IEICE TRANS. INF. & SYST., VOL.E92–D, NO.10 OCTOBER 2009

PAPER *Special Section on New Technologies and their Applications of the Internet*

Partially Eager Update Propagation and Freshness-Based Read Relaxation for Replicated Internet Services

Ho-Joong KIM[†a)], ***Student Member and*** **Seungryoul MAENG**[†], ***Nonmember***

SUMMARY We propose an Edge-write architecture which performs eager update propagation for update requests for the corresponding secondary server, whereas it lazily propagates updates from other secondary servers. Our architecture resolves consistency problems caused by read/update decoupling in the conventional lazy update propagation-based system. It also improves overall scalability by alleviating the performance bottleneck at the primary server in compensation for increased but bounded response time. Such relaxed consistency management enables a read request to choose whether to read the replicated data immediately or to refresh it. We use the age of a local data copy as the freshness factor so that a secondary server can make a decision for freshness control independently. As a result, our freshness-controlled edge-write architecture benefits by adjusting a tradeoff between the response time and the correctness of data.

key words: *distributed system, edge service, data replication, freshness, consistency*

1. Introduction

Edge service architecture [2], [3] is widely adopted nowadays in order to reduce response time between client and origin server. Edge server is located geologically close to clients and serves them cached contents which were generated by origin server. Thus it absorbs network traffic and liberates the origin server from heavy loads. As the needs for dynamically generated contents increases, edge server not only caches static contents like movie files but also deploys business logic and performs dynamic contents generation to alleviate server load further as shown in Fig. 1 (b).

But data still remains at origin server in such environment, since frequently changed data are difficult to cache or replicate. Thus edge server should communicate with origin server to manipulate data safely. Such data access causes increase in the response time and hence it diminishes the benefit of edge service architecture.

The solution is to replicate data near the edge like 1 (c). But data replication in a distributed environment needs a synchronization mechanism between the original and the replicated data.

There are two major approaches to handle data replication; decentralized and centralized approaches. While decentralized approaches keep the consistency of data by participation of member nodes, centralized approach lean on a authorized server to maintain consistency.

Manuscript received February 14, 2009.
Manuscript revised May 25, 2009.

[†]The authors are with Korea Advanced Institute of Science and Technology (KAIST), Korea.
a) E-mail: hjkim@camars.kaist.ac.kr

DOI: 10.1587/transinf.E92.D.1986

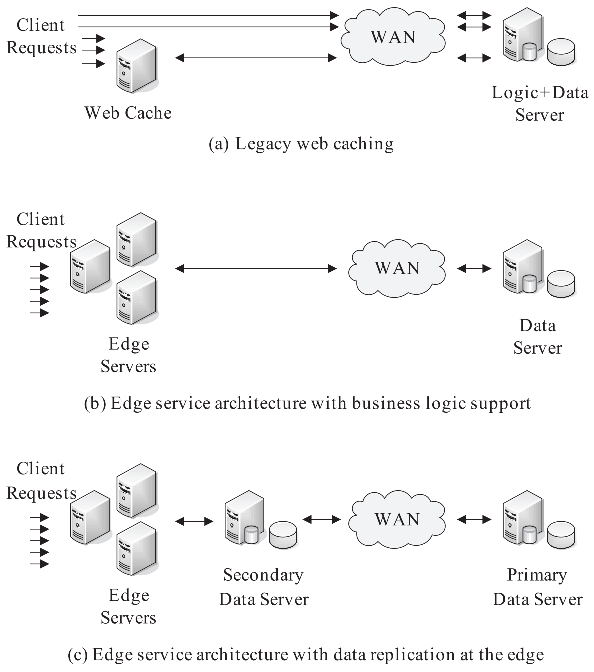


Fig. 1 Edge service architecture and data replication.

In centralized approach, an update request for certain data is handled first at the primary server, which has ownership for the original data, and is then propagated to secondary servers that have replicated copies. Edge service architectures commonly adopt the lazy update propagation technique [6], [11], which reduces synchronization overheads by decoupling data updating at the primary server and propagation to the secondary servers. As a result, a replicated copy at the secondary server is stale until the update is propagated. Due to such staleness, some types of client requests that require correct data cannot be executed at the secondary server. This reduces the efficiency of the edge service architecture, particularly when update requests arrive more frequently.

Weak consistency models such as Eventual consistency [8] can solve the problem. In the eventual consistency model all replicas eventually be identical, but at a certain point one replica can have different value from another. This model is well suitable for numerous Internet applications since it allows a client to manipulate data with less concern for other clients.Session consistency [6], [7] is a practical form of eventual consistency. Session consistency keeps the execution sequence of requests in a single client session in a consistent order.

Unfortunately, lazy update propagation cannot guarantee session consistency, Because data updating and propagation are separated, a client read request at the secondary server might not read the previous update request executed at the primary server but not yet propagated. Thus, additional synchronization is required, such as blocking a read request at the secondary server until propagation has been accomplished.

In the present paper we suggest an edge-write architecture that strengthens synchronization between the primary and the secondary server. This architecture performs eager update propagation for update requests for the corresponding secondary server, whereas it lazily propagates updates from other secondary servers. Moreover, the primary server is hidden from real clients and each client is connected to a secondary server only. When an update request arrives at a secondary server, it is forwarded to the primary server first. The primary server serializes the request with updates pending for other clients and then returns it immediately with any prior update requests on which the request depends that have not yet been propagated to the secondary server. The secondary server executes all requests received in sequential order. By blocking requests, the response time increases in comparison to the immediate response at the secondary server, but the blocking time is limited to the round-trip latency between the secondary and the primary server.

For a read-only request, the secondary server decides between two alternatives: whether to read the possibly stale data immediately, or to wait until data are refreshed. Some distributed systems use a freshness scheme to compare differences between original and replicated data [5], [15], [16]. However, it is hard to compute data differences between secondary servers and the primary server correctly in Internet environments. Besides, many Internet applications tend to have time-related characteristics that are time-sensitive, time-tolerant, or both. We can use the difference in update time between the secondary server and the primary server as a freshness factor. Our freshness scheme regards a replicated data copy to be fresh only when the related update propagation is accomplished at the secondary server. The actual freshness of a data copy is the time elapsed since the last update. This measured freshness value is compared with the required freshness given by the service provider. If the measured freshness is higher than the required freshness, a read request can access data immediately, otherwise the secondary server sends a refresh action request to the primary server. This refresh scheme allows precise freshness control in comparison with primary server-driven update propagation. Our proposed architecture can be used for various and mixed Internet applications having different time constraints.

2. Related Work

There are numerous researches to solve the data replication

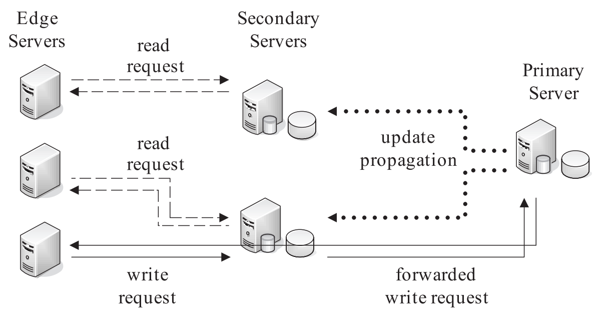


Fig. 2 Lazy update propagation model.

problem in distributed environment. Distributed file systems like Coda [17] and Intermezzo [4] are designed to improve performance and support disconnected operations. But their approach is not well suited to Internet applications which are usually consist of many small independent and interactive requests.

Decentralized replica management is a hot issue in Internet research area. Distributed Hash Table (DHT) [19] gives scalability to distributed systems. It is widely used for searching and maintaining data in P2P networks. Amazon's Dynamo [8] is classified as a zero-hop DHT which offers high availability and good performance for distributed data access. One of Dynamo's key feature is eventual consistency model which allows temporarily different version of data to clients. Decentralized approaches are also strong at failures like temporal server crash or network partitioning.

