A Partial Replication Approach for Anywhere Anytime Mobile Commenting

Huanhuan Xia[1], **Tun Lu**[1], **Bin Shao**[2], **Guo Li**[1], **Xianghua Ding**[1], **Ning Gu**[1]

[1]School of Computer Science, Fudan University, Shanghai, China

[2]Microsoft Research, Beijing, China

{huanhuanxia,lutun}@fudan.edu.cn, binshao@microsoft.com, {guoli,dingx,ninggu}@fudan.edu.cn

ABSTRACT

Commenting systems play increasingly important roles in the interactive web applications. Meanwhile, more and more web applications are visited on mobile devices. However, the intermittent connection of mobile networks and resource limitation of mobile devices pose great challenges, mainly in terms of interactive responsiveness and data consistency. In this paper, we present the first work of partial replication solution based on collaborative editing techniques, which can address the issues of local responsiveness and resource limitation on mobile commenting systems. We report how we address the consistency maintenance challenges that come with the partial replication approach. With this approach, users are allowed to smoothly comment anywhere anytime. The comment thread can be incrementally updated and automatically synchronized with strong data consistency guarantees. We implemented a system prototype called Hydra and evaluated it on a real data set.

Author Keywords

Commenting System; Partial Replication; Consistency Maintenance; Synchronization.

ACM Classification Keywords

H.5.3 Information Systems: Group and Organization Interfaces; C.2.4 Computer Systems Organization: Distributed Systems—*Client/Server*.

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

The last few years have witnessed that the Web has been evolving into an interactive and collaborative platform for a great variety of web media, such as news, blogs, goods, music, videos, tweets. Commenting systems can greatly improve user interaction and collaboration on the web media, by which users express their individual opinions or sentiments, and even contribute their insights, knowledge and creativity. There are many popular commercial commenting systems such as Disqus[1], IntenseDebate[2] and Livefyer[3], which are widely adopted by web sites to encourage user interaction and participation. Taking Disqus for example, it has over 70 million users; over 1.4 million sites are using its commenting service; and about hundreds of thousands of comments are generated everyday.

With the rapid development and penetration of mobile technologies, commenting activities become more pervasive. Marketing-Charts' mobile research found that 34% Facebook users and 43% Twitter users access their accounts via mobile phones [2]. For Sina Weibo, the biggest microblogging system in China, 60% active users are mobile [1]. And the growth still continues. The mobile users of Twitter increased 182% in 2011 [3].

However, we are still in the era when the internet mainstay is not optimized for mobile devices. Specifically, most existing commenting systems do not pay enough attention to mobile users. Even some commenting systems have already taken the screen size into consideration, many of them overlooked another important difference between desktop PCs and mobile devices, namely, network connection. Most mobile users can surf the web, but not in a comfortable way due to intermittent connection and high latency issues of mobile networks.

The network connection issues make smooth commenting without disruption a big challenge in the mobile environment. Users need to wait for the server response after clicking the submit button. It usually gives users a long pause in the high-latency mobile network. Even worse, sometimes nothing but a failure notification is returned after the long pause when the network connection is temporarily unavailable . And this really leads to bad user experience.

A widely used approach to addressing this problem in disconnected environments is *optimistic replication* [18]. With local data replica, operations can be performed on the local replica immediately in a non-blocking way. It is especially promising to solve the responsiveness problems in mobile environment by allowing offline operations when network is disconnected. However, replication can lead to data inconsistency. How to make an optimal trade-off between high responsiveness and

CSCW'14, February 15–19, 2014, Baltimore, Maryland, USA.

http://dx.doi.org/10.1145/2531602.2531609

[1]http://disqus.com
[2]http://intensedebate.com/
[3]http://www.livefyre.com/

data consistency is a key technical challenge of interactive applications, such as online gaming [26], co-authoring [6, 23] and collaborative software development [10].

Collaborative editing techniques, such as Operational Transformation (OT) [6, 23] and Address Space Transformation (AST) [7], are representative optimistic concurrency control methods to address this challenge. To achieve high responsiveness, they adopt the *full replication architecture*. Local operations can be applied immediately on the local replica to obtain unconstrained local user experience. To maintain the data consistency, before applying the remote operations on local data replica, the remote operations (as in OT) or local data replica (as in AST) are transformed to correctly incorporate the effects of remote operations.

