the order that updates are observed, it does allow time delays between when an update occurs and when it is seen by an edge server. Also, it does not require different edge servers to operate in lock step. For example, if a web page is updated while an edge server, *se1*, is unable to connect to the backend server, another edge server, *se2*, may still read and make use of this updated page while *se1* continues to use the old version.

In our prototype, the *catalog* object uses a push-all update strategy to distribute updates. Once the update is made at the backend, the *catalog* object immediately hands it to the local messaging layer for forwarding to all edge servers. Some time later, the update arrives at each edge server. The *catalog* instances at edge servers read the update, apply it to the local database, and serve it when requested by clients. Although this model can potentially use a lot of bandwidth by sending all updates, we see little need to optimize the bandwidth consumption for our TPC-W catalog object because the writes to reads ratio is quite small for the catalog information. In particular, TPC-W benchmark defines the catalog update operations as 0.11% of all operations in the workload.

This simple implementation meets our system design priorities. It provides high availability and excellent latency to our system because edge servers can always respond immediately to requests using local data. Furthermore, this implementation provides FIFO/PRAM consistency for shared catalog information using a straightforward approach.

Variations of the *catalog* object may be useful for other applications that require one-to-many *scatter* semantics. For example, a *scatter* object could provide a mechanism for propagating edge service infrastructure information such as program or configuration updates. Similar behavior can also be found in other applications such as IBM's geographically-distributed sporting and event service [9], traditional web caching, edge-server content assembly, dynamic data caching [10] and personalization. Note that different workloads may benefit from additional features or optimizations than we choose for the TPC-W *catalog* object. In the design of our prototype, we considered but ultimately did not include these features that may be of use in other contexts.

1. Atomic multi-object update: Some distributed applications require a mechanism to atomically update multiple objects. For example, it may be desirable to atomically update several component parts that are assembled into a single page [11]. Given the support of transactional updates provided by most persistent messaging layers, it should be straightforward to modify the *catalog* object to support atomic multi-object operations (read/write). Potential costs for this feature include a slightly more complex interface and/or a reduction in concurrency of writes and reads due to locking.

2. Data lease: The data served by some time critical applications, such as stock quotes, are meaningful only within a fixed interval. If the local data becomes excessively stale (for instance due to a network partition), some time-critical applications may prefer to deny service rather than serve bad data. To extend our *catalog* object to support such functionality, we could add a new parameter in the write operation to specify a lease period [40] for each update. Of course such a feature is a reduction in availability because servers may be forced to deny service rather than serving stale data.

3. Bandwidth constrained update: Applications that have high write/read ratio with large data objects might not want to use a push-all strategy for propagating updates because it would take a lot of bandwidth to send all updates to all edge servers. Thus, applications with high write/read ratio might need a more sophisticated algorithm to propagate updates. In another study we examine a self-tuning one-to-many data replication algorithm that maximizes availability given a bandwidth constraint by sending FIFO updates for some objects but sending FIFO invalidations for others [26].

3.3.2 The order object

The abstraction of the *order* object is that of many-to-one updates. It gathers writes at various locations and forwards them to a single place for reading. In our distributed online bookstore application, we use the *order* object to manage the propagation of completed orders. Locally, edge servers accept user orders, which need to be processed at the backend server for fulfillment.

The interface for the *order* object includes an insert operation that takes an *order*, an *order sequence ID*, and an *edge server ID*, and a *message handler* that processes orders when they arrive from edge servers. Each order is identified by the pair, *edge server ID* and *order sequence ID*, which increments by one whenever a new order is created on an edge server. Orders are sent by each edge server in the sequence that they are initially created on that edge server, and the messaging layer delivers messages in the same sequence as they are inserted. Therefore, orders from the same edge server maintain FIFO consistency at the backend server but different servers' orders can be arbitrarily interleaved. The handler interacts with the persistent message layer to ensure that all orders are guaranteed to be processed exactly once by the backend order object instance.

An incoming message is deleted from the local messaging layer only if the handler successfully processes the order. If a crash happens while an order is being processed, the incomplete processing is rolled back during database recovery. Since the message handler did not complete, the messaging layer invokes the handler again during recovery. The handler detects duplicates when it processes an order. In that case, it does a *no-op* and returns to the messaging layer as if the order had been successfully processed.

The *order* object provides high availability and excellent latency to our system by decoupling edge servers' local requests processing from the persistent store-and-forward processing of orders to the backend server.

The mechanism of the *order* object can be extended for other applications. For example, since it supports FIFO consistency for updates from the same machine, we can use it to gather the system logs in distributed systems to, for example, gather user click patterns at a web site.

3.3.3 *The profile object*

The *profile* object handles reads/writes with low concurrency and high locality. Each entry contains information about a single user such as name, password, address, credit card information, and the user's last order. Users can only access or modify fields of their own profile records.

The interface of the *profile* object includes a simple read operation and a write operation. The read operation takes the *user ID*, and returns the corresponding profile record. The write operation takes the *user ID*, the *field ID*, and a *value*. The read operation provides access to all fields of the profile record, and the write operation updates a specified field of the record. The profile information has a low write/read ratio of less than 12.86% [34]. We assume the server selection logic that directs users to specific edge servers will generally send the same user to the same edge server for relatively long periods of time so that the user usually modifies his/her profile record on the same edge server. Therefore, the chances for concurrent access of the same profile record at two edge servers is generally low. However, sometimes users will be switched from one edge server to another (e.g. in response to geographic movement of the user, load balancing, or network or server failures). A simple and correct requirement for the *profile* object is that any profile object must be accessible from any edge server.