Primary-secondary data replication schemes [6], [18] are based on traditional centralized system. Although centralized services are weak at single point failure and relatively lacks in scalability, they are intuitive, easy and efficient to maintain data consistency. Primary-secondary replication schemes usually adopt lazy update propagation schemes, as shown in Fig. 2. Whereas eager update propagation suffers from a relatively long response time and heavy synchronization overhead, lazy update propagation has advantages of a short response time and a high throughput [10].

The benefits of lazy update propagation mainly arise from read request handling at the secondary server. In conventional Internet applications, a large proportion of read requests are regarded as allowing some staleness. For example, web content generated by a service provider is uploaded at the origin server first and then propagated to edge servers like the content-delivery network (CDN) [2], [3]. Clients connect to the nearest CDN server and read the replicated news document. Although there might be some differences between the original data at the primary server and its replicated copy at the secondary server until its update is propagated, such differences do not significantly affect the client's perception. If a client request requires fully trusted data, it is served at the primary server rather than at its connected secondary server.

However, such server-driven consistency management model [14], [22] is not suitable for interactive services. With the increasing sophistication of Internet applications, there is more widespread use of interactive data. Clients write and read their own content and share interaction with other clients. As a result, an interactive application generates update requests and propagation messages more frequently. A pure lazy update propagation is not very suitable for such interactive requests.

A major drawback of lazy update propagation is the session consistency problem [6]. In session consistency model any modification of the data induced by a client request should be visible to successive requests by the same client. But the pure lazy update propagation protocol cannot guarantee the session consistency because update at the primary server is decoupled with read at the secondary server. If a read request arrives at the secondary server before its precedent update propagation has been accomplished, the client cannot see the correct result because his modification has not yet been applied to that secondary server. Such inconsistency increases when propagation is delayed or periodic.

Figure 4 (a) illustrates the session consistency problem. Requests from the same client session on some data *d* arrive at the secondary server. Let the first request is composed of a write and its corresponding read ($W1 \rightarrow R1$) and the next request is a read only request $R2$. The arriving sequence at this secondary server is $(W1 \rightarrow R1) \rightarrow R2$, while the actual execution sequence is $R1' \rightarrow R2' \rightarrow W1'$, which is different from the arrival sequence. Since read requests cannot see the modification of previous write request, the session consistency is broken. The dependency between $W1$ and $R1$ can be kept if $R1$ is bounded with $W1$ in the same transaction and executed at the primary server rather than at the secondary server, but $R2$ still cannot see $W1$.

Daudjee et al. solved the session consistency problem in two ways [6]. In one solution, successive read requests are forwarded to the primary server, while the other solution involves blocking successive requests at the secondary server until their precedent updates are propagated. Figure 4 (b) shows the latter solution, where $R2$ is blocked at the secondary server until $W1$ is propagated later. This solution offers session consistency but it lacks scalability when the number of update requests handled at the primary server increases [7]. Therefore, as well as read requests, update requests should be distributed among the secondary servers.

For certain types of applications, update requests can be executed at the secondary server. Requests with low concurrency and high locality can easily be distributed over the network [9]. GlobeDB [18] moves the responsibility for data updating from the primary server to the secondary servers. It divides whole data sets into clusters and assigns a master secondary server to each cluster. An update request to a data cluster is sent to its master secondary server rather than to the primary server. Using data access locality, GlobeDB reduces traffic and loads on the primary server. Although GlobeDB does not guarantee that all secondary servers have identical data sets at the same time, all replicated data copies are eventually consistent, since it propagates all updates in the same order. However, GlobeDB does not fully solve the scalability issue of lazy update propagation, since data partitioning and locality management are additional issues.

3. Proposed System Architecture

3.1 Edge-Write Architecture

As outlined in the previous section, Primary-Secondary data replication model rely on a centralized primary server that is not only responsible for update propagation, but also serves trusted data. Thus, the primary server should handle all complex updates and read requests alone, whereas the secondary servers serve simple jobs for which little consistency management is needed [23]. This imposes heavy loads and traffic on the primary server. If update requests can be executed at each secondary server, the performance bottleneck at the primary server might be alleviated. Although some approaches involve distribution of write requests over secondary servers, applications are limited to low-concurrent data access [13] or high access locality [18].

We propose an update propagation protocol in which not only read requests but also all update requests are executed at the secondary servers only as shown in Fig. 3. We call it edge-write architecture because secondary servers reside at the edge and allows direct write to its replicated data.

When an update request R arrives at the secondary server, it is forwarded to the primary server first, the same as for conventional models. However, the forwarded update request is not executed at the primary server and is only serialized with other update requests from different secondary servers. The request is then immediately returned to its corresponding secondary server with any other update requests on which R depends. On receiving the response message from the primary server, the secondary server executes the update requests it receives in sequential order. Thus, the original update request R is executed last. Until all updates are complete, any read requests that arrived later than R and depending on it are blocked. By serializing all update requests, replicated data at any secondary server are eventually identical to the original data at the primary server. Session consistency is maintained by blocking and serialization of local read requests.

The main difference between our design and the others is that all update requests and corresponding read re-

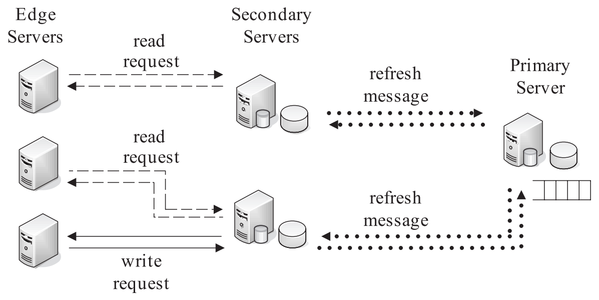


Fig. 3 Edge-write architecture.

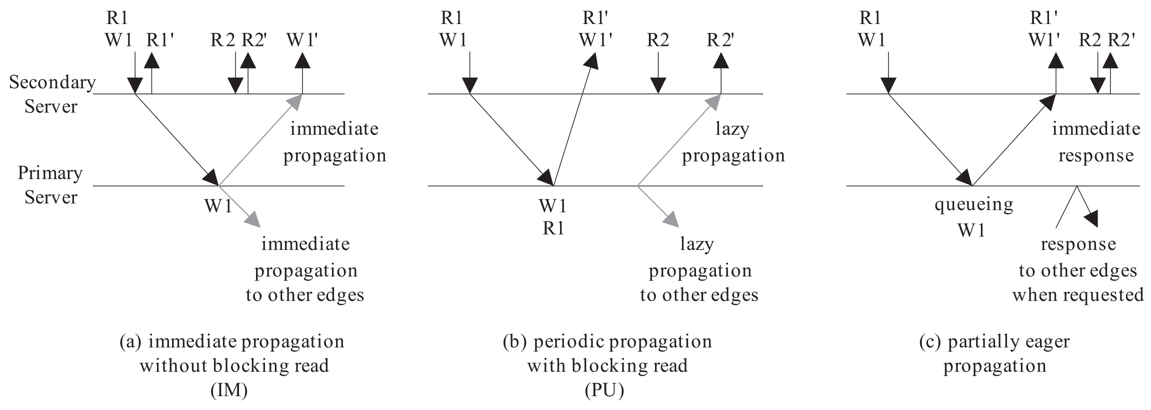

R1
W1
R1'
R2
R2'
W1'
Secondary Server
Primary Server
immediate propagation
W1
immediate propagation to other edges
(a) immediate propagation without blocking read (IM)
R1
W1
R1'
W1'
R2
R2'
lazy propagation
W1
R1
lazy propagation to other edges
(b) periodic propagation with blocking read (PU)
R1
W1
R1'
W1'
R2 R2'
immediate response
queueing
W1
response to other edges when requested
(c) partially eager propagation

Fig. 4 Synchronization mechanisms in the lazy update propagation.

quests for the latter have to be executed at the primary server that has the original data, whereas our proposed architecture forces each update request to be executed locally at each secondary server. By restricting the role of the primary server to sequencing of updates, we eliminate the performance bottleneck at the primary server [6] and distribute requests among the secondary servers. In addition, the blocking latency at the secondary server is not restricted to the primary server's update propagation interval, but is only limited by the round-trip time between the secondary and the primary server. Figure 4 (c) Illustrates how the edge-write architectures works.