However, it suffers from resource constraints, and is usually not necessary, to make full replicas for a commenting system on mobile devices. Due to the spotty nature of comments and the diversity of user interests, a user may slide many screens before encountering worth-reading items, which makes full replication useless in most cases. *Partial replication* of the online content can not only save the network bandwidth and battery life of the device, but also potentially make better user experience of web surfing. Besides consistency maintenance issue, partial data replication brings new challenges to OT and AST.

In this paper, we contribute a novel partial replication approach to anywhere anytime mobile commenting experience. It can be used by existing commenting systems to fetch and construct a comment thread incrementally for mobile users, achieve high local responsiveness in intermittent mobile networks, and synchronize the states of comment threads automatically with strong data consistency guarantees. It is important to note that local responsiveness is different from real time synchronization: the former can be achieved through data replication, whereas the latter relies on network conditions.

The contribution of this work are threefold. First, to the best of our knowledge, it is the first work of applying collaborative editing techniques in partially replicated architecture for mobile commenting. Second, we addressed data consistency maintenance issues that come with partial replication approach by devising a partial replication scheme and a synchronization protocol to synchronize the states of comment threads. Third, a scalar timestamp based synchronization protocol is designed for achieving efficient and scalable commenting systems.

The rest of the paper is organized as follows. Section 2 lays out the approach overview and points out the technical challenges that have not been fully addressed by prior work. The partial replication of comment threads and the corresponding replication synchronization protocol are elaborated in Section 3 and 4, respectively. Section 5 introduces the underlying consistency maintenance technique in our approach. Our approach is evaluated in Section 6. Related work is discussed in Section 7. Section 8 concludes the paper by summarizing our contributions and discussing possible future research direction.

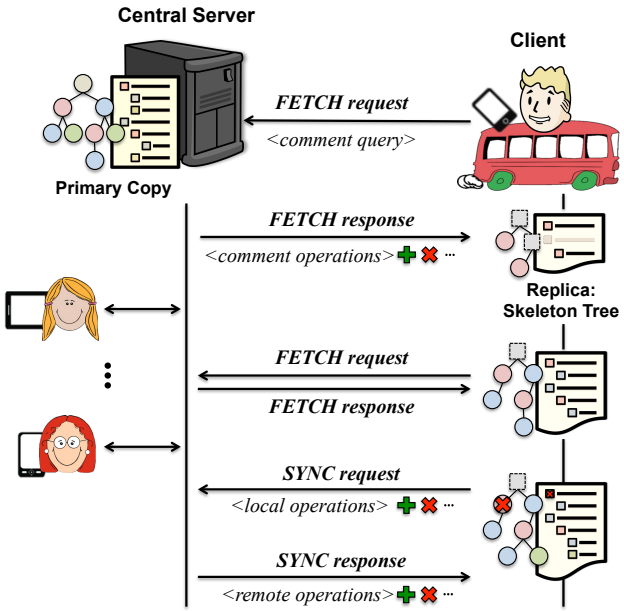

Figure 1. Approach sketch.

2. OVERVIEW

2.1 Approach Sketch

A commenting system consists of a dedicated server and a number of clients as shown in Figure 1. For each comment thread, the primary copy is maintained by the central server. Users can fetch and replicate any comments they want in the thread, and local replicas are built incrementally on their client devices.

When a user starts to participate in an online discussion, a *FETCH* request will be sent to the server specifying what comments are requested by the user, for example, the latest comments. The requested comments will be returned to the client in the form of commenting operations, and are maintained in a special data structure of the replica, which is called the *skeleton tree* in our approach. The skeleton tree is responsible for organizing the replicated comments and maintaining their relationships, i.e. the parent-child relationship and sibling relationship. To avoid transferring unnecessary data and thus save the bandwidth of mobile users, the central server tracks the states of each replica and responds to client's subsequent *FETCH* requests incrementally.

Users are allowed to freely perform commenting operations on their local replicas in a non-blocking way, even in disconnected situations, e.g. posing, editing or deleting comments, voting for other users' comments etc.. Local operations are executed immediately by the clients. Offline local operations will be submitted to the server in a *SYNC* request to synchronize the local replica with the primary copy when the network is available. The primary copy on the central server will get updated by integrating client's local operations, and the operations from other users will be identified and returned to the client for updating the local replica.