Given the low concurrency and high locality of access to profile records and relatively low volume of writes, our prototype implementation (1) uses a write-any read-any policy that does not require locking across servers, (2) propagates updates among all edge servers with best effort to propagate all changes quickly, and (3) applies object-specific "reconciliation rules" [27, 32] to resolve conflicting updates to the same field of the same record on multiple edge servers. Whenever a profile record is modified, the update is enqueued in the message layer to be sent to the other edge servers. If a set of edge servers are disconnected at the time of the update, the local messaging layer ensures delivery of the update after those servers recover. If two concurrent write operations update the same field of a record on different edge servers, the object code resolves the conflict with per-field reconciliation rules based on the information type. For example, the reconciliation rule for the last order field is to compare the time that the order is placed and the most recent order wins; the rule for credit card records or shipping addresses is to merge multiple updates and prompt the user for selection when the client makes an online purchase.

The design of the *profile* object ensures availability and minimizes latency by relaxing consistency compared to sequential consistency [7]. Updates can take place on any edge server without having to lock the targeted record. Access locality and rapid best-effort propagation of all updates to all locations reduce the number of conflicts [19], and rare update conflicts are satisfactorily resolved by simple per-field reconciliation rules.

Our decision to replicate all profile records on all edge servers maximizes availability, optimizes response time, and emphasizes simplicity at the cost of increasing storage space and update bandwidth in keeping with our design priorities. Since the profile objects are small and updates to them are infrequent, partial replication would likely reduce overhead modestly at best and may hurt performance, availability, or simplicity. However, systems with large number of replicas could see large benefits to considering more sophisticated partial replication.

A wide design space exists for providing consistency on read/write objects in distributed systems [31], and the trade-offs selected for the *profile* object may not be appropriate for other read/write records. In an environment where access patterns and object semantics are less benign than the *profile* object, general approaches might proceed in two dimensions.

1. Strengthening consistency from the underlying FIFO/PRAM propagation of updates to provide stronger semantics such as casual consistency (which may require more complex communication mechanisms such as Bayou's log exchange protocol [29]) or sequential consistency (which may require locking). Quorum based solutions such as [13] could also be explored.

2. The "reconciliation rules" currently hand-coded in the *profile* object logic might be made more general by, for instance, providing an interface on a read/write object to specify reconciliation rules as a parameter [32].

3.3.4 *The inventory object*

To examine consistency constraints beyond that of the standard TPC-W benchmark, our distributed-bookstore benchmark adds the constraint of a finite inventory for each item. It requires that if the inventory of an object is 0, users requesting this object must be notified that delivery may take longer than normal (e.g. the item is not in stock and is on back-order). We enforce this constraint with an *inventory* object. We observe that the actual count of the inventory is not important for processing order requests as long as stock is sufficient. The inventory responds either "OK" to process the order or "warning" for back-orders. It is acceptable to be conservative and issue warnings when the inventory is unsure whether items remain. (The downside is that users may cancel orders when they receive warnings in the ordering process. But we can minimize these false positives with careful system design and implementation.)

The inventory information can be interpreted as *ID* and *quantity* pairs. Every pair maps a particular book in the store to the number of copies of the book. The interface of the *inventory* object is the *reserve* operation, which takes a *numeric value* and a *book ID*, and returns a boolean value. If the returned value is *true*, it implies that the *reserve* operation successfully decrements the number of copies of the specified book by the given amount. If the inventory is insufficient to accommodate the request, *false* is returned. Note that the use of a transactional database and persistent messaging layer allows us to restore this escrowed inventory if the transaction fails to complete due to a failure or user cancellation. A key observation is that edge servers care little about the actual inventory of each book, as long as the inventory is sufficient for them to continuously process order requests.

In our simple prototype system, the total available inventory is divided among edge servers by giving each object instance a *localCount* and enforcing the invariant that the sum of all local counts across all instances equals the global inventory count. Initially, inventory is evenly distributed among all edge servers. Edge servers process requests with their local inventory without communicating with the backend or other edge servers, and their local inventory decreases over time. We implement a simple protocol between the backend server and edge servers for inventory re-distribution. By observing the orders received at the backend server (see section 3.3.2), the *inventory* object instance at the backend server keeps track of the edge server with the most inventory and the edge server with the least inventory. Whenever the inventory difference between the two edge servers exceeds a certain threshold, the inventory instance at the backend server requests inventory re-distribution between these two edge servers. Note that such a redistribution request may fail because the backend might have stale information about donor's inventory. Such a failure is benign because the backend server eventually becomes aware of the donor's true inventory and selects a different donor. Also note that our use of

Object	Object State Replication	Updates Propagation
Catalog	all records at all servers	backend ⇒ all edges
Order	1/N at edge; N/N at backend	edges ⇒ backend
Profile	all records at all servers	all edges ⇒ all edges and backend
Inventory	local view at edge; all local views at backend	on threshold: an edge ⇒ backend ⇒ an edge
Best-seller-list	approximate view at edge; current view at backend	on threshold: backend ⇒ all edge

Table 1: Distributed object state replication and propagation.

persistent messaging layer generally simplifies the design of the redistribution by ensuring that inventory is never lost or duplicated in transfer.

The inventory implementation meets our design goals by increasing the overall availability of the system while providing acceptable consistency guarantees on the data served to clients. It also reduces the communication between edge servers and the backend because edge servers do not need to check availability of the central inventory upon every order request. Therefore, we improve the system response time and make the system more tolerant to network partitions. The limitation of our design is occasional "false positives" when local count is 0 and inventory instance reports *false* while counts on other edge servers is not 0. We examine this issue in Section 4.4.

The inventory object's performance could potentially be further optimized by including any of the following enhancements that we considered but did not adopt in our implementation because we have not felt the need for the extra complexity. However, these optimizations may be of use in optimizing this object for other environments.

1. Fetch on-demand: When the system realizes the local inventory is insufficient to accommodate an incoming request, it could delay processing the request and send messages to other edge servers to request more inventory. If it receives a positive response, the request could then be processed. If no positive response is returned within a time period, the request would be reported as back-ordered as it is now.

2. Sophisticated redistribution: For example, when a particular edge server experiences heavy demand for an item, the system might allocate a larger percentage of inventory to that edge server.