3.2 Eventual Consistency and Concurrency Control

Our proposed architecture adopts session consistency, which is a form of eventual consistency [21]. As described in the previous section, each secondary server can see temporarily different version of data copy in the eventual consistency model, although these copies will be finally merged into identical value some time.

Let's see the example Fig. 4 (c). after $W1$ is serialized at the master server with a data $d = d_1$, $R1$ and $R2$ at the secondary server E_1 can read d without any intervention.

If another secondary server E_2 writes $W3\_E2$ on $d = d_2$ at the master server, the change of d is not propagated and the value of d still remains d_1 at E_1. This does not violate the session consistency because a session in E_1 can see its write d_1, but it violates the transactional consistency model. Since the other secondary server can read d as d_1 or d_2, concurrency control is required for the eventual consistency model.

Fully distributed approaches [8] require multi-version control to solve such inconsistency. It requires extra overhead and needs additional programming rather than traditional internet service models.

Our proposed architecture also requires concurrency control, but in highly alleviated manner. Since we serialize all write requests in a primary server, all secondary servers except E_1 who wrote $d = d_1$ can see the latest version $d = d_2$. E_1 does not need to update $d = d_2$ until it requires to update d again. Assume E_1 writes $W4$ where $d = d_3 = d_1 + 1$. $W4$ should be serialized first at the primary server before it is done at E_1. Since the primary server keeps the propagation history d, it can easily find out that $W3_{E2}$ was not propagated to E_1 and solve the conflict.

Although the eventual consistency model requires such extra mechanism, such consistency model can distribute requests among distributed replicas and therefore improve the overall system performance in the most case.

3.3 Freshness

Generally, read requests in Internet services can be categorized into two groups: those that require careful handling with a guarantee of full consistency, and those that can be executed without regarding the correctness. In the latter case, a read request reads replicated data that are possibly outdated at the secondary server. Updates for such data are initiated by the primary server immediately, with a delay, or periodically after the update is accomplished at the primary server. However, sophisticated consistency control can improve the performance of Internet service systems and distribute certain services that are currently considered not to be replicated at the secondary server.

We use the concept of freshness in our edge-write architecture to adjust a tradeoff between the response time and the correctness of data. Commonly, the freshness of a replicated data set D' is defined as the difference between D' and the original data set D. Various freshness metrics can be used, such as version difference or the amount of modified data [5], [12] and the time difference [1], [16]. High freshness means that replicated data are close to the original data, and thus the replication service is reliable to a certain degree.

While most distributed systems focus on maintaining the freshness of replicated data as high as possible, Röhm et al. [16] defined freshness as a QoS (quality-of-service) parameter in an online analytical processing cluster system. Every read request submitted to the centralized scheduler is forwarded to a data server, which satisfies the freshness requirement of the request. When there is no server to meet the requirement, one server is chosen to be refreshed. By allowing different freshness values for each data server, the system can avoid the overheads involved in refreshing the whole server system. Since many Internet applications permit a certain degree of staleness, freshness-based QoS control can improve the performance of Internet servers.

Unfortunately, it is not so easy to apply existing freshness-based schemes directly to our edge-write architecture. Each client's request message is connected to the geographically or logically closest server, rather than first being serialized by a centralized scheduler. Thus, there are only two options for each request: execution at the connected secondary server or forwarding to another server, in particular one with an up-to-date copy.

Since secondary servers are scattered over the wide-area network, it is difficult to compare the freshness of a data copy at a particular secondary server with the original data. Thus, we use the age of a local data copy rather than the amount of difference as a measure of the freshness. A data copy d at a secondary server is considered to be fresh at $t_0(d)$, the instant at which the secondary server received and executed a refresh message regarding d from the primary server. A refresh message consists of zero or more updates on d that have not yet been propagated to this secondary server. Each update for d is executed in sequential order and then the timestamp and the version number of d are updated to $t_0(d)$. If there are no updates to propagate, only the timestamp is updated. This freshness scheme is similar to lease-based consistency management [22], but in our scheme a refresh action is initiated by the secondary server on demand.

As time elapses since d was refreshed at $t_0(d)$, the measured freshness fm decreases. For convenience, we evaluate fm in indexed form. When a read request R for data d arrives at the secondary server at time t_R, the measured freshness $fm(d)$ and corresponding staleness $sm(d)$ can be calculated as follows:

$$\begin{aligned} sm(d) &= \frac{\textit{elapsed time since the latest refresh}}{P} \\ &= \frac{t_R - t_0(d)}{P} \\ fm(d) &= MAX[0, 1 - sm(d)] \end{aligned}$$

P is the period length provided by the service provider. If P has elapsed since the latest refresh, then data copy d is permanently stale and thus expired. fm declines with a slope of $1/P$.

Each read request has freshness requirement $fr(d)$ for accessing data d. Different services might have different freshness requirements for the same data object.

When a read request R having freshness requirements $fr(d)$ arrives at the secondary server E:

- If $fr(d) < fm(d)$, then R is executed immediately at E.
- Otherwise, E sends a refresh message regarding d to the origin server and waits for a response.

Although we define $sm(d)$ as a simple linear function of the elapsed time in this paper, $sm(d)$ can be changed to a complex timing function. Various factors such as the types and characteristics of data and access patterns can be used to adjust the adequate staleness function. Service policy such as SLA (service level agreement) can also considered on the decision of $sm(d)$.

We describe the freshness comparison mechanism and the request flow in detail in the next section.

4. System Design

4.1 Data Partition

Our proposed system replicates a complete set of data to all secondary servers. Data can be also kept at the primary server as an option, but in our model this is only used for backup storage.

A whole data set D is partitioned into n disjoint objects. Each data object $d_i \in D$ in a secondary server with latest refresh time and local update version $d_i(t_i, v_i)$. The primary server also has $d_i(t_i, v_i)$ for each d_i, which is the latest request arrival time and update version for d_i from all secondary servers.

The granularity of a data object can vary from a table entry to a whole database site. When the size of a data object is too small, our system suffers from frequent refresh requests. On the other hand, data objects that are too large may lose the benefit of freshness control and suffer from frequent update interrupts, although the overall freshness of data is kept high. The granularity may be dictated by the service provider, but we expect that the proper size of data objects can be calculated by observing data access patterns and the number of refresh messages.

4.2 Freshness Tables

As shown in Fig. 5, each secondary server maintains two freshness tables: one is the freshness requirement table (FRT) for each service, and the other is the freshness measurement table (FMT) for each data object.

A service is a predefined set of codes used to manipulate one or more data objects. A client request consists of services and the corresponding parameters. For example, "show the current bidding price for a certain item number" is an inquiry service and "show the current bidding price for item #123" is an inquiry request. For service S_i, freshness requirement fr_i is the set $\{fr(d_1), fr(d_2), \ldots, fr(d_n)\}$ for data set $D = \{d_1, d_2, \ldots, d_n\}$. Thus, the FRT is a $m \times n$ array for m services and n data objects.

