Improving Availability and Performance with Application-Specific Data Replication*

Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng

The University of Texas at Austin

Arun Iyengar

IBM T.J. Watson Research Center

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 suffering the availability and performance penalties that would be incurred by accessing a traditional centralized database. This article explores using a distributed object architecture to build an edge service data replication system for an e-commerce application, the TPC-W benchmark, which simulates an online bookstore. We take advantage of application specific semantics to design distributed objects that each manages 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, our application realizes both high availability and excellent performance. For example, in one experiment we find that our object-based edge server system provides five times better 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.

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

Categories and Subject Descriptors: C.2 [COMPUTER-COMMUNICATION NETWORKS]: Distributed Systems. C.4 [PERFORMANCE OF SYSTEMS]: Design studies, Fault tolerance, Reliability, availability, and serviceability.

General Terms: Availability, Design, Measurement, Performance.

Keywords: Availability, Data Replication, Distributed Objects, Edge Services, Performance, Wide Area Networks (WAN).

1 Introduction

The emerging edge services architecture distributes web services to a collection of edge servers across WAN and near end users to process requests [1, 5, 8, 44, 47]. This approach minimizes communication across the wide area network during request processing 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 [16, 35, 53]. Furthermore, although 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 [25].

Many systems for business logic (code) distribution and execution at edge servers have been built [2, 5, 8, 44, 47], 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 are 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 they are subject to network **P**artitions. As a result, distributed code is used for caching and content assembly [1, 8, 44] but seldom used for replication of web services with a rich mix of reads and writes.

Our goal is to build an edge service replication architecture using application-specific dis-

tributed objects [45] 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. However, if business logic were distributed, accesses to a central database would become costly remote operations. Therefore, our edge service architecture replicates both business logic and data to edge servers by encapsulating the service's shared data within application-specific distributed objects that manage this distributed state. 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 interpose between the 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, 26, 40] to manage data replication and consistency.

In this paper, we demonstrate that an edge services architecture is feasible for the TPC-W benchmark [15], which simulates an online bookstore. 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 develop five simple distributed objects to manage the consistency of different subsets of the distributed bookstore's shared data. The *catalog* object maintains 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 multiple locations and process them at the backend server. This object takes advantage of the fact that many nodes write orders but only one needs to read them, and it exploits the loose requirements on sequencing updates across nodes. The *profile* object represents the user profile information. It takes advantage of the low concurrency of accesses to each record, and it makes use of field-specific reconciliation rules [42]. The *inventory* and *best-seller-list* objects track a bookstore's inventory and best seller lists. The *inventory* object exploits the fact that edge servers care about whether the inventory is zero but do not need to know the actual value. The *best-seller-list* object takes advantage of the fact that a few purchases of a non-popular book do not necessarily alter its ranking.

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; and because requests are satisfied locally at edge servers, the response time is better than that of the centralized system. 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 permits us to make simplifying assumptions in implementing the objects' consistency protocols.

We construct and evaluate a prototype system based on Apache web servers, Tomcat Servlet engines, the JORAM implementation of the Java Message Service, and a DB2 database, and we find that the prototype 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, for the experts constructing the distributed objects, the restricted interface makes it easier to build distributed objects with the ability to handle consistency than to write reconciliation rules [42] 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 [31, 32, 42, 45, 51], 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 a focus on the five 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.

2 TPC-W Background

TPC Benchmark W (TPC-W) is an industry-standard transactional web benchmark that models an online bookstore [15]. 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 a 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, such as shopping cart activities and customer registrations. The *shopping scenario* consists of a mix of 80% browsing interactions and 20% ordering interactions. The *ordering scenario* comprises equal amounts of browsing and ordering. For scalability measurements, the benchmark defines the number of data entries which include numbers of unique books, registered customers, and book photos of various sizes.

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

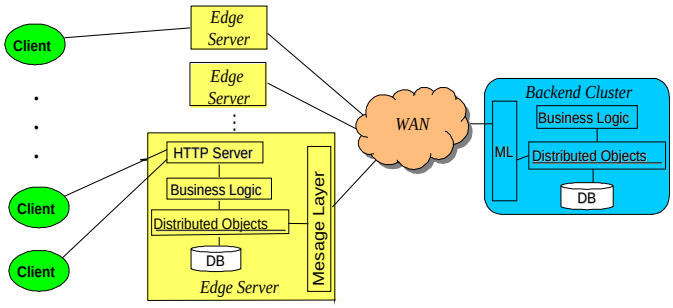


Figure 1: The edge services architecture

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 server, 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, 18, 44, 50], 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 executes 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 [45]. Distributed objects use JDBC to operate on the local database and use the messaging layer to communicate with instances on other servers.