In replica synchronization, concurrent operations may cause conflicts and lead to inconsistent comment thread state. In our

approach, AST [7] (Address Space Transformation), which is originally proposed for group editing, is used to maintain the data consistency. Before executing a remote commenting operation, the concurrent operations are identified to transform the local replica or the primary copy to the state in which the operation was generated.

2.2 Technical Challenges

Existing collaborative editing techniques require that the shared document are fully replicated on all collaborative sites. To use them for consistency maintenance in partial replication and synchronization of comment threads, the following technical challenges must be addressed:

- First, *how to partially replicate a comment thread and incrementally construct the replica*. In collaborative editing, the shared document is fully replicated on all collaborative sites. However, in commenting systems, users may want any part of a comment thread. Moreover, during a comment conversation, users may fetch their required comments via several *FETCH* requests. Therefore, a replication scheme must be designed for incrementally constructing the replica to save the bandwidth of mobile users.
- Second, *how to identify concurrent operations and correctly transform the comment thread state*. In collaborative editing, fully shared document and the complete operation history can be used to easily identify concurrent operations. In AST, the document transformation is realized by traversing the whole document. So when the comment thread is partially replicated, the AST must be adapted and the replication scheme should ensure adequate information can be provided for AST to correctly perform the transformation.
- Third, *how to meet the scalability requirement of commenting systems and support user's dynamic participation*. Unlike collaborative editing, a hot topic may attract thousands of users in a short time, and users often join and leave a conversation highly dynamically. Most existing collaborative editing algorithms use vector timestamps to determine the concurrent relation between operations. The size of the vector timestamp is proportional to the number of the participants and will consume considerable mobile bandwidth under high degree of concurrency. Therefore, in our approach, an efficient concurrent operation detection method and a scalable synchronization protocol based on the scalar timestamp are devised for large-scale commenting systems.

3. PARTIAL REPLICATION OF COMMENT THREADS

Almost all existing commenting systems fetch a comment thread incrementally, for example, loading a comment thread page by page, expanding the nested comments only if a user wants to read more. In addition to these common schemes, a commenting system can even track and learn user commenting behaviors, and provide personalized commenting experience. Our approach allow clients to incrementally fetch comments with different schemes (e.g. "next 20 comments", "all comments posted by Alice" etc.). However, how to define a comment query and perform a query are application specific, and it is not the focus of this paper.

In this section, we first introduce a tree structure for modeling comments threads. Then we explain how a tree-structure comment thread is partially replicated on clients as well as the process of incrementally constructing a partial replica.

3.1 Comment Tree Structure and Primitive Operations

User interactions in a comment thread naturally form a tree structure. Although it is not necessary for all commenting systems to store comment threads as tree structures, it would be very convenient to hide the differences of various physical storage models (e.g. database, xml, disk file etc.) using a generic data structure. In our approach, a comment thread is modeled as a generic tree structure of a set of comments. A comment is uniquely identified by a global identifier and has a set of attributes, e.g. *author*, *content*, *likes* (i.e. the number of users who like the comment) etc.. Replies to the same parent comment node constitute an ordered list of child comment nodes. Therefore, a tree structure for modeling a comment thread is defined as a comment node set $\{N|N = \langle id, parent, children, attributes\rangle\}$, where:

- $N.id$ denotes the global unique identifier of N;
- $N.parent$ denotes the parent comment node of N;
- $N.children$ denotes the child comment node list of N;
- $N.attributes$ denotes the attributes of N, which is a set of name-value pairs.

Various comment operations are supported by different commenting systems, and typical comment operations include *Post* (post or reply a comment), *Edit* (edit the comment content), *Delete* (delete an existing comment), *Like* (like a comment by another user) etc.. For general purpose, four primitive operations on the tree structure are defined as follows:

- *Append (parentId, id, attributes)*: Create a new comment node with id and $attributes$, and add it to the end of the child node list of the parent node indicated by $parentId$.
- *InsertBefore (parentId, refId, id, attributes)*: Create a new comment node with id and $attributes$, and insert it before the existing child node indicated by $refId$ in the child node list of the parent node indicated by $parentId$.
- *Delete (id)*: Delete the comment node indicated by id.
- *Update (id, name, value)*: Update the attribute indicated by $name$ of the comment indicated by id with the $value$.