The FMT is an array of current freshness values for each data object. It contains (t, v, seq) for each data object d_i, where $t(d_i)$ is the latest update time and $v(d_i)$ is the version number. The version number given by the primary server is used to maintain the global update sequence. A measured freshness value $fm(d_i)$ is derived from $t(d_i)$. The FMT also maintains the sequence number seq for each data object. It is used to achieve session consistency by serializing and blocking requests locally. Thus, seq is not visible to the primary server.

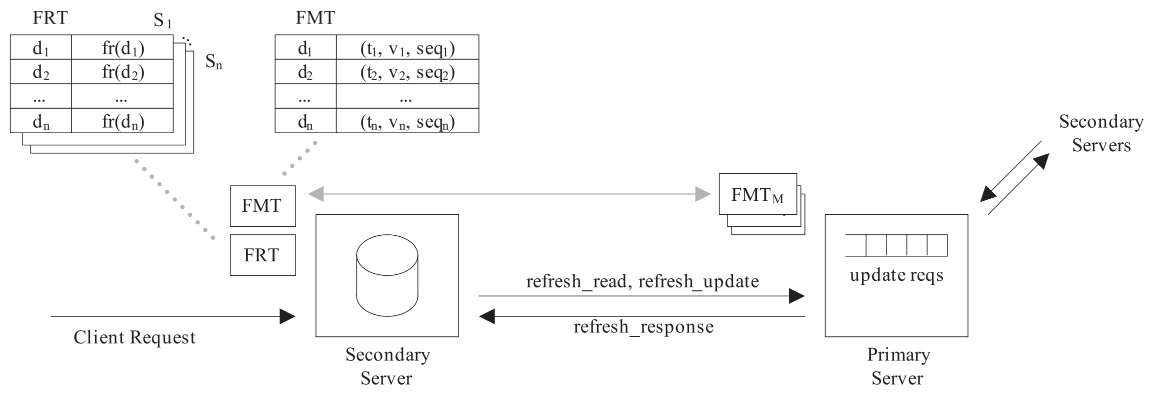


Fig. 5 System design.

The primary server has FMT$_M$ only, which is a collection of FMTs for all secondary servers. It consists of a two-dimensional array of $k \times n$ for k secondary servers and n data objects. Each entry has a couple of values (t, v), which are the latest propagation time and the version number. FMT$_M$ does not keep the sequence number seq because the version number v serializes all write requests arriving at the primary server.

4.3 Types of Client Request

A single client request accesses one or more data objects in the connected secondary server. Every request calls on a service, and thus its consistency requirements are specified in the FRT. The request is immediately executed at the connected secondary server if it is a read request and all requirements are satisfied by the current FMT. If not, the request is blocked until its related data objects in the secondary server are updated to meet the requirements.

A local sequence number seq is assigned to each request. This prevents the sequence inversion that can occur in distributed environments. Session consistency is also achieved using the local sequence number [6]. Even if all freshness requirements are met, a client request should be held until previous update requests in the same client session have been performed.

4.4 Types of Messages between the Secondary Server and the Primary Server

There are three types of messages transferred between the secondary server and the primary server: $refresh\_read$, $refresh\_update$, $refresh\_response$. The first two are requests from the secondary server to the primary server, while the last is the response from the primary server for these two types of requests.

- A $refresh\_read$ message is initiated by a secondary server when the consistency requirement of a data object d_i is not fulfilled for any d_i. This message requests all prior updates for d_i that have not been applied to the secondary server (Algorithm 1).

```
On receiving the client request $R$ at time $t_R$,
$D_R$ = a set of data object $d$ accessed by $R$
if $R$ is an update request then
    $m = refresh\_update(D_R)$
else
    $D_S$ = a subset of $D_R$ where $fm(d) < fr(d)$
    if $D_S \neq \{\}$ then
        /* some data objects should be refreshed
           */
        $fm'(d)$ : an expected $fm(d)$ at $t'_R = t_R + RT$
        $D'_S$ = a subset of $D_R$ where $fm'(d) < fr(d)$
        $D_S = D_S \cup D'_S$
        $m = refresh\_read(D_S)$
    else
        /* all data objects meet fm(d) ≥ fr(d)  */
        Execute $R$ immediately at the secondary server
        Exit
    end
end
Send the message $m$ to the primary server
Block $R$ until the response for $m$ arrives
```

Algorithm 1: refresh requests by the secondary server

- A $refresh\_update$ message acts similar to $refresh\_read$, except this message contains a latest update for d_i.
- A $refresh\_response$ message is the response message from the primary server for the above two types of requests. The primary server packs and returns all update messages on which is d_i dependent in sequential order (Algorithm 2).
- On receiving a $refresh\_response$ message, the secondary server executes all updates in this message and then executes locally blocked read requests (Algorithm 3).

We compare the measured freshness $fm(d)$ and the required freshness $fr(d)$ twice in Algorithm 1. Assume $d_i, d_j \in D_R$ and d_i fails to meet the freshness requirement but d_j is fresh at t_R, then request R should be blocked until t'_R when the response from the primary server arrives and d_i is refreshed. But d_j might be stale at t'_R since it didn't request to be refreshed. Thus we should examine freshness of d'_j once more before sending a refresh request to the primary server in this case. Because a secondary server cannot know

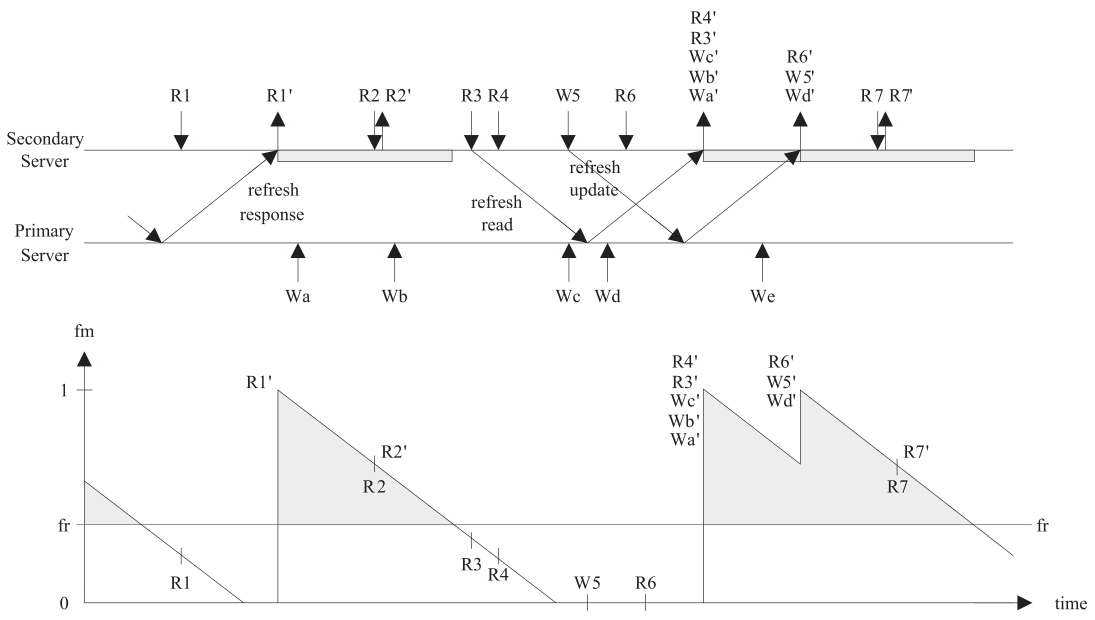


Fig. 6 Types of refresh messages and their flow.

```
On receiving a refresh request $m$ from a secondary server $E$,
$D_m$ = a set of data object $d$ accessed by $m$
if $m$ is $refresh\_update$ then
    if $\exists d_i \in D_m$ is not the latest version in $E$ then
        Solve conflict on $d_i$
    end
    Add $m$ in the global update queue $Q$
end
/* find all enqueued messages dependent with $m$,
   by reverse order                                     */
$Q_m$ = a set of messages $\forall m' \in Q$ not propagated to $E$
$Q_{res} = \{\}$
while $Q_m \neq \{\}$ do
    $m'$ : the latest message in $Q_m$
    $D_{m'}$ = a set of data objects accessed by $m'$
    if $D_m \cap D_{m'} \neq \{\}$ then
        $D_m = D_m \cup D_{m'}$
        Add $m'$ to $Q_{res}$
    end
    Remove $m'$ from $Q_m$
end
/* send all update messages dependent with $m$        */
Send $refresh\_response(Q_{res})$ to $E$
foreach $d \in D_m$ do
    Update $FMT_M(E, d)$
end
```