The messaging layer uses persistent message queues [14, 26, 40] 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 logs messages on the local disk before attempting to send them. Upon the arrival of each message at its destination, the destination's messaging layer instance 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 [27], 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 [43] to fit in our object-based edge service architecture. We add five 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 resolve 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 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 set of priorities are, therefore, availability and latency 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 that restricts the range of *observable* behaviors by the memory system [19] is a high priority because a key challenge in any

relaxed consistency system is reasoning about subtle corner cases. Increasing the strength of consistency guarantees makes this reasoning more straightforward for system designers. Simplicity is important for making the approach useful in practice by making the system feasible to understand, build, and deploy. The third and lowest set of priorities is optimizing resource usage such as network bandwidth, processing power, and storage. Therefore, 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 [52], and this assumption also simplifies the design of distributed objects. Because 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, and we bypass a number of potentially attractive optimization options for each distributed object. 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 hundreds or thousands of replicas. This trust model is reasonable in the environment where the service provider owns and manages geographically distributed service replicas, and it also is 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 although our current prototype does not encrypt network traffic.

3.3 Distributed objects

Distributed objects may be a simple way to achieve high availability, good performance, and good consistency compared to a more general shared data interface for two reasons. First, the restricted interface of a given object encapsulates the internal state of the object and may prevent data incon-

sistencies from being observed. Second, the fact that the workload may be known to each object allows the data replication protocol used by the object to exploit the specific workload characteristics to improve availability, responsiveness, or consistency.

In this section we discuss the design of the key distributed objects in our distributed TPC-W system. We seek to demonstrate insides on how application specific designs of data replication/distribution can enhance the availability and responsiveness of the system.

3.3.1 The catalog object

The *catalog* object provides 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 [41] (aka PRAM consistency [29]) 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, then 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 [32]. 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, FIFO consistency 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 simple 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 simple strategy 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 *data dissemination* semantics. For example, a *dissemination* object could provide a mechanism for propagating edge service infrastructure information such as program or configuration updates. Similar behaviors can also be found in other applications such as IBM's geographically-distributed sporting and event service [9], traditional web caching, content assembly, dynamic data caching [10], and personalization. Systems may benefit from additional features/optimizations under different workloads. We discuss three of such features/optimizations that may be useful to other distributed applications, but that are not included in the design of the *catalog* object.

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 [17, 21, 49] for each update. Of course such a feature may reduce 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 [32].

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*, 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 guarantee that all orders are 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. In such a case, because the message handler did not complete, the messaging layer invokes the handler again during recovery. The handler also detects duplicates when it processes an order. In that case, it executes 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, because 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% [15]. 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). Therefore, we require an implementation of the *profile* object to allow edge servers to access any profile.

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" [34, 42] 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 and then sent to the other edge servers. If a set of edge servers is disconnected at the time of the update, the persistent 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 reconciliation rules at the field level. For example, the object-specific reconciliation rule for the last-order field of a profile record is to compare the orders' timestamps and to select the more recent order; the rule for credit card records or shipping addresses is to merge multiple updates and prompt the user for selection when the client makes a subsequent 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 [22], 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. Because the profile objects are small and updates to them are infrequent, partial replication would modestly reduce overhead and might hurt performance, availability, or simplicity. However, systems with large numbers of replicas could see benefits from more sophisticated partial replication.

A wide design space exists for providing consistency on read/write objects in distributed systems [41], 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 Bayou's anti-entropy [36]) 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 [42].

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.

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 *in-*

ventory 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 these two servers exceeds a certain threshold, the inventory instance at the backend server requests inventory re-distribution between them. In this edge server pair, the one with higher inventory is the donor and the other is the recipient. Note that such a re-distribution 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 a persistent messaging layer greatly simplifies the design of this 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. However, "false positives" only occur under some extreme conditions as illustrated in Section 4.4. Furthermore, we can eliminate those subtle cases with enhancements described below that we considered but did not adopt in our implementation for the purpose of simplicity.

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: 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 [51]. Unlike TACT, our system adjusts the

local inventory with a centralized coordinator for simplicity. We could change this 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 selling books for each subject. In TPC-W the best sellers are the fifty most popular books computed for each subject based on the 3,333 most recent orders with each order containing up to 100 books.

The interface of this object includes a read operation that takes a *string* as the subject and returns a list of best selling books under the subject. The best sellers 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 may not change the lists. 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.