The structural operations *Append*, *InsertBefore* and *Delete* can support adding or deleting a node at any position in a tree structure. The *Update* operation can be used to change the content of a comment. Note that the comment content represented as a set of *attributes* is application-specific. In a commenting system, in general, a comment has two types of attributes: *string-valued attribute* (e.g. author name, content) and *numerical attribute* (e.g. the number of likes). The string-valued attributes are commonly not allowed to be modified (e.g. the author's name) or only allowed to be modified by the comment's author (e.g. the comment's content). The numerical attributes are allowed to be updated by any user in an incremental manner, e.g. the value of *'likes'* of the comment *'A'* will be increased by 1 after the operation *Update(A, 'likes', 1)* is executed. Each comment node is associated with

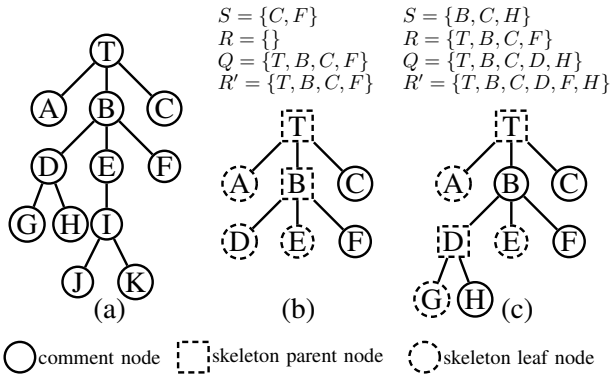

Figure 2. An example comment tree structure and the incremental update of the partial replica. (a) The example comment tree structure. (b) The replica after comments *C* and *F* are fetched for the user. (c) The updated replica after comments *B*, *C* and *H* are fetched.

a history buffer which logs the operations that have been executed on this node.

3.2 Primary Copy and Partial Replicas

Figure 2 (a) shows the tree structure of an example comment thread. It is stored as a *primary copy* at the server side. Users may want only a specific set of comments when they participate in the online discussion, and the required comments, for example the comments *C* and *F*, may be distributed within the tree structure arbitrarily. It does not make sense for a comment thread to only replicate and store the required nodes on the client independently. The relationships between the replicated nodes must be preserved and consistent with those in the primary copy, including the *parent-child* and *sibling* relationships.

To preserve the relationships between the replicated nodes, the rule "*If a comment node is replicated, its parent node (if any) must also be replicated*" is applied when replicating a subset of a comment tree structure. In this way, it is easy to show that all replicated nodes can be organized in a single rooted tree structure on the client, and the parent-child relationships are naturally preserved. As shown in Figure 2 (b), when comment nodes *C* and *F* are required to be replicated, their parent nodes *B* and *T* (which is also *B*'s parent node) are replicated on the client as well. The tree structure of the partial replica is called *skeleton tree* in our approach, and the parent nodes replicated to form the skeleton tree, e.g. *B* and *T*, are called *skeleton parent nodes*.

When a node is replicated, the history operations logged in its history buffer will be transferred to the client and executed. As shown in Figure 2 (b), the *skeleton leaf nodes A*, *D* and *E* are also generated in the skeleton tree after executing history operations logged in *T* and *B*, respectively. The skeleton leaf nodes are not explicitly required to be replicated on the client, however, they are important for maintaining consistent sibling relationships across replicas. The underlying consistency maintenance technique will be elaborated in Section 5. It should be pointed out that, to save the bandwidth of mobile users, all skeleton nodes, including the skeleton parent and skeleton leaf nodes, are maintained in the skeleton tree without content data (i.e. the nodes' attributes).

3.3 Incremental Construction of Partial Replicas

When a user starts to participate in an online discussion, the first *FETCH* request will be sent to the server to fetch the specific comments wanted by the user. The initial skeleton tree can be constructed from top to bottom by executing the operations returned to the client. When subsequent *FETCH* requests are sent to refresh the current replica or fetch additional comments for the user, some of the required nodes may exist in the client replica. To save the bandwidth of mobile users, the client replica should be updated incrementally. In other words, only incremental operations should be returned by the server to update the skeleton tree of the partial replica.

To determine which nodes should be replicated and how to respond to a *FETCH* request in an incremental manner, the server maintains a *replica metadata* for each replica to track the replica's state. It records which nodes have been replicated and their *replication types* in the skeleton tree. *CN* and *SN* denote the replicated comment node and skeleton parent node, respectively.