Algorithm 2: refresh_response by the primary server

```
On receiving $Q_{res}$ which is a queue of update messages,
/* execute update messages in a sequence order  */
while $Q_{res} \neq \{\}$ do
    $m'$ : the oldest update request in $Q_{res}$
    $D_{m'}$ = a set of data object $d$ accessed by $m'$
    Execute $m'$
    foreach $d \in D_{m'}$ do
        update $FMT(d)$
    end
    Remove $m'$ from $Q_{res}$
end
Execute blocked requests
```

Algorithm 3: refresh action at the secondary server

correct t'_R, we use an estimation of the round trip time RT such that $t'_R = t_R + RT$.

When a refresh request message m, which is either $refresh\_read(D_R)$ or $refresh\_update(D_R)$, arrives at the primary server, it requires all previous updates for any data object $d \in D_R$. The simplest way to find out dependencies between m and the queued update messages is to return whole messages queued in the primary server. This also can improve overall freshness of whole data set D, but it might be wasteful when update locality is high. If a secondary server updates a certain data object frequently and another secondary server rarely accesses it, the data object is not necessarily to be propagated to the latter at every update. To find out dependencies between update requests precisely, we travel the global update queue Q in the reverse order as written in Algorithm 2. Even though $D_{m_1} \cap D_m = \{\}$, if $D_{m_1} \cap D_{m_2} \neq \{\}$ and $D_{m_2} \cap D_m \neq \{\}$ then $m_1 \rightarrow m_2 \rightarrow m$. If any dependency between a queued update message m' and a queue of messages Q_{res} is found, m' is added to Q_{res}. Q_{res} is then sent back to the secondary server and executed in a sequence order, as Algorithm 3. Finally, the secondary server can execute any blocked requests accessing $d \in D_R$.

4.5 Example of a Request Sequence

Figure 6 illustrates an example of a request sequence for a data object d in a single client session. The above figure is a flow of request sequence, and the below is the measured freshness value at each event. Shadowed area means $fm(d)$ is high enough so that a read request for d can be executed immediately.

When a read request $R1$ arrives at the secondary server, the required d is not fresh enough. Since one or more refresh request messages are pending already, $R1$ waits until the update is propagated. After the $refresh\_response$ message arrives and is refreshed at the secondary server, The secondary server can perform $R1'$, which is the actual execution of $R1$. $R2$ also accesses fresh data, but when $R3$ arrives at the secondary server, d is stale again. $R3$ initiates a $refresh\_request$ message and waits. $R4$ is blocked again but does not issue a new request refresh message since the response for $R3$ is not yet returned. But an update request $W5$ should initiates another request message. When the response for $R3$ is returned, it contains other updates for d from different secondary servers. After updating Wa, Wb, and Wc, $R3$ and $R4$ can be executed. Execution of $R6$ is not yet allowed since it is dependent on $W5$. Thus, $R6$ is executed after Wd and $W5$ are updated.

5. Evaluation

5.1 Simulation Environments

We made a simulation environment for lazy update propagation architecture models in which several secondary servers and one primary server are distributed over a simulated WAN environment. Clients, which are edge servers in our simulation, are uniformly assigned to each secondary server, and each client communicates only with the connected secondary server. secondary servers communicate with the primary server to update replicated data copies or to forward client requests to the primary server.

Since existing web benchmarks such as TPC-W [20] are designed to evaluate a single site and their data access patterns have low concurrency, we cannot use such benchmarks directly for our environment. Instead, we generate synthetic requests to access shared and replicated data sets. A client generates a request under given parameters and submits it to a secondary server. The simulation parameters are summarized in Table 1.

5.1.1 Simulation Parameters

There are two types of client requests in our simulation: read and update requests. A read request involves a single read operation, whereas an update request is a combination of one write operation and a subsequent read operation. A read operation accesses one or more data objects and a write operation modifies a small entry in a single data object. The default update request ratio is 0.2, which is quite high compared to the TPC-W benchmark. Although TPC-W chooses the read/update ratio for "shopping mix" as 80/20, each update transaction comprises several read operations and fewer write operations rather than a couple of single write and read operations in our simulation.

The execution time for a read operation belongs to a uniform distribution with an average of 0.15 sec and 20% variation. Since a write operation usually involves indexed access to a single entry rather than a ranged access, the execution time is much shorter for a write operation than for a read operation.

We define a freshness normalization factor P of 10 seconds for freshness-based models. This means that any replicated data are permanently stale when 10 sec has elapsed since the latest update. The WAN latency $RT/2$ is 0.5 sec. In the periodic update model, update propagation occurs every 5 sec, which is $P/2$.

Client thinking time is the idle time between two successive requests in a single client session. In our evaluation, client thinking time is an exponential distribution with a mean value of 10 sec. Note that the periodic update propagation interval is shorter than the mean client thinking time. It gives sufficient time for the previous requests in the same session to be propagated, thus a subsequent read request is seldom blocked at the secondary server in the periodic update propagation model.

Table 1 Simulation parameters.

number of CPUs at the primary server	4
number of secondary servers	2–32
number of clients per secondary servers	50
update request ratio	0.01–0.2
execution time for a read operation	0.15 sec
execution time for a write operation	0.01 sec
request handling time	0.001 sec
average WAN latency ($RT/2$)	0.5 sec
freshness normalization factor P	10 sec
periodic update propagation interval	5 sec
mean client thinking time	10 sec
total execution time	1000 sec

5.2 System Models

Besides our freshness-based (FR) model, periodic update propagation (PU) and immediate update propagation (IM) models are evaluated for comparison. The PU model is a traditional lazy update propagation model with guaranteed session consistency, whereas the IM model does not guarantee session consistency.

- **IM** : An update request is divided into write and read operations. Write operations are sent to the primary server and propagated to all secondary servers immediately, whereas subsequent read operations are considered to be an independent read requests and executed at the secondary server immediately without waiting for propagation of previous write operations. The IM scheme shows maximum performance, but fails to maintain session consistency. See Fig. 4 (a).
- **PU** : An update request is forwarded to the primary server and executed, but is only propagated periodically. Subsequent read requests should be blocked until previous update requests have been propagated to the secondary server. PU guarantees session consistency. See Fig. 4 (b).
- **FR** : In our freshness-based scheme, a read-only request is immediately executed at the secondary server only when its freshness requirements are met. If the requirements are not met or the request is updated, it is blocked until the secondary server receives a