3. Peer-to-peer inventory exchange: The mechanism of the *inventory* object is similar to the *numerical error* guarantee mechanism in TACT [42]. Unlike TACT, our system adjusts the local inventory with a centralized coordinator for simplicity. We could change the *inventory* object to employ the peer-to-peer to model in which edge servers "exchange" inventory directly.

3.3.5 The best-seller-list object

The *best-seller-list* object maintains lists of best seller books for each subject. The best seller books are computed for each subject based on the sales volume in the 3,333 most recent orders. The lists contains 50 books in each subject with the highest sales volume.

The interface of this object includes a read operation that takes a *string* as the subject and returns a list of best seller books under the subject. The best seller books change over time as different books are sold. For the best seller lists to be accurate on every edge server, all sales activities on all edge servers must be taken into account when computing the lists. However, the lists may not change on every sale. For example, several additional purchases of books that are already in the best seller lists often do not change the lists. In this sense, the system only cares about the sales activities exceeding some threshold. Furthermore, it is preferable to return slightly stale best seller lists rather than to stop serving requests. Some delay in propagating order information is also acceptable.

In our prototype system, we maintain a copy of the best seller lists on every edge server. The approach that we take to maintaining the best seller lists is similar to that for maintaining the inventory among edge servers. By observing the orders received at the backend server (see section 3.3.2), the *best-seller-list* object instance at the backend server keeps track of the sales volumes of all books. As soon as the lists change, the instance at the backend server sends messages through the messaging layer to *best-seller-list* instances on all edge servers to update the lists.

This simple implementation meets our design goal. It improves system response time and increases system availability by minimizing the communication among edge servers and to the backend server for computing and updating the lists and detecting the changes in the lists. It reduces bandwidth consumption and dependencies among edge servers by monitoring all orders at the backend server instead of exchanging order information among edge servers.

3.4 Issues

In the above subsections, we discussed the design of the distributed objects used in building our distributed TPC-W system. Distributed objects are designed based on the specific application semantics such that they provide simple interfaces and implementations. In addition, objects encapsulate consistency guarantees and those guarantees are straightforward and easy to reason about for both developers and users of the objects. We believe the overall system consistency is preserved when objects are integrated together in the system. However, reasoning about consistency of distributed systems is difficult, and subtle interactions between distributed objects, each of which maintains its own consistency using specific strategies, could conceivably result in unexpected behaviors. Precisely characterizing the interactions among different consistency models across objects is an important task for future work.

The distributed objects maintain the consistency of each edge server such that each edge server has a consistent view of the shared state. However, occasionally the edge server selection algorithm may switch clients from one edge server to another to balance load or in response to node failures, network partitions, or client mobility. Clients could then observe inconsistency. For example, edge server *se1* may have a newer version of the catalog information than edge server *se2*. When a client is switched from *se1* to *se2*, this client may see older catalog information on *se2*. One solution to resolve this issue is to use client browser cookies to enforce a Bayou-like client consistency model [32] to ensure that clients always communicate with sufficiently updated servers. We will consider this feature in our future work.

Table 1 contains the summary of state replication and update propagation of each distributed object.

4. SYSTEM EVALUATION

The experiments target the availability, performance, and consistency of the distributed bookstore system in normal operation and while the system is partitioned due to network failures.

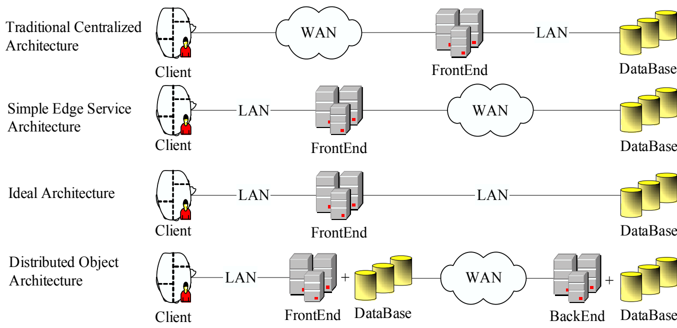


Figure 2: The network configuration of WAN service architectures.

4.1 Environment

To demonstrate our distributed bookstore system, we deploy a prototype across 4 servers, three of which act as edge servers and one as the backend server. Each server runs on a Pentium 900MHz machine with 256MB memory. IBM DB2 databases are installed on all server machines. On the three edge servers, we use Apache and Tomcat to host the servlets that implement the server logic. Machines in our lab are connected via 100Mbit Ethernet connections. But in order to simulate a WAN environment among servers during experiments, we direct all the traffic (both in and out) of server machines to an intermediate router, which simulates WAN delays and temporary network outages with Nistnet. In the remaining discussion, we refer to links via Nistnet with bandwidth of 10Mbit/s and latency of 50ms as WAN links and we refer to direct 100Mbit/s links between machines as LAN links. We use three client machines to generate workload. These three machines have Pentium 900MHz processors, and each of them connects to a separate edge server via a LAN link. One instance of the TPC-W client program is running on each client machine generating a predefined workload against each edge server. TPC-W defines three workloads mixes, each with a different concentration of writes. In our experiments, we focus on the *ordering mix*, which generates the highest percentage of writes (50% of browsing and 50% of shopping interactions in this mix).

4.2 Performance

In this section, we evaluate the performance of our distributed TPC-W system with respect to two criteria: latency and throughput. As noted in Section 3, our most important performance goals is to minimize the system latency because latency alters the "human waiting cost." At the same time, we want to see if our system throughput is competitive with a traditional centralized architecture.

To evaluate the system performance, we run the TPC-W on four architectures as illustrated in Figure 2. We use one front-end machine and one back-end machine in this experiment to evaluate the performance of each architecture. The *traditional centralized* architecture has both its front end and central database connected by LAN links, but end users must access the front end via WAN links. The *simple edge service* architecture replicates its front ends at the edges of the network near end users. The front ends connect to end users via LAN links and connect to the central database via WAN links. The *ideal* architecture has end users, front ends, and the central database all within a LAN environment. This architecture is unrealistically optimistic, but it serves as a point of reference. The *distributed object* architecture, presented in this paper, replicates both its front ends and databases at the edges of the network near end users. The front ends (edge servers) connect to end users via LAN links and connect to the core server and other front ends (edge servers) through the distributed objects via WAN links. In addition to the one front-end system, we examine the performance of the distributed architecture with 3 edge servers (front ends). For the communication layer, we use both JORAM that uses persistent message queues to send messages and the quick messaging layer that asynchronously sends messages without storing them on the local disk. Note that the latter configuration is intended to illustrate the range of performance that different messaging layers might provide, but because it does not provide reliable messaging across failures, it would not be appropriate for actual deployment.