Algorithm 1 $Fetch(R, Request) : Response$

1: $Response \leftarrow Sync(R, Request)$
2: $S \leftarrow$ *comments specified by* $Request.query$
3: $Q \leftarrow$ *required nodes for replicating nodes in* S
4: **for** each $N \in Q$ **do**
5: *Determine what data should be returned to the client for* N *according to* N*'s replication type, and update* R *and* $Response$.
6: **end for**
7: **return** $Response$

Given the replica metadata *R*, Algorithm 1 specifies the procedure of responding to a *FETCH* request which consists of a comment query and unsubmitted client operations before the request. The server first calls Algorithm *Sync*, which is elaborated in next section, to execute the client's local operations within the request (line 1), and then performs the query on the latest primary copy to get the comment set *S* wanted by the client (line 2). For example, $S = \{B, C, H\}$ in Figure 2 (c) indicates that the comments B, C, and H are required by the client. According to the replication rule described above, the nodes that should be replicated are identified and stored in $Q = \{T, B, C, D, H\}$ (line 3). The set Q can be calculated by including all nodes on the paths from the root to the nodes in S (e.g. $Q = \{T, B\} \cup \{T, C\} \cup \{T, B, D, H\}$).

Because some of nodes in *Q* may have existed in the client's replica, the server determines what data should be returned for each node $N \in Q$ according to their replication types as follows (line 4-6):

- Case 1: $N \notin R \wedge N \notin S$. N will be replicated as a new skeleton parent node, e.g. *D* in Figure 2 (c), so only the history operations on N will be returned to construct its child node list. *Update* operations and *attributes* fields of *Append* and *InsertBefore* operations will be excluded. N's replication type is set as *SN*.

- Case 2: $N \notin R \wedge N \in S$. N will be replicated as a new comment node, e.g. *H*. Similarly with Case 1, history operations on N will be returned. In addition, N's attributes will returned as well. N's replication type is set as *CN*.
- Case 3: $N \in R \wedge N \notin S$. N has existed in the replica and is not wanted by the user, e.g. *T*, so nothing needs to be done.
- Case 4: $N \in R \wedge N \in S$. If N is currently replicated as skeleton parent node, e.g. *B*, N's attributes will be returned to the client and its replication type will be changed from *SN* to *CN*, otherwise nothing needs to be done for N, e.g. *C*.

Note that Algorithm *Sync* is called at line 1 to update the primary copy and identify incremental remote operations for the replicated nodes. So actually, for $N \in R$ in Case 3 and 4, incremental operations on N (if any) will also be returned to the client. This can be considered as the synchronization of the two corresponding nodes between the client and the server, and it will be elaborated in next section.

When the client receives the response returned from the server, it first executes the incremental operations within the response to update the structure of the skeleton tree, and then sets the *attributes* fields of the nodes whose contents are also returned along with the operations.

4. PARTIAL REPLICA SYNCHRONIZATION

In this section, we introduce how replicated nodes are synchronized with their corresponding nodes on the server. A batch of local operations, which are executed since the last synchronization with the server, are incrementally submitted using a single *SYNC* message. Then the server applies all received operations on the primary copy. After that, remote operations which are concurrent with the submitted operations are identified and returned to update current client's local replica.

4.1 Concurrent Operation Detection

In a typical commenting system, a server may serve thousands of online users at the same time, and users may join and leave a discussion dynamically. Therefore, the concurrent operation detection method must be efficient enough. In this section, a concurrent operation detection method with $\log(n)$ time complexity is presented, where n is the length of a node's operation history buffer.

On the server side, the received operation sequences will be stored in the sequence history buffers (*SHB*) of corresponding nodes in the order of execution time. On the client side, an operation sequence that has been synchronized with the server is stored in the *SHB* of the corresponding node, and operations that have not been submitted to server are stored in the local history buffer (*LHB*). Each sequence S_i in the history buffer is a tuple $\langle R, T, T', L \rangle$, where:

- $S_i.R$ denotes the replica on which S_i was generated;
- $S_i.L$ denotes the list of operations constituting S_i;
- $S_i.T$ denotes the finish time of executing S_i on server;
- $S_i.T'$ denotes the finish time of $S_i.R$'s last synchronization with the server.