Object	Object State Replication	Updates Propagation	Concurrent Updating Rules
Catalog	all records at all servers	backend ⇒ all edges	n/a
Order	1/N at edge; N/N at backend	edges ⇒ backend	timestamp ordering
Profile	all records at all servers	all edges ⇒ all edges and backend	field-level reconciliation rules
Inventory	local view at edge; all local views at backend	on threshold: an edge ⇒ backend ⇒ an edge	on threshold: timestamp ordering
Best-seller-list	approximate view at edge; current view at backend	on threshold: backend ⇒ all edge	on threshold: timestamp ordering

Table 1: Distributed object state replication and propagation.

3.4 Issues

As mentioned before, distributed objects are designed based on the specific application semantics such that they hide the complexity of WAN data replication and consistency with simple interfaces. In addition, objects provide consistency guarantees that are straightforward and easy to reason about for both developers and users of the objects. Our distributed TPC-W system works well using distributed objects because the consistency guarantee of one object has little dependency to other objects. However, the assumption we used in building the distributed TPC-W system may not be held in other distributed WAN applications/services. One important future work is to precisely characterize the system consistency guarantees in presents of interactions among different consistency models.

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, and 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 Bayou's session guarantees [42] to ensure that clients always communicate with sufficiently updated servers. In this example, we would need to bring the state of *se2* up to that of *se1* before allowing the client to interact with *se2*. We will consider this feature in our future work.

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

4 System Evaluation

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

4.1 Environment and implementation

To demonstrate our distributed bookstore system, we deploy a prototype across four 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. However, in order to simulate a wide area network (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 [33]. 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 network links in a local area network (LAN). 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 pre-defined workload against each edge server. TPC-W defines three workload 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).

One of our goals was simplicity. It is difficult to precisely characterize the extend to which the goal was met. Qualitatively, the designers regard the system as easy to understand. As a very rough quantitative measure, we note that the latest distributed TPC-W implementation consists of 6600 lines of source code, which is about 1000 lines more than that of the centralized version. The additional source code is primarily for the implementation of the specialized and simple replication protocols employed by distributed objects for managing shared data.

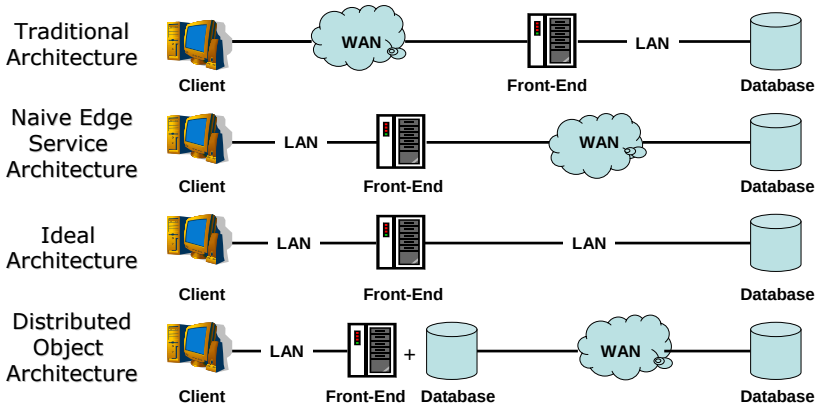


Figure 2: The network configuration of WAN service architectures.

4.2 Performance

In this section, we evaluate the performance of our distributed bookstore system with respect to two criteria: latency and throughput. As noted in Section 3, our most important performance goal is to minimize the system latency because latency alters the expensive "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 benchmark on four architectures as illustrated in Figure 2. We use one frontend machine and one backend 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 *naive 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 production deployment.

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 of the ideal architecture and is significantly better than that of the traditional architecture or the naive edge server architecture. 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.

We measure both system throughput and response time while varying the request rate. 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 the maximum system throughput is reached and the system becomes saturated, at which point the response time will increase sharply.