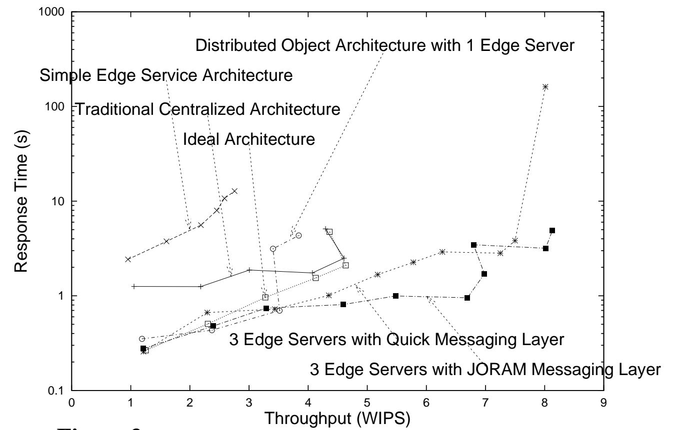


Figure 3: System response time as the workload increases.

By comparing the performance results of the distributed bookstore application across four architectures, we seek to demonstrate three points. First, at low workloads, the latency when using the distributed object architecture matches that when using the ideal system and is significantly better than that of the traditional system or the simple edge server system. Second, the throughput when using the distributed object architecture is competitive with the ideal or the traditional architecture. Third, when the edge server becomes the bottleneck under heavy workloads, we can increase system throughput by adding more edge servers. While varying the request rate, we measure both system throughput and response time. In all systems we expect to have the best response time when the request rate is low. Then, as the request rate increases, the response time will increase as well, until when the maximum system throughput is reached and the system becomes saturated, at which point the response time will increase sharply.

Figure 3 shows the system performance as we vary the workload. In the graph, the x-axis represents the throughput in WIPS (web interactions per second), and the y-axis represents the response time of the TPC-W application deployed on four architectures. We explain the curves from the top to the bottom. The top most curve represents the response time for the simple edge service architecture. This system experiences the worst minimum response time of 2.42(sec/req) because many client requests to the edge server trigger multiple requests from the edge server to the core server across the WAN. The WAN delays, which are set to 100ms RTT, dominate the system response time. In contrast, under the traditional centralized architecture, every client request goes across WAN links just once. The overall response time for the traditional centralized system is indicated by the second curve from the top, and it shows nearly a factor of two improvement to 1.25(sec/req). Note that once the system is saturated, increasing the offered load actually reduces throughput slightly. The curve indicating the response time of the ideal architecture improves response time by nearly another factor of five, to 0.26(sec/req). The response time of the distributed object architecture with one edge server is approximately the same as that of the ideal architecture, as indicated in the graph.

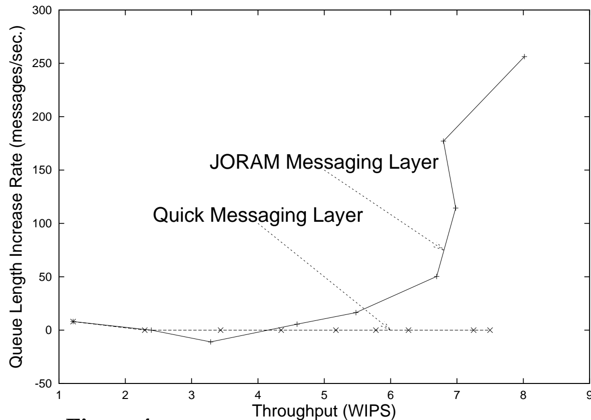


Figure 4: The growth rate of message queue length.

Our distributed object architecture with one edge server reaches its maximum throughput of 3.7(WIPS), which is approximately 25% less than that of the traditional centralized architecture's maximum throughput of 4.6(WIPS) (or the ideal architecture's maximum throughput of 4.6(WIPS)). This difference is due both to the extra overhead of persistent messaging and the fact that the traditional architecture uses different machines for the front end and database. We believe that production deployment of our edge server system would likely make use of separate front-end machine(s) and database machine(s) at each edge server site to increase throughput.

After we add two more edge servers, the throughput of the distributed object architecture using either messaging layer continues increasing to 8.1(WIPS), which is roughly 119% performance improvement. Our edge service architecture appears to send all updates to the backend server, which does not seem to have potential for any speedup. However, two facts allow adding more machines to yield this speedup of our system. First, the read operations, which constitute more than 50% of the workload, are distributed among edge servers. Second, technology exists to make backend database quite scalable. Our architecture should be similarly scalable even with a single backend server that sees all updates. We believe that the distributed architecture approach should be viewed as a way to increase availability and improve latency while scalability of throughput is improved with cluster technology.

The throughput of our distributed TPC-W bookstore system is competitive with that of other academic systems [3, 18, 33]. If we assume that a typical Pentium III machine costs roughly $800, the price/performance cost of our system is roughly 250 ($/WIPS), which falls in the range of published standard industry TPC-W performance results, 24.50-277.80($/WIPS) [39]. We are primarily limited by our machine memory capacity - frequent paging activity seems to limit our system maximum throughput. Future work is needed to explore the scalability of our system as it is tuned to match the throughput of highly tuned commercial systems.