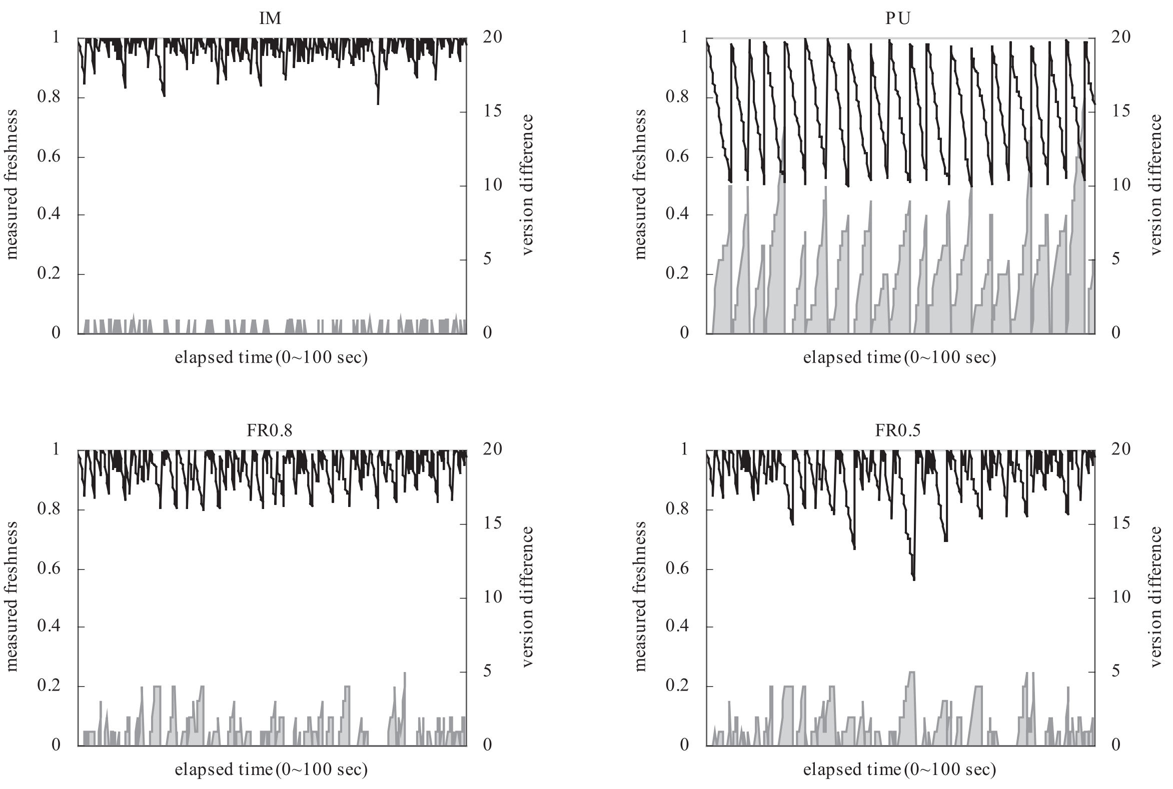


Fig. 7 Measured freshness and version difference (update ratio = 0.2).

refresh_response message from the primary server and refreshes its replicated data. FR guarantees session consistency by additional sequence numbering for each client request. See Fig. 4 (c).

Traditional lazy update propagation architectures are close to PU in that update requests are sent to the primary server and read-only requests are executed immediately at the secondary server, although PU blocks some read-only requests for session consistency reasons.

5.3 Freshness and Version Difference for Base System

We compare measured freshness and correctness for several systems in Fig. 7. There are two secondary servers running, each has 50 clients accessing a single data object with update ratio 0.2. In each figure, the black line above means the measured freshness at the secondary server, and the shadowed region below represents the version difference, i.e., the number of updates queued at the primary server but not yet propagated to the secondary server.

The measured freshness of a data object is boosted to 1 when it is updated, or refreshed in our FR schemes, and then declines with a slope of $1/P$. The two FR schemes exhibit relatively high freshness of replicated data, but the IM and PU schemes show different results. Although IM provides the highest measured freshness on average, the freshness value is not bounded since it depends on the primary server's update events. The value is low when no updates occur in the whole system. However, this does not mean that the secondary server has incorrect data in the IM scheme. If the network is stable, the time skew between updating at the primary server and the secondary server does not exceed $RT/2$. On the other hand, the measured freshness severely fluctuates for the PU scheme due to lazy update propagation. Although session consistency is maintained within the sequence of a single client session, any updates occurring at the primary server are not visible to other sessions that are reading at secondary servers.

Figure 7 also shows the relationship between the freshness and the actual version difference. Lower freshness value means higher version difference between the primary and the secondary server, except IM. IM naturally exhibits little version difference since it propagates all updates immediately to all secondary servers. Unless a read request arrives in bursts or the update propagation delay is too long, the secondary server has nearly the same data as the primary server. On the other hand, PU shows significant version differences. Moreover, an update message is not visible to the different client sessions in the same secondary server until it is propagated. Our FR schemes exhibit fewer version differences and smaller peak differences compared to PU. The variation and peak difference increase when the consistency requirements are relaxed from FR0.8 to FR0.5.

We observed the effect of update ratio in Fig. 8 by lowering the update ratio of one secondary server to 0.1. The other secondary server's update ratio remains 0.2. As compared to Fig. 7, the measured freshness is quite low and *refresh_read* occurs more often because the freshness boost by update request occurs infrequently.

The average measured freshness and version difference are shown in Fig. 9. IM shows low fm when update ratio is low, but it does not mean that data is stale. The value for FR0.8 is close to that for IM, and even FR0.5 shows a relatively high measured freshness and low version difference in spite of its relaxed constraint. It implies that we can achieve both reasonable performance and accuracy compared to the existing data consistency management schemes. Note that PU offers "data updated per 5 seconds" in this case, whereas FR0.5 means "data updated within 5 seconds".

5.4 Number of Refresh Messages Transferred by the Primary Server

Previous results indicate that immediate update propagation is the best approach for obtaining the highest consistency for replicated data at secondary servers. The FR schemes shows closer results to the IM scheme as the freshness requirement fr is restricted to higher values. However, IM cannot guarantee session semantics or stronger consistency because it decouples read and write operation. Besides, the IM performance suffers as the number of update requests increases.

The total number of update requests in the system is determined by the number of clients and the update ratio. Assume that there are E secondary servers and n clients per each secondary server. Each client generates requests at frequency q with update ratio w. Then each secondary server

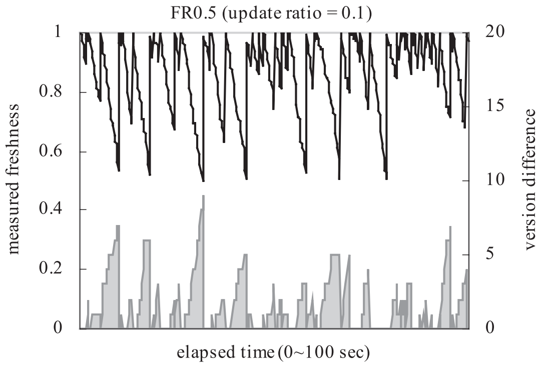


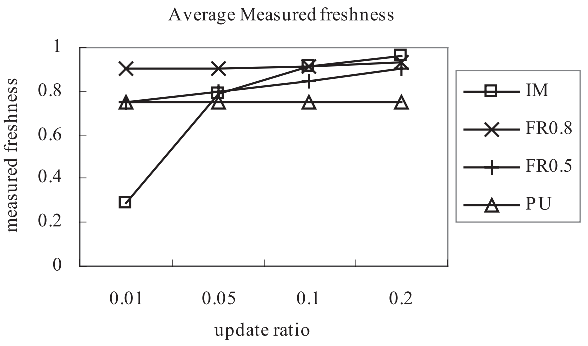
Fig. 8 Measured freshness and version difference under low update ratio (update ratio = 0.1).

receives $n{\cdot}q{\cdot}w$ update requests from its connected clients and send them to the primary server. In the FR schemes, a $refresh\_response$ message is immediately returned to the corresponding secondary server, and thus the primary server generates $n{\cdot}q{\cdot}w{\cdot}E$ messages in total. The PU scheme executes every update request at the primary server, but it propagates updates periodically. It's total number of update requests are $p{\cdot}E$ where p is the frequency of periodic propagation. On the other hand, IM propagates all write operations immediately to all secondary servers. Therefore, the primary server has to send $(n{\cdot}q{\cdot}w{\cdot}E) \times E$ messages in total. As the number of secondary servers increases, this can lead to a bottleneck. Furthermore, it also burdens the secondary servers. Each secondary server receives $n{\cdot}q{\cdot}w{\cdot}E$ propagation messages equally, which is E times larger than for the FR schemes. Such frequent propagation can lead to a decrease in performance, since write operations might invalidate the local query cache, although our simulation does not consider such effects.