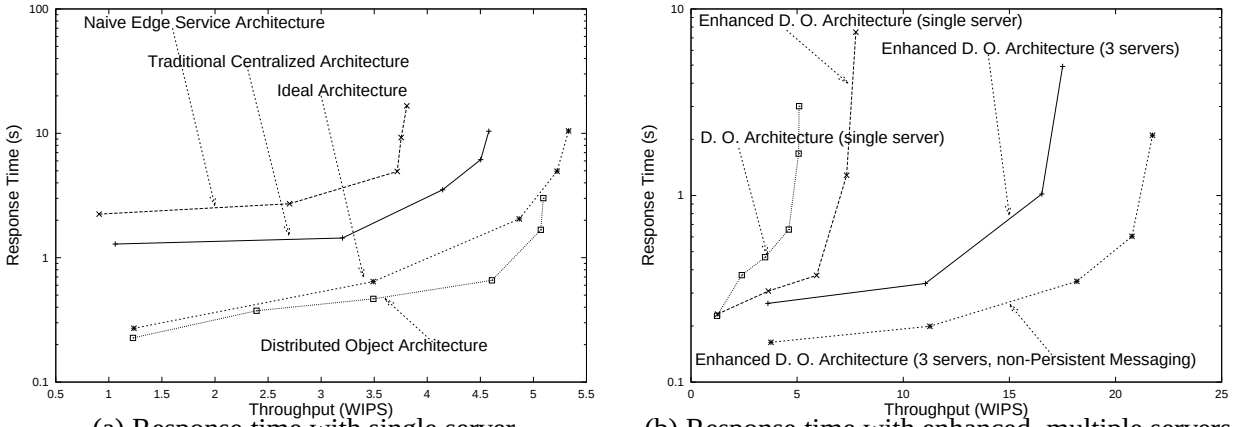
In Figure 3, curves from graph (a) indicate the performance of four architectures when using only one edge server. Graph (b) shows the performance of the distributed object architecture with additional enhancements and the scalability of the enhanced architecture when running on two different messaging layers, the JORAM messaging layer and the quick messaging layer. In both graphs, the x-axis represents the throughput in WIPS (web interactions per second), and the y-axis represents the response time of the bookstore application deployed on different architectures.

First, we explain the curves in graph (a) from the top to the bottom. The top curve represents the response time for the naive edge service architecture. This system experiences the worst minimum response time of 2.42s/req because a client request to the edge server usually triggers multiple requests from the edge server to the central database at 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 1.25s/req. The third curve from the top indicates that the response time of the ideal architecture improves response time by nearly another factor of five, to 0.26s/req. The response time of the distributed object architecture is

slightly better than that of the ideal architecture while both architectures yield approximately the same maximum throughput of 5.2WIPS, as indicated by the forth curve from the top in graph (a). The slight improvement of response time for the distributed object architecture is due the caching of *Shopping Cart* information at the edge server. (In all three other architectures, *Shopping Cart* information is stored in the central database only).

After demonstrating the excellent performance of our distributed object architecture with one edge server, we evaluate the performance of this architecture with multiple edge servers. Before we add more edge servers to the system, we try to reduce implementation specific bottlenecks that may limit the system performance when heavier workloads are applied. In the subsequent tuning process: (1) we double the size of the main memory to accommodate message buffering; and (2) we cache the best-seller lists in each distributed object instance instead of computing them from the local database for client request. Note that both of these improvements are orthogonal to our decision to use distributed objects and the similar optimizations are applicable to the other architectures as well. The performance improvement of the enhanced architecture is illustrated by the second curve from the left in graph (b). Comparing with the first curve from the left in graph (b), which represents the performance of the original distributed object architecture, the enhanced architecture shows a consistent response time at approximately 0.26s/req under the moderate workload and an improved maximum throughput of nearly 8WIPS. After we add two more edge servers, the maximum system throughput of the enhanced architecture increases to 17.5WIPS as shown by the third curve from the left. Notice that we do not achieve a full linear improvement in throughput by adding two edge servers. This speedup shortage is due to both the overhead for the message logging and the inefficiency of the JORAM messaging layer implementation. We also implement a quick messaging layer that sends messages without logging them to disk first. The maximum throughput of the system with three edge server reaches 21.74WIPS when using the quick messaging layer for the message exchange, as illustrated by the bottom curve in graph (b).

The edge service architecture sends all updates to the backend server, which ultimately limits its scalability of throughput. However, two facts allow adding more machines to increase system throughput. First, the read operations, which constitute more than 50% of the workload, are distributed among edge servers. Second, technology exists to make the backend database scalable, and indeed current centralized architectures achieve good scalability by directing all of their queries to a



(a) Response time with single server (b) Response time with enhanced, multiple servers

Figure 3: System response time as the workload increases.