Figure 4 suggests another advantage of the distributed object architecture. The system provides stability against workload bursts in terms of the response time by buffering the updates on its local disk. In this graph, the x-axis represents the system throughput and y-axis represents the growth rate of the buffered messages. When the throughput exceeds 6 web interactions per second (WIPS), the message insertion rate exceeds the constant message forwarding rate of JORAM messaging layer and the queuing time of all messages starts to increase. As shown in the graph, the growth rate of buffered messages on the messaging layer increases sharply after the throughput reaches 6 WIPS. Those buffered messages will be sent as the system load reduces to normal so that no request is rejected during the bursty period. Note, however, that the maximum steady-state throughput of JORAM is roughly 4-5 WIPS and the burst throughput of JORAM is roughly 8 WIPS. The quick messaging layer does not buffer messages so it can not defer message processing during bursts of demand load.

4.3 Availability

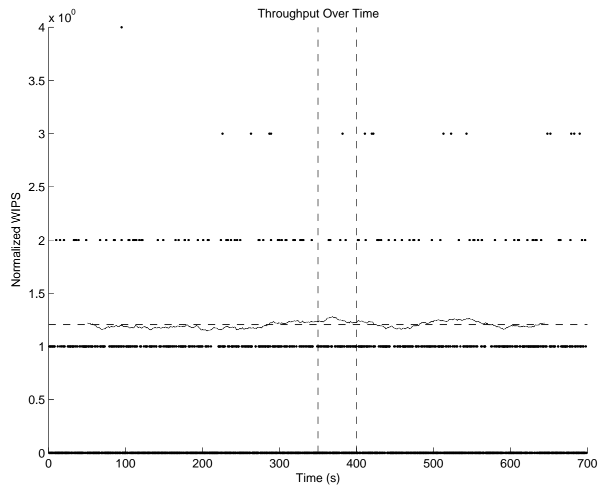
A key aspect of our design is that each edge server processes all requests with only local information. As long as a client can access any edge server, it can access the service even if some of the servers are down or if network failures prevent communication among some or all of the servers. In this section, we examine the performance impact of message buffering and processing during and after failures with both *JORAM Messaging Layer* and *Quick Messaging Layer*. We use two different workload rates on each of the messaging layers.

Figure 5 shows the system throughput, average response time, and message queue lengths when the system uses either *JORAM Messaging Layer* or *Quick Messaging Layer* before, during, and after a network failure. Each run lasts for 700 seconds, and a network outage occurs roughly 350 seconds after the experiment starts and lasts for 50 seconds. During the network outage, no server can communicate with any other server, but the normal communication among servers resumes once the network is restored. In order to provide a moderate load that does not cause queues to develop before the network fails, we apply a workload of 1.2 WIPS to the system that runs on the top of JORAM, and apply the workload of 6.8 WIPS to the system on the top of the quick messaging layer.

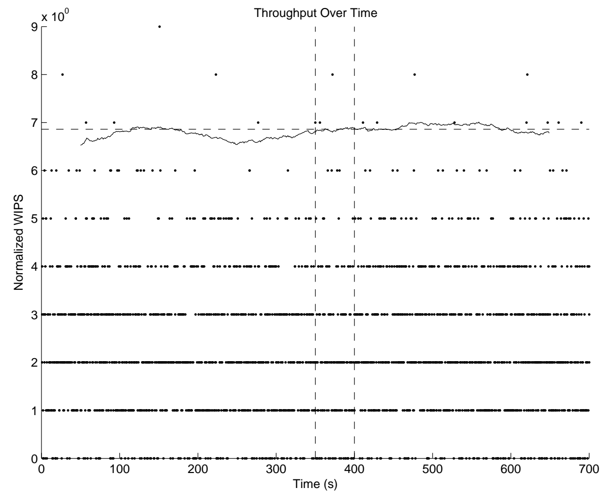
Graph (a) and (b) in Figure 5 indicate the system throughput throughout the 700-second session. The x-axis represents the time progression and the y-axis represents the system throughput. In each graph, the straight horizontal dash line represents the average throughput of the 700-second session and the solid slightly wiggly line represents the running average of throughput over 300-second intervals. The wiggly line stays close to the straight dash line in both graphs. It implies that the throughput of systems with both messaging layers is consistent throughout the session, and the network failures during the session have little effect on the system. Our distributed TPC-W system can operate normally while being partitioned because the databases are replicated locally through distributed objects, and they can continuously provide data for server computation while partitioned by network outage.

Figure 5 (c) and (d) show the average response time for the two systems, one on the top of JORAM, another on the top of the quick messaging layer. The x-axis in the graph represents the time progression in seconds and the y-axis represents the system response time. The system response time appears unaffected by the network outage during the session because the graph does not show an increase in response time during the failure interval, between 350 and 400. Because the response times for different interactions vary, two curves in this graph tend to fluctuate throughout the session.

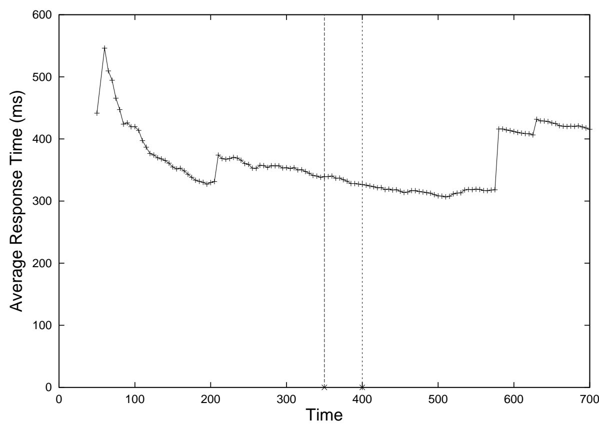
Figure 5 (e) and (f) show the average queue lengths in the two messaging layers. The x-axis represents the time progression and the y-axis represents the queue length in the number of messages queued. There are few messages queued by the messaging layers before the failure starts, but the number of queued messages starts growing after 350 seconds. The curve that represents the queue length of the quick message layer indicates a sharper increase in message length than that of JORAM because the workload used on the quick message layer is about 4 times bigger than the workload on JORAM. But all messages are quickly cleared out of the queues after the network partition is fixed. Note that JORAM Messaging Layer clears out queued messages relatively slower because it has a