When the number of secondary servers is small, the FR schemes generate a little more traffic than IM because both $refresh\_update$ and $refresh\_read$ requests are sent to the primary server. This effect is revealed more clearly when the update ratio is low so that the proportion of $refresh\_read$ messages increases. However, as shown in Fig. 10, traffic does not scale with the number of secondary servers in the FR schemes. Thus, the FR model is much more scalable than IM in terms of traffic and primary server overheads.

Update ratio directly affects to the number of messages in FR schemes. Though High update ratio directly increases $refresh\_update$ messages, it also decreases $refresh\_read$ messages because frequent update makes the replicated data more fresh. This effect is well shown in Fig. 10. In the figure for update ratio of 0.2, total number of messages for FR schemes is high but the difference between FR0.5 and FR0.8 which is caused by $refresh\_read$ is relatively small.

Figure 11 shows, in contrast, the average size per each update propagation messages. The size means an average number of requests to be propagated in a chunk. Since PU gathers requests and periodically send them as a chunk, the size of propagation message is much higher than other schemes. As the number of secondary servers increases or the update propagation period is longer, the size even in-



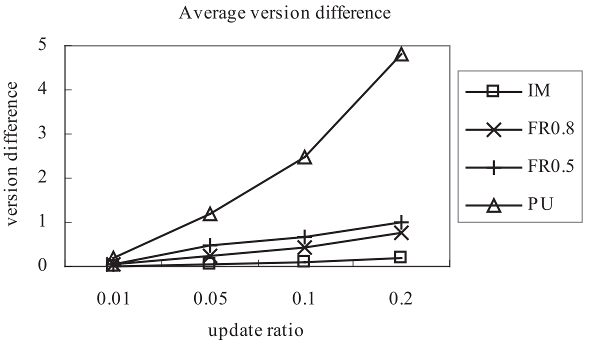


Fig. 9 Effect of update ratio.

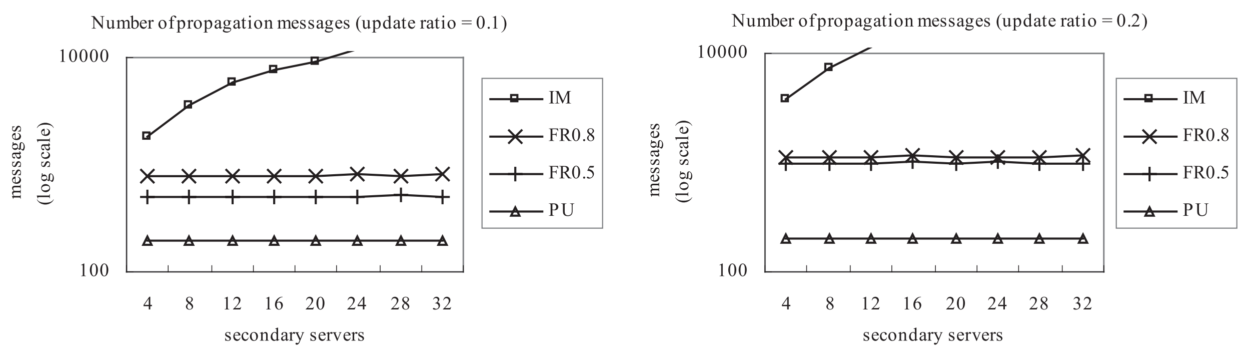


Fig. 10 Number of refresh messages per secondary server.

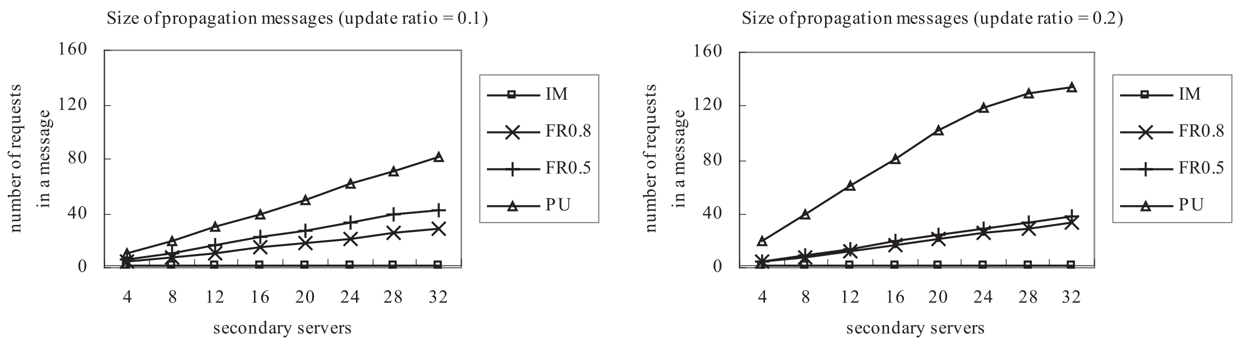


Fig. 11 Size of refresh messages per secondary server.

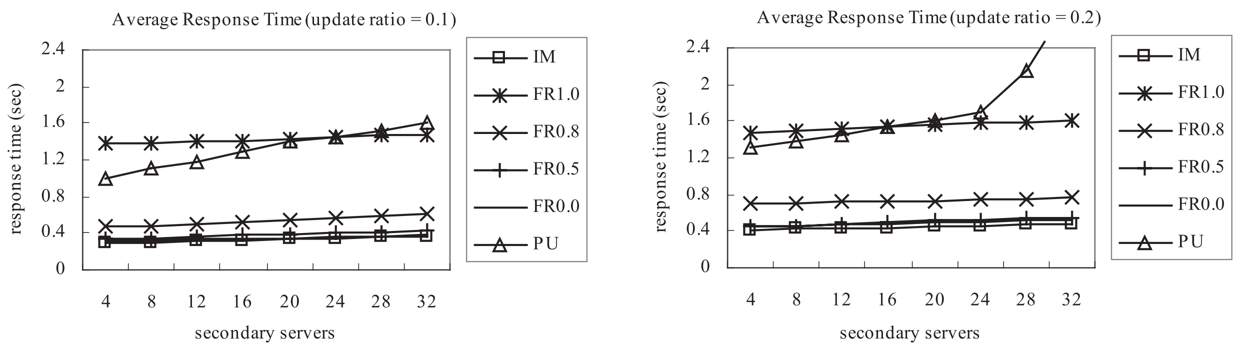


Fig. 12 Average response time.

creases higher. Thus it can be burden to the network at the propagation time.

Moreover, it can cause additional delay because of mass update of secondary server.

5.5 Response Time and Scalability

Figure 12 shows the average response time for update ratios of 0.1 and 0.2 for 50 clients per secondary server. The right figure shows higher response time than the left in general, because it communicates more with the primary server, and thus it blocks more requests.

We added two extreme environments for the evaluation. For FR1.0, every read request requires fresh data, and thus it should be blocked until its corresponding data copies are refreshed, which is the worst-case scenario for our FR scheme. On the other hand, for FR0.0, all read requests are served freely at the secondary server. A read request is blocked only when its preceding requests have not yet been propagated to the connected secondary server to meet session consistency. It is as same as PU in the manner of handling a read request.