Here $S_i.T$ and $S_i.T'$ are global scalar timestamps, which can be realized by real time or logic time on the server. Some existing centralized collaborative systems adopt two-dimensional timestamp, one dimension is for the client, the other is for the server. For example, Jupiter [13] uses two-dimensional timestamps to transform concurrent operations in the transformation state space. However, Jupiter assumes the shared object is fully replicated. If the shared object is partially replicated on individual clients, two-dimensional timestamps generated in local partial states are not comparable globally.

For any sequence S in the *SHB* of a comment node N, S is generated after $S.T'$. When S is generated on the replica $S.R$, all sequences before $S.T'$ must have been executed on $S.R$, and no sequence after $S.T'$ is returned and executed on $S.R$. Sequences generated on the same local replica are submitted and executed on the primary copy in their generating orders. Therefore, given two sequences S_i and S_j in the *SHB* of a node N, where $0 \leq i \leq j < |N.SHB|$, one of the following relations is hold between S_i and S_j:

- *casually-before*: $S_i \rightarrow S_j$. S_i happens before S_j if $S_i.T < S_j.T'$ or $S_i.R = S_j.R$;
- *concurrent*: $S_i \shortparallel S_j$. S_i is concurrent with S_j if $S_i.T > S_j.T'$ and $S_i.R \neq S_j.R$.

Since operations in a sequence are ordered by their generating times, the relations between two operations can also be specified as following: Given two operations O_1 and O_2, where $O_1 \in S_i.L$ and $O_2 \in S_j.L$,

- $O_1 \rightarrow O_2$: O_1 happens before O_2 if $S_i \rightarrow S_j$ or $S_i = S_j$ and O_1 is before O_2 in $S_i.L$.
- $O_1 \shortparallel O_2$: O_1 is concurrent with O_2 if $S_i \shortparallel S_j$.

Given an operation sequence, concurrent operation sequences are those whose execution times T are greater than $S.T'$ except for those from $S.R$, where T' is the time of the last synchronization time of an operation sequence S. Because operation sequences are ordered by their execution times T in the *SHB* of a node, a binary search can be performed on the *SHB* to identify all concurrent sequences that have not been executed on the replica.

4.2 Partial Replica Synchronization

A synchronization process can be started manually, triggered periodically, or triggered automatically when the mobile device is reconnected. Figure 3 illustrates two synchronization processes between a client and the server.

When the local operation sequence S_j is ready to be submitted to the server at the state (1), i sequences (i.e. $S_{0..i-1}$ in $N.SHB$) have been synchronized with the server. The sequence S_j was generated after the client's last synchronization with the server, and the finish time T_1 of the last synchronization will also be submitted to the server with S_j to identify concurrent remote operation sequences.

After the *SYNC* request is received from the client at the state (2), j sequences (i.e. $S_{0..j-1}$ in $N.SHB$ shown in Figure 3) have been received and executed on the primary copy. The server first identifies the concurrent sequences after $S_j.T' =$

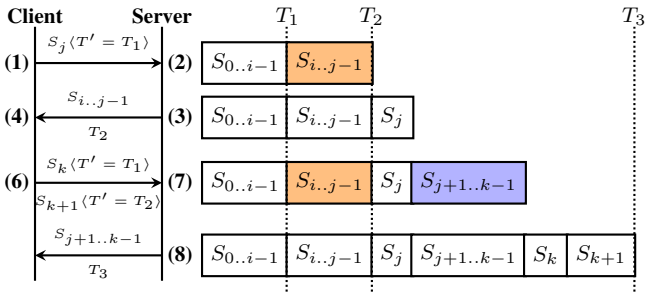

Figure 3. Illustration of two synchronization processes between a client and the server.

State	SHB	LHB
(1)	$S_{0..i-1}$	$S_j\langle T' = T_1\rangle$
(4)	$S_{0..i-1}$	$S_j\langle T' = T_1\rangle$, $S_k\langle T' = ?\rangle$
(5)	$S_{0..j}$	$S_k\langle T' = T_1\rangle$
(6)	$S_{0..j}$	$S_k\langle T' = T_1\rangle$, $S_{k+1}\langle T' = T_2\rangle$

Table 1. The *SHB* and *LHB* on the client shown in Figure 3.