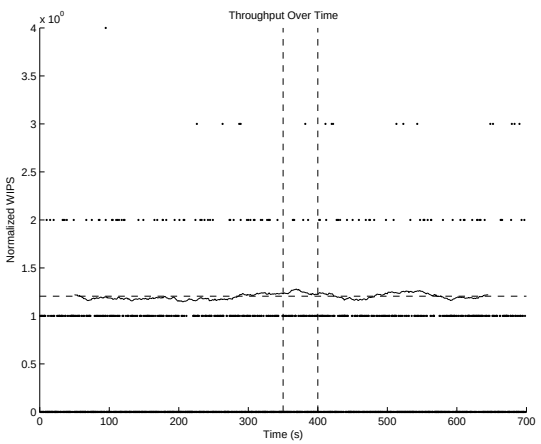
scalable backend database. We believe that the distributed architecture approach should be viewed as a way to increase availability and improve latency while scalability of throughput is improved with cluster technology.

The throughput of our distributed TPC-W bookstore system is competitive with that of other academic systems [3, 20, 43]. If we assume that a typical Pentium III machine costs roughly $800, the price/performance cost of our system is roughly 147.19-182.85$/WIPS, which falls in the range of published standard industry TPC-W performance results, 24.50-277.80$/WIPS [48]. The throughput of the enhanced distributed object architecture is primarily limited by the throughput of its underling databases, which is not the concern of our current investigation.

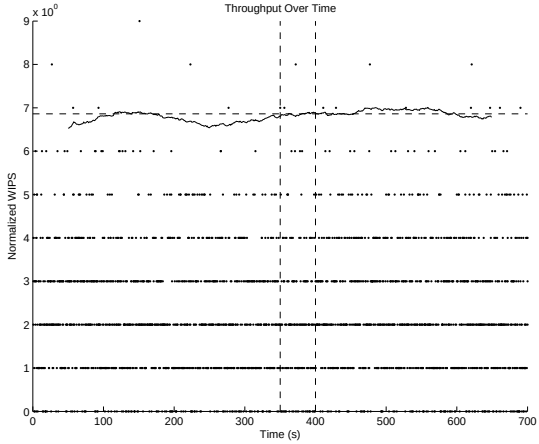
4.3 Availability

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*.

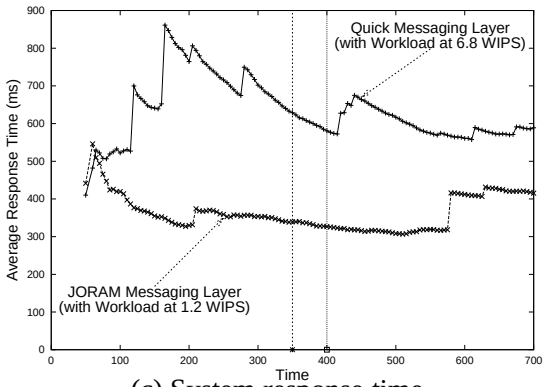
Figure 4 shows the system throughput, average response time, and message queue lengths when the system uses either *JORAM Messaging Layer* or *Quick Messaging Layer* before, during, and after a network failure. Each run lasts for 700 seconds, and a network outage occurs roughly 350 seconds after the experiment starts and lasts for 50 seconds. During the network outage, no server



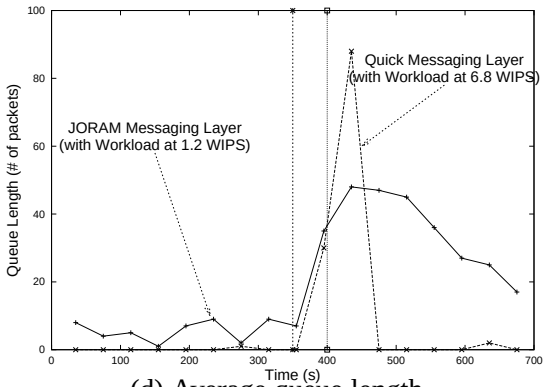
(a) System throughput (JORAM Messaging Layer) with Workload at 1.2 WIPS



(b) System throughput (Quick Messaging Layer) with Workload at 6.8 WIPS



(c) System response time



(d) Average queue length

Figure 4: 700-second session with network outage lasting for 50 seconds.

can communicate with any other server, but the normal communication among servers resumes once the network is restored. To provide a moderate load that does not cause queues to develop before the network fails, we apply a workload of 1.2WIPS to the system that runs on the top of JORAM and apply a workload of 6.8WIPS to the system on the top of the quick messaging layer.

Graph (a) and (b) in Figure 4 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 dashed line represents the average throughput of the 700-second session and the solid slightly wiggly line represents the running average of throughput over 50-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 4 (c) shows the 50-second average response time for the two systems, one running JORAM messaging layer, the other running 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, the curves in this graph tend to fluctuate throughout the session.