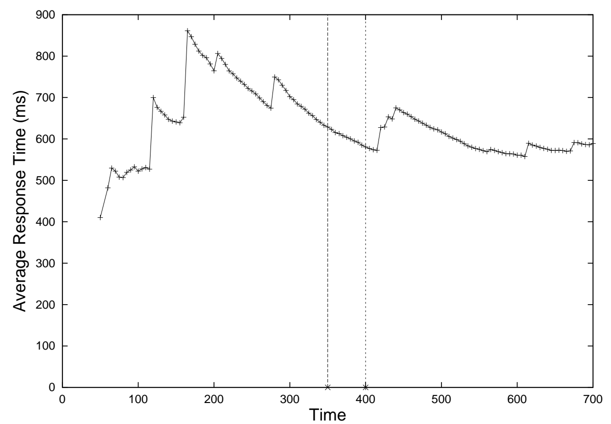
(a) System throughput (JORAM Messaging Layer)



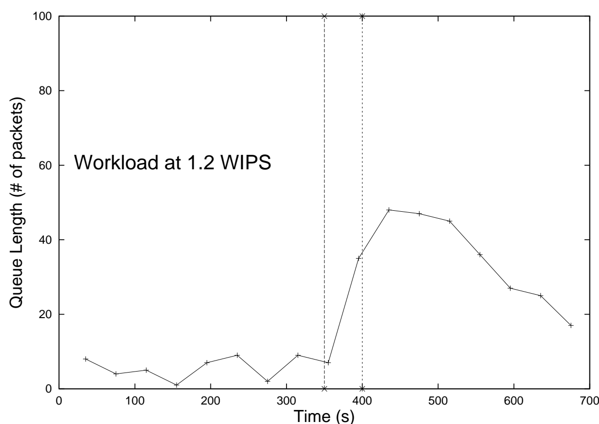
(b) System throughput (Quick Messaging Layer)



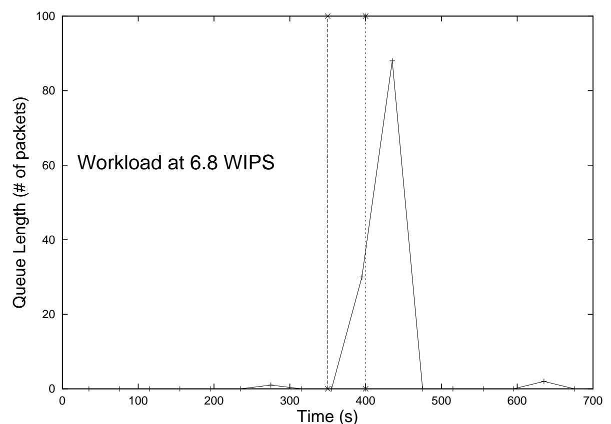
(c) System response time (JORAM Messaging Layer)



(d) System response time (Quick Messaging Layer)



(e) Average queue length (JORAM Messaging Layer)



(f) Average queue length (Quick Messaging Layer)

Figure 5: 700-second session with network outage lasting for 50 seconds.

fixed message forwarding rate, approximately 4 msg/sec, which is much less than that of the Quick Messaging Layer. This behavior is due to the persistent queuing overhead and the vender specific design of JORAM Messaging Layer.

During the network failure, the information on each edge may become stale. However, instead of completely stopping sales during these failures, the service provider prefers to continue serving users with stale information, such as stale catalog and stale best seller lists, accepting orders with stale inventory which may increase back-order rate, and delaying orders to be processed at the backend server by buffering them on local disks. The trade-offs seem appropriate and acceptable for this application.

4.4 Consistency

Because the system slightly relaxes consistency for higher availability and performance, during the normal system operation or network failures users may view stale information. In this section, we evaluate the consistency of our distributed TPC-W system during normal operation by examining the staleness of local inventory.

By distributing the bookstore inventory among all edge servers, the system allows edge servers to accept orders locally. However, when a heavy workload is unbalanced across servers and the inventory is low, some books may be sold out on a particular edge server during a short time frame before the inventory re-distribution arrives from other edge servers. In this case, some order requests targeting the sold-out books may pessimistically report that the shipment may be delayed. In this experiment, we examine the back-order rate under a condition where the inventory is low and workload is unbalanced. We expect the back-order rate to approximate the ideal back-order rate seen by a centralized system as long as the inventory re-distribution time is less than the inter-arrival time between requests targeting the same book.

In order to create a purchasing imbalance across edge servers, we direct all order requests to only one of the three edge servers. To maintain a low inventory count at each edge server, we choose three sets of inventory for each run of the experiment: *2 copies per title with 5 titles*, *4 copies per title with 5 titles*, and *6 copies per title with 5 titles*. The workload is designed such that each order request randomly targets one of 5 books, and we run the experiment long enough so that the average total number of books ordered is 50% of the inventory on the edge server, which is roughly 16.7% of overall inventory in the system. By varying the average inter-arrival time of requests targeting the same book, we can measure the average back-order rate for different sets of inventory. If we run against the traditional centralized architecture with given sets of inventory and workload, there will be no back-order because even under the most extreme case where all requests target the same book in the centralized system, the total number of requested copies is less than the number of copies of any particular book. The ideal back-order rate is zero for the defined sets of inventory and workload.

Furthermore, we speculate that if the distributed-object architecture has the inventory re-distribution time (RDT) much less than the requests inter-arrival time (RIT) per title, the distributed-object architecture can approximate the ideal back-order rate, i.e.:

$$RDT \ll RIT/titles$$

Figure 6 shows the percentage of the back-orders due to the inventory shortage as we vary the request inter-arrival time per book title. In the graph, the x-axis represents the average inter-arrival time of requests targeting the same book and the y-axis represents the percentage of rejected requests over all requests. All three curves approach the x-axis (the ideal back-order rate) as they extend to the right where the request inter-arrival time is large. The workload that has the average request inter-arrival time of less than 2 seconds has the back-order rate greater than 1%. It indicates that our system inventory re-distribution process takes roughly 2 seconds or less to complete, which is expected because edge servers use asynchronous message exchange across the WAN for computing and re-distributing inventory.