Note that the response time for all schemes except PU does not change rapidly as number of secondary servers increases. The IM scheme shows little latency increase because it separates write operations and the corresponding read operations from update requests. A read operation is never blocked to wait for the preceding write operation. Therefore, there is no extra overhead in the IM scheme.

Our FR model blocks not only update requests, but also some read-only requests. As the freshness requirement fr increases, so does the number of blocked read-only requests.

Thus, FR0.8 sends more refresh messages to the primary server and therefore shows longer average response time than FR0.5. However, The number of messages of FR0.8 is much lower than that of FR1.0. This means that a large number of read requests can be absorbed at the secondary server even if the required freshness is relatively high.

There is little difference in the response time of FR0.5 and FR0.0, especially when the update ratio is 0.2. It is because *refresh_read* seldom occurs in this case. Moreover, FR schemes with lower freshness requirements perform close to IM. A little increase in the response time mainly comes from update propagation prior to the blocked read requests.

Interestingly, the number of secondary servers does not significantly affect the FR schemes too. Although an increase in the number of secondary servers also increases the number of write operations at the primary server, the primary server does not propagate its updates unless these are requested by a secondary server.

Although the PU shows slightly better performance than FR0.5 when the number of secondary servers is small, its response time increases rapidly with the number of secondary servers. This is because the PU scheme reads and writes data at the primary server for every update request. As the number of secondary servers increases, a greater number of update requests from clients connected at each secondary server are forwarded to the primary server, leading to a resource contention. This effect is more notable for an update ratio of 0.2. The PU scheme fails to scale with the number of secondary servers or update ratio if the primary server does not have enough capacity.

In this evaluation, we assume that the number of CPUs in the primary server is 4, which means four times capacity than secondary servers like Table 1. It means even PU suffers bottleneck with powerful primary server, thus load at the primary server should be eliminated in order to achieve scalability.

Traditional lazy update propagation architectures like PU benefit by reading at the secondary server under low update ratio. On the other hand, our FR schemes shows relatively high response times in such case because the secondary server refreshes its data copy frequently even if there are no updates. Proper assignment of freshness requirements can solve such false-refresh effects.

6. Conclusion and Future Work

In this study we developed an edge-write architecture in which all client requests are executed at the secondary server. Every update request at any secondary server is serialized at the primary server first, and then propagated eagerly back to the corresponding secondary server. On the other hand, updates from other secondary servers are propagated on demand. Our architecture resolves consistency problems caused by read/update decoupling in the lazy update propagation model. It also eliminates the performance bottleneck of a single primary server, and thus improves overall system scalability, in compensation for increased but bounded response time. Although data version conflict can occur in our eventual consistency model, primary server can easily detect the conflict and solve it. Moreover, write serialization prevents the conflict in most cases.

We also proposed a freshness evaluation scheme that can handle time-restrained read requests for various Internet applications. A read request at the secondary server can either be read immediately or be blocked to refresh the replicated data by comparing its measured freshness with given requirements. Our edge-write architecture offers freshness values that are relatively high compared to traditional periodic update propagation schemes. Furthermore, our scheme ensures certain lower bounds for the freshness of replicated data copies.

In conclusion, our edge-write architecture provides a tradeoff between performance and freshness. This system can be used for various Internet services which have relatively high update ratio. Many interactive applications which were yet to be classified as not replicable or distributable can be classified into this criteria.

Since many factors affect measured freshness, detailed observations are needed to optimize the performance and consistency of the whole system. By alleviating consistency requirements, we can achieve a target level of system performance. In contrast, we can serve fresher data to privileged clients instead achieving a higher throughput. Precise modeling of freshness requirements and measurements will help for the decision between such tradeoffs.

Acknowledgements

This work was supported by the IT R&D program of MKE/KEIT [2009-F-039-01, Development of Technology Base for Trustworthy Computing].

References

[1] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen, “Fine-grained replication and scheduling with freshness and correctness guarantees,” Intl. Conf. on Very Large Data Bases Conference, Aug. 2005.
[2] Akamai Inc., http://www.akamai.com
[3] Amazon AWS., http://aws.amazon.com
[4] P.J. Braam, M. Callahan, and P. Schwan, “The InterMezzo file system,” The Perl Conference 3, O’Reilly Open Source Convention, Aug. 1999.
[5] J. Cho and H. Garcia-Molina, “Synchronizing a database to improve freshness,” Technical Report, Stanford University, Oct. 1999.
[6] K. Daudjee and K. Salem, “Lazy database replication with ordering guarantees,” Intl. Conf. on Data Engineering, March 2004.
[7] K. Daudjee and K. Salem, “A pure lazy technique for scalable transaction processing in replicated databases,” Intl. Conf. on Parallel and Distributed Systems, May 2005.
[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” ACM Symposium on Operating Systems Principles, Oct. 2007.
[9] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, “Application specific data replication for edge services,” ACM Intl. World Wide Web Conf., May 2003.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," ACM SIGMOD Intl. Conf. on Management of Data, June 1996.

[11] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," ACM Trans. Database Syst., vol.25, no.3, pp.333–379, Sept. 2000.

[12] A. Labrinidis and N. Roussopoulos, "Balancing performance and data freshness in web database servers," Intl. Conf. on Very Large Data Bases Conference, Sept. 2003.

[13] W. Li, O. Po, W. Hsiung, K.S. Candan, and D. Agrawal, "Engineering and hosting adaptive freshness-sensitive web applications on data centers," ACM Intl. World Wide Web Conf., May 2003.

[14] A. Nayate, M. Dahlin, and A. Iyengar, "Transparent information dissemination," ACM/USENIX Intl. Middleware Conf., Oct. 2004.

[15] E. Pacitti and E. Simon, "Update propagation strategies to improve freshness in lazy master replicated databases," VLDB Journal, vol.8, no.3-4, pp.305–318, Feb. 2000.

[16] U. Röhm, K. Böhm, H. Schek, and H. Schuldt, "FAS - A freshness-sensitive cooprdination middleware for a cluster of OLAP components," Intl. Conf. on Very Large Data Bases Conference, Aug. 2002.

[17] M. Satyanarayanan, "Coda: A highly available file system for a distributed workstation environment," IEEE Workshop on Workstation Operating Systems, Sept. 1989.

[18] S. Sivasubramanian, G. Alonso, G. Pierre, and M. Steen, "GlobeDB: Autonomic data replication for web applications," Intl. World Wide Web Conf., May 2005.

[19] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-to-Peer lookup serice for internet applications," ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocls for Computer Communications, Aug. 2001.

[20] Transaction Processing Performance Council, TPC Benchmark W (Web Commerce), Feb. 2001.

[21] Werner Vogels, Eventually Consistent - Revisited, http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, Dec. 2008.

[22] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar, "Engineering server-driven consistency for large scale dynamic web services," ACM Intl. World Wide Web Conf., May 2001.

[23] C. Yuan, Y. Chen, and Z. Zhang, "Evaluation of edge caching/offloading for dynamic content delivery," ACM Intl. World Wide Web Conf., May 2003.

Ho-Joong Kim received his B.S. and M.S. degrees in Computer Science from KAIST, in 1998 and 2000 respectively. He is currently in the Ph.D. course of the Department of Computer Science, KAIST. His research interests include cluster computing, internet servers and large-scale distributed systems.

Seungryoul Maeng received the B.S. degree in Electronics Engineering from Seoul National University, Korea, in 1977, and the M.S. and Ph.D. degrees in Computer Science from KAIST, in 1979 and 1984, respectively. Since 1984 he has been a faculty member of Department of Computer Science of KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include micro architecture, parallel computer architecture, cluster computing, and embedded systems.