Figure 4 (d) shows 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 the JORAM Messaging Layer clears out queued messages relatively slower because it has a 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 vendor 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 a stale catalog and stale best seller lists, accepting orders with stale inventory which may cause false-positive back-order rate, and delaying orders to be processed at the backend server by buffering them on local disks. These trade-offs seem appropriate and acceptable for this application.

4.4 Consistency

Because the system slightly relaxes consistency for higher availability and performance, users may view stale information even during normal system operations. In this section, we evaluate

the impact of the relaxed consistency model on the distributed bookstore system, during normal operations, by examining the staleness of local best-seller lists and local inventory.

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 false-positive back-order rate under a condition where the inventory is low and workload is unbalanced. We expect the false-positive 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 5 shows the percentage of orders resulting in false-positive back-orders due to the in-

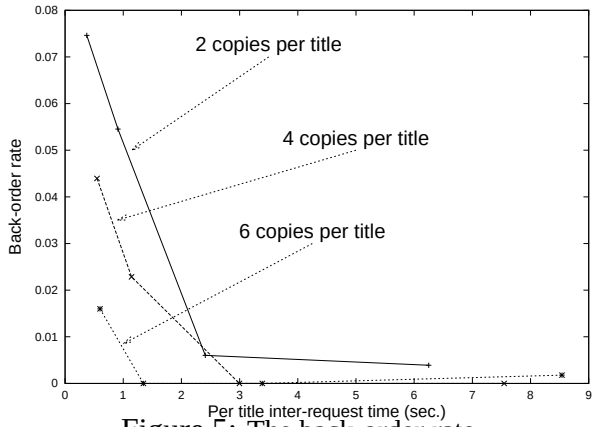


Figure 5: The back-order rate.

ventory 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 true back-order rate of orders) 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 false-positive 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 WAN for computing and re-distributing inventory.

It is worth noting that the small back-order rate shown in Figure 5 only represents the system consistency in the extreme cases where inventory is small, 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 the system inconsistency.

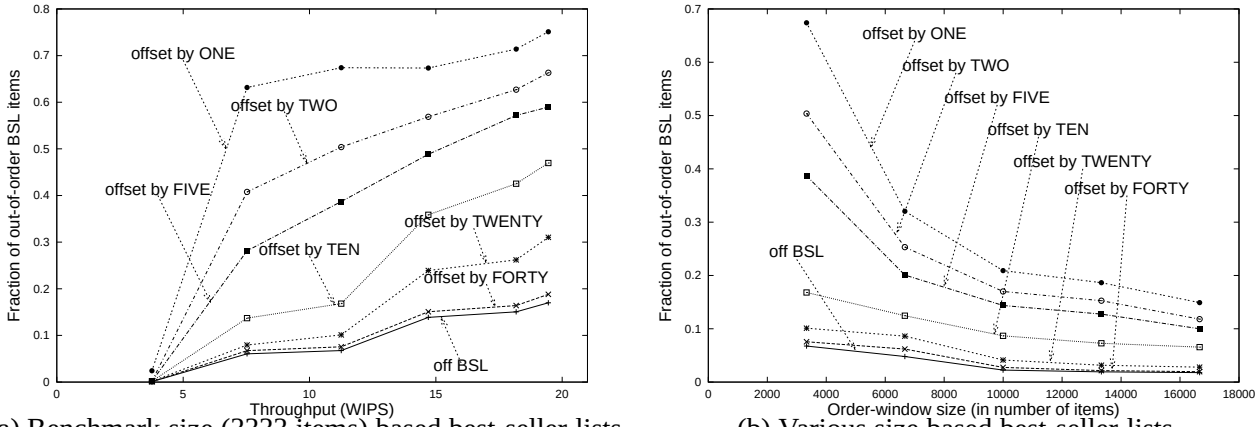
Local best-seller lists: To maximize the availability and performance, every front-end in our system keeps a local copy of best-seller lists, which refer to the fifty most popular items in the most recent *order-window* in every category. Note that an *order-window* refers to a given number of purchases completed. The back-end monitors incoming orders that can potentially alter best-seller lists. Whenever incoming orders trigger a change in best-seller lists at the back-end, the back-end multicasts the change to all front-ends. However, changes may not immediately be reflected at all front-ends because of the asynchrony of updates to front-ends. Because front-ends always serve clients with their local copies of best-seller lists, stale best-seller lists are sometimes returned to clients. In this experiment, we evaluate the effect of such a lazy update approach on best-seller lists in the distributed bookstore, and we show that the amount of stale best-seller lists served by front-

ends is small and tolerable under two conditions: (1) moderate workload that is within the system steady-state throughput; and (2) a reasonably sized order-window for computing best sellers.