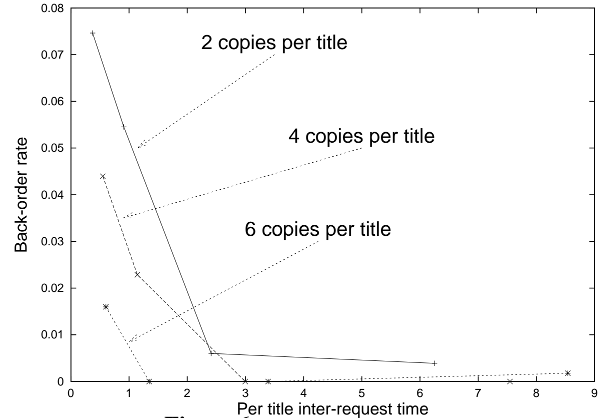


Figure 6: The back-order rate.

It is worth noting that the small back-order rate shown in Figure 6 only represents the system consistency in the extreme cases where inventory is small and low, 2-6 copies per book with 5 different books, and the workload is unbalanced. Also as noted in section 3.3.4 several optimizations can be applied to further reduce inconsistency.

5. RELATED WORK

Our general approach is similar to that of the Globe system [36] which proposes a uniform framework for distributed computing in wide area networks based on the concept of distributed shared objects. A distributed Globe object is built from a set of local objects in different address spaces, and each local object interacts with local objects in other address spaces. Separately, sub-objects within a local object handle the flow-control within the object, communication between the object and objects in other address spaces, replication strategy of the shared state, and semantics of the particular object. The goal of this design is to let programmers exploit application semantics in the design and implementation of individual objects and allow programmers to make use of pre-constructed replication modules to easily invoke standard consistency algorithms with different objects. This distributed objects model seems to provide a flexible and powerful way to build distributed applications in the wide area. But to our knowledge, there is little work quantitatively evaluating the benefits of this approach in building data-oriented services, such as e-commerce applications. In our project, we apply a distributed object approach to build the TPC-W benchmark online bookstore and quantitatively show that we can achieve higher system availability and better performance by leveraging the distributed objects architecture with specific application semantics. Our specific implementation differs from Globe in that we do not follow the same uniform internal structure of Globe objects that separate "semantics object" from "replication object". The advantage of separating these modules is that a set of standard consistency implementations may be reused in different objects. However, we found it simpler to integrate semantics and replication consistency code. Future work is needed to see if our consistency algorithm can be modulated in a way that allows simple reuse in different objects. Also our implementation uses transactional persistent messaging for all communication across objects. Our experience is that this choice generally simplifies the design of the

object by eliminating the need to ensure reliable message delivery at the object level.

Gribble uses distributed objects as building blocks for providing cluster services on the Internet [20]. He employs the general distributed object approach to hide behind objects the implementation complexity of availability, performance, scalability, and consistency and provide a simple interface to programmers. The work focuses on a restricted environment (clusters) where partitions are rare.

Garcia et al. [18] study the TPC-W benchmark, including its architecture, operational procedures for carrying out tests, and the performance metrics it generates. Their experimental results demonstrate that TPC-W is a useful tool for generating a standard metric of the transactional capacity of servers working in e-commerce environments. The PHARM project [33] at the University of Wisconsin focuses on the micro-architectural characterization of the TPC-W defined workload such as branch predictability, caching behaviors, and multiprocessor data sharing patterns. Amza et al. [3] characterize the bottleneck of dynamic web site benchmarks, including the TPC-W online bookstore and auction site. Their study focuses on discovering and explaining the bottleneck resources in each benchmark.

Many studies have addressed the importance of caching dynamic content to improve system performance and scalability. Challenger et al. [9] develop an approach for consistently caching dynamic Web data that became a critical component of the 1998 Olympic Winter Games Web site. But it concerns only the single writer case. Arlitt et al. [4] studied the scalability of a large online shopping system by performing workload characterization, and they conclude that linear scalability is not always adequate in case of workload bursts. They suggest efficient caching and capacity planning techniques to increase the system scalability and performance.

Most commercial databases support data replication with an *eager* or *lazy* consistency model [19]. The eager update model considers updating every replica as part of a single transaction, which may decrease the system availability and response time when used in wide area replication. The lazy update model is usually preferred for WAN replication because updates are asynchronously propagated to other replicas. Although general database systems support procedures for resolving conflicts, those procedures are normally defined with database level semantics [27].

Our *order*, *inventory*, and *best-seller-list* objects take advantage of the fact that updates are commutative and can be slightly reordered. The value of commutativity for simplifying consistency has also been used in write-anywhere databases [19].

Bayou [29] and TACT [42] have explored the space of relaxed consistency models. The Bayou replication framework uses mechanisms like the anti-entropy protocol to guarantee the eventual consistency of the system, and it uses version vectors to ensure client consistency. TACT constructs a model for evaluating the trade-offs between availability and consistency. The system can be tuned to provide availability that is subject to the specified consistency requirements. Vahdat et al. build the TPC-W benchmark on top of TACT to demonstrate the feasibility of using TACT as a database middleware for traditional, SQL-based database applications [37]. They evaluate both the performance benefit and consistency costs of continuous consistency for their TPC-W implementation across a variety of replication scenarios and consistency bounds. Both Bayou and TACT provide hooks for application developers to attach specific reconciliation rules to resolve update conflicts [32]. The design of some of our distributed objects make use of these ideas.

6. CONCLUSIONS

Our TPC-W bookstore is built using a distributed object architecture to provide high availability and excellent performance. The throughput and response time of our system are consistent before, during, and after network partition. By measuring latencies of four architectures, we show that the response time of our system closely approximates that of the ideal system, and our system performance is dramatically improved comparing to the traditional architecture.