Figure 6 shows, in the best-seller lists served to clients, the fraction of items that are out of position relative to their positions in the best-seller lists at the back-end. For this set of experiments, we use 3 enhanced edge servers all running atop the non-persistent messaging layer. Graph (a) shows that with the TPC-W defined order-window size of 3333, the fraction of out-of-position items increases as client workload increases. Graph (b) illustrates that, given the workload of 11.5WIPS, the fraction of out-of-position items decreases when the order-window for computing best sellers increases.

In graph (a), the x-axis represents the system throughput in WIPS, and the y-axis represents the fraction of client-received items that are out of position with respect to their positions in lists at the back-end. In this experiment we use TPC-W defined parameters: the order-window for computing best-sellers is 3333 and each order purchases up to 100 items [15]. Each curve in the graph indicates the fraction of client-received lists with items that are off their original positions at the back-end by at least the specified number of positions. For instance, the top most curve shows the fraction of items that are off their original positions by at least one. Respectively, curves below the first one indicate the fraction of list entries displaced by at least two positions, five positions, etc. The bottom most curve shows the fraction of items that are not in the client-received best-seller lists but are in the lists at the back-end. Notice that when the workload is light (e.g., less then 5WIPS), we see few (less then 1%) out-of-position items returned to clients. Under a light workload there is only a small probability for a best-seller query to closely follow a purchase request that alters the best-seller lists, but as the workload increases, this probability increases. Under the high workload the fraction of out-of-position items is relatively high, e.g., the fraction of items with minimal off-distance of one is around 70% and the fraction of items that are entirely displaced from the best-seller lists is between 10% and 15%. However, more then half of the out-of-position items have an off-distance of less than 10.

This relatively high inconsistency ratio is due to the small size of the order-window: an order-window of 3333 purchase transactions spans only the past few minutes, and when the order-window is small, the probability that a new order can change the best-seller lists at the back-end is high. Graph (b) illustrates that as the order-window becomes reasonably large, the fraction of



(a) Benchmark size (3333 items) based best-seller lists (b) Various size based best-seller lists

Figure 6: Staleness of local best-seller lists subject to workload & the base size.

out-of-position items decreases. The x-axis represents the order-window size and the y-axis still represents the fraction of client-received items that are out of position with respect to their positions in lists at the back-end. Similar to graph (a), each curve in this graph represents the fraction of items with some minimal off-distance. Those curves indicate that when the order-window increases from 3333 (about 50 minutes) to 16665 (about 250 minutes) under a fixed workload of 11.256WIPS, the fraction of out-of-position items decreases by 75% on average, e.g., the fraction of items with minimal off-distance of one drops from 67.4% to 14.9% and the fraction of items that are incorrectly moved off the best-seller lists drops from 6.76% to 1.77%.

This result suggests that a system that takes a lazy update approach for maintaining best-seller lists can provide clients with the consistent data when the size of the order-window is sufficiently large. In practice, online bookstore systems usually track orders over days or weeks for computing best-seller lists, and the fraction of out-of-position items will be smaller in such systems. We should note that even in cases where inconsistent best-seller lists are returned to clients, most client-received lists are useful since most items are only out of position by a small distance.

5 Related work

The idea of taking advantage of type-specific properties in managing replicated data is also proposed by Herlihy [24]. In his work, Herlihy introduces a replication technique that systematically exploits properties that are specific to data types and derives constraints on building a quorum-based replication system. Herlihy argues that understanding type-specific properties can reduce the constraints on the availability of the replication system. This idea is similar to ours except

that we focus on applying such techniques to e-commerce applications and do not use quorum approach for building our replication framework.

Our approach is similar to those in other systems [23, 30, 37, 39, 45] which propose to encapsulate the complexity of structures and computations in distributed systems with objects. Both Gribble [23] and Litwin et al. [30] use distributed data structures to abstract the implementation details and provide a scalable and efficient data sharing and distribution in a cluster environment. Porcupine [37] is a mail service that runs on a cluster of commodity PCs. This mail service consists of a collection of data structures, each of which manages the associate shared data with specific policies to achieve the desired overall system manageability, availability, and performance. Shapiro [39] proposes to structure a distributed system as a set of services or subsystems, each of which may be made of a number of communicating objects across the network. But this work focuses on low level components such as system processes, services, and resources. Globe objects [45] are application level objects specifically for the WAN data replication. They are similar to our distributed objects in that they both allow application programmers to make use of pre-constructed replication modules to easily invoke standard consistency algorithms with different objects and let programmers exploit application semantics in the design and implementation of individual objects. This distributed objects model provides a flexible and powerful way to build distributed applications in WAN. 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. 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". We found it simpler to integrate the code for object semantics and replication consistency. Future work is needed to see if our consistency algorithm can be modularized 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 objects by eliminating the need to ensure reliable message delivery at the object level.

Garcia et al. [20] 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 [43] 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] study 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 [22]. 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 [34].

Our *order*, *inventory*, and *best-seller-list* objects take advantage of the fact that updates are commutative and can be slightly reordered before the threshold is reached. The value of commutativity for simplifying consistency has also been used in write-anywhere databases [22].

Nayate et al. examine data dissemination services with self-tuning, push-based prefetch from the server [32]. In this work, the authors argue for separating invalidations from the actual updates of the shared data. Synchronizing invalidation messages arriving at replicas enforces the system's consistency requirement, and prefetching updates in the background maximizes the hit rate at replicas, minimizes the response time, and maximizes the service availability. We could incorporate this approach to enhance the *catalog* object.

Previous studies [28, 36, 38, 51] have explored the space of relaxed consistency models. Ladin et al. [28] propose a lazy replication technique to increase the system availability and performance with the guarantee for causal ordering. All updates are processed asynchronously while queries

are processed in a sequence that reflects casual ordering with support from both the information in the *label* associated with every query and the *gossip* process among replicas. The Bayou [36] replication framework uses an anti-entropy protocol to guarantee the eventual consistency of the system, and it uses version vectors and application-specific reconciliation to ensure client consistency. Saito et al. [38] focus on minimizing the space and the computation overhead using the optimistic replication approach to provide the eventual consistency. TACT [51] 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. Both Bayou and TACT provide hooks for application developers to attach specific reconciliation rules to resolve update conflicts [42]. The design of some of our distributed objects make use of these ideas.

Walsh 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 [46]. 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.

6 Conclusions

Our TPC-W bookstore is built using a distributed object architecture and appears to provide high availability and good performance. The throughput and response time of our system are consistent before, during, and after network partition. By measuring all WAN latencies of four architectures, we show that the response time of our system closely approximates that of the ideal system under a normal workload.

Replicating shared data everywhere might be seen limit the system's scalability. We do not view the distributed architecture approach primarily as a way to improve the system scalability in terms of throughput, but mainly as a way to increase availability and improve latency. Still, edge services architecture provides opportunities to improve throughput by processing reads at edge servers and absorbing bursts of updates in edge servers' message queues for deferred processing.

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 software engineering strategy is to combine WAN availability, performance, and consistency expertise with semantics to craft distributed objects for a class of distributed application/services on the Internet. In our particular case, we provide WAN replication primitives for building distributed e-commerce applications.

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.

References

[1] Inc. Akamai Technologies. Akamai-The Business Internet - A Predictable Platform for Profitable E-Business. http://www.akamai.com/BusinessInternet/whitepaper_business_internet.pdf, 2004.

[2] Inc. Akamai Technologies. Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite. White paper, Akamai Technologies, Inc., 2004.

[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 Technology*, 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 *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. 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] Transaction Processing Performance Council. TPC BENCHMARK W. http://www.tpc.org/tpcw/spec/-tpcw_V1.8.pdf, 2002.

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

[17] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Lease: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of IEEE Infocom*, March 2000.

[18] 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.

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

[20] D. Garcia and J. Garcia. TPC-W E-Commerce Benchmark Evaluation. *IEEE Computer*, February 2003.

[21] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[22] 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.

[23] 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.

[24] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[25] IBM. *The Economic Value of Rapid Response Time*, pages 11–82. Number GE20-0752-0. 1982.

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

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

[28] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

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

[30] W. Litwin, M-A. Neimat, and D. Schneider. LH* - A Scalable, Distributed Data Structure. In *ACM Transactions on Database Systems*, December 1996.

[31] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

[32] A. Nayate, M. Dahlin, and A. Iyengar. Data Invalidation and Prefetching for Transparent Edge-Service Replication. Technical Report TR-03-44, University of Texas at Austin Department of Computer Sciences, Nov 2002.

[33] NIST Net Home Page. http://snad.ncsl.nist.gov/itg/nistnet/.

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

[35] 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.

[36] 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.

[37] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[38] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Proceedings of the Fourteenth International Conference on Distributed Computing*, October 2000.

[39] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[40] 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.

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

[42] 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.

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

[44] 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.

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

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

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

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

[49] 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.

[50] 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.

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

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

[53] 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.