Replicating shared data everywhere seems to limit the system scalability. But the speedup of our system is possible by adding more hardware resources (machines). Although we propagate updates to all edge servers, the percentage of replicated updates is less than 50% of overall workload in the system (reads constitute more than 50% of the TPC-W workload). We do not view the distributed architecture approach merely as a way to improve the system scalability in terms of throughput, but as a way to increase availability and improve latency.

Building the replication framework with a distributed object approach is relatively straightforward. We design the consistency model for each individual distributed object by using the corresponding application specific semantics. It then becomes easy to reason about the trade-offs between availability and consistency for each object. Usually, we can slightly relax the consistency of a distributed object to achieve high availability and efficiency. In addition, distributed objects encapsulate the complexity of data replication and provide simple interfaces for applications to access shared data. Thus, an attractive deployment strategy may be for experts in WAN performance and consistency to construct useful distributed objects that non-expert programmers can use for building web services.

Using persistent message queues is crucial in building our system, and doing so also simplifies the design of the distributed objects. The persistent messaging mechanism provides asynchronous, reliable, and transactional message delivery, which are essential for e-commerce applications. In addition, it provides simple interfaces for communication among all distributed objects.

Our distributed objects can be optimized and tuned for use in other environments. In order to avoid complexity in our evaluation, we keep the design of distributed objects simple while meeting the performance and consistency demands of our TPC-W system. However, there are opportunities for optimizations and tuning for those distributed objects as we have discussed in Section 3. We plan to further explore the design space and the applicable environment of those distributed objects in our future work.

As pointed out, we need further study on consistency issues across distributed objects and across edge servers. In the future, we will investigate the interactions among consistency models as the models become more sophisticated in other environment. We will also study the impact on system consistency as users move from one edge server to another.

7. REFERENCES

[1] Akamai, Inc. Home Page. www.akamai.com.

[2] Inc. Akamai. Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite. White paper, Akamai, 2002.

[3] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck Characterization of Dynamic Web Site Benchmarks. Technical Report TR02-391, Rice University, Feb 2002.

[4] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the Scalability of a Large Web-based Shopping System. *ACM Transactions on Internet Technoogy*, June 2001.

[5] A. Awadallah and M. Rosenblum. The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content

Distribution. In *7th International Workshop on Web Content Caching and Distribution*, August 2002.

[6] S. Bhattacharjee, K. Calvert, and E. Zegura. Self-organizing wide area network caches. Technical Report GIT-CC-97/31, Georgia Tech, 1997.

[7] E. Brewer. Lessons from giant-scale services. In *IEEE Internet Computing*, July/August 2001.

[8] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 98*, 1998.

[9] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *In Proceedings of ACM/IEEE, Supercomputing '98 (SC98)*, November 1998.

[10] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE Infocom*, March 1999.

[11] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE Infocom*, March 2000.

[12] B. Chandra. Web workloads influencing disconnected services access. Master's thesis, University of Texas at Austin, 2001.

[13] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data. *IEEE Transactions on Knowlegde and Data Engineering*, 1(3):387–397, September 1989.

[14] IBM Corporation. MQSeries: An Introduction to Messaging and Queueing. Technical Report GC33-0805-01, IBM Corporation, July 1995. ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf.

[15] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN Service Availability. *IEEE/ACM Transactions on Networking*, 2003. To appear.

[16] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proceedings of IEEE Infocom*, March 1998.

[17] M. Frigo. The Weakest Reasonable Memory Model. Master's thesis, MIT, 1988.

[18] D. Garcia and J. Garcia. TPC-W E-Commerce Benchmark Evaluation. *IEEE Computer*, pages 42–48, February 2003.

[19] J. Gray, P.Helland, P. E. O'Neil, and D. Shasha. Dangers of Replication and a Solution. In *Proceedings of SIGMOD*, pages 173–182, 1996.

[20] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[21] IBM. *The Economic Value of Rapid Response Time*, pages 11–82. Number GE20-0752-0. White Plains, N.Y., 1982.

[22] Java Message Service (JMS). http://java.sun.com/products/jms.

[23] JORAM. http://www.objectweb.org/joram.

[24] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[25] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *SOSP95*, December 1995.

[26] A. Nayate, M. Dahlin, and A. Iyengar. Data Invalidation and Prefetching for Transparent Edge-Service Replication. Technical report, University of Texas at Austin Department of Computer Sciences, November 2002.

[27] Oracle7 Server Distributed Systems: Replicated Data. http://www.oracle.com/products/oracle7/server/whitepapers/replication/html/index, 1994.

[28] V. Paxson. End-to-end Routing Behavior in the Internet. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1996.

[29] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[30] Charles Sterling. Programming Best Practices with Microsoft Message Queuing Services (MSMQ). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmqqc/html/msmqbest.asp.

[31] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*, chapter Consistency and Replication. Prentice Hall, 2002.

[32] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 172–183, December 1995.

[33] The PHARM Project at the University of Wisconsin. http://www.ece.wisc.edu/ pharm/tpcw/.

[34] Transaction Processing Performance Council. Home Page. http://www.tpc.org.

[35] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *The Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[36] M. van Steen, P. Homburg, and S. Tanenbaum. Globe: A Wide-Area Distributed System. Technical report, Vrije Universiteit, March 1999.

[37] K. Walsh, A. Vahdat, and J. Yang. Enabling Wide-Area Replication of Database Services with Continuous Consistency. Unpublished Manuscript.

[38] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *OSDI02*, December 2002.

[39] TPC-W performance result in price/performance . http://www.tpc.org/tpcw/results/tpcw_price_perf_results.asp.

[40] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large scale dynamic web services. In *Proceedings of the 2001 International World Wide Web Conference*, May 2001.

[41] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, January 1997.

[42] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[43] H. Yu and A. Vahdat. Minimal Cost Replication for Availability. In *Proceedings of the Twenty-First Symposium on the Principles of Distributed Computing*, 2002.

[44] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.