T_1 for S_j, i.e. sequences $S_{i..j-1}$. After executing operations in S_j and adding it into N.*SHB* at time T_2 (shown as state(3)), the server returns the concurrent sequences $S_{i..j-1}$ and T_2 to the client.

During the synchronization, i.e. before responded by the server, local operations on the client replica are still allowed to be executed for high local responsiveness. For example at the state (4), besides the sequence S_j that is submitted to the server, the local operations generated during the synchronization are stored in the sequence S_k in N.*LHB*. After receiving the *SYNC* response, the client first executes remote operation sequences on N; then moves S_j into N.*SHB* and sets $S_k.T'$ for the next synchronization (shown as state (5)). It is worth to note that, $S_k.T'$ is set to $S_j.T' = T_1$ rather than the finish time T_2 of the current synchronization. This is because the operations in S_k are executed on the replica which have not integrated the operations in $S_{i..j-1}$, therefore, when S_k is submitted to the server in the next synchronization, $S_{i..j-1}$ must be identified as concurrent remote sequences for correctly executing S_k on the primary copy.

For the operations executed locally after the synchronization, i.e. S_{k+1} in N.*LHB* at state (6), they are generated after $S_{i..j-1}$ and S_j have been executed on the replica, so when it is submitted to the server, T_2 will be used to identify its concurrent remote sequences $S_{j+1..K-1}$ at the state (7) on the server. It is easy to show that, in a *SYNC* request, there are at most two local sequences for a node to be submitted to the server. After executed on the server, S_k and S_{k+1} submitted by the client are sequentially added into N.*SHB* (as shown in state (8)). The finish time T_3 of the synchronization and the concurrent sequences $S_{j+1..K-1}$ after T_2 are returned to the client.

Given the replica metadata R of the client from which the *SYNC* request is received, Algorithm 2 specifies the procedure of responding to a synchronization request. The server first executes the submitted operations for each node N in the request (lines 1-9) to update the primary copy. For each operation sequence S of N, the concurrent operations in $N.SHB$ are identified (line 3), and then Algorithm *Execute* is called to execute the operations in S (line 5). An underlying consistency maintenance technique, which will be presented in next section, is used in *Execute* to maintain the consistent sibling relationships across replicas and resolve operation conflicts automatically. S is appended into the node's history buffer after the operations in it are executed (line 6). Note Algorithm *Execute* returns a list of comment nodes which are created on the client. These nodes should be added into the replica metadata R as the replication type *CN* (line 7). After all submitted operations are executed on the primary copy, concurrent operation sequences are identified for each nodes in the replica and added into the response (lines 11-14). However, for nodes that are replicated on the client as skeleton parent node, *Update* operations on them are excluded. Finally, the finish time of this synchronization process is set for the replica metadata and the response (line 15).

Algorithm 2 $Sync(R, Request) : Response$

```
1: for each N ∈ Request do
2:   for each operations sequence S of N do
3:     CSL ← {S'|S'∈N.SHB ∧ S' ∥ S}
4:     S.T ← now
5:     NL ← Execute(N, S, CSL)
6:     Append S into N.SHB.
7:     Add nodes in NL into R as CN.
8:   end for
9: end for
10: Response ← [ ]
11: for each N ∈ R do
12:   CSL ← {S'|S'∈N.SHB ∧ S'.T>R.T' ∧ S.R≠R}
13:   Add CSL into Response for N. Update operations are
      excluded if N is replicated as a skeleton parent node.
14: end for
15: Response.T' ← R.T' ← now
16: return Response
```

When the client receives the response returned from the server, it first executes the remote operations within the response to update the replica. Similarly with the operation execution on the server side, Algorithm *Execute* will be called for consistency maintenance and conflict resolution. However, it should be noted that, besides concurrent operations in a node's *SHB*, all operations in the node's *LHB* (if any) are also concurrent with the returned remote operations targeting the same node. After the remote operation are executed, the last synchronization time will be set for operation sequences in *LHB* , and the submitted sequences will be moved from *LHB* to *SHB*.

5. CONSISTENCY MAINTENANCE

For high local responsiveness, the shared comment thread is replicated on clients and operations are allowed to be concurrently performed on any part of the local replicas. When a local operation is submitted to the server or a remote operation is returned to the client, the state of the tree structure may be different from the state in which the operation was

Algorithm 3 Execute(N, S, CSL):L

```
1: for each o ∈ CSL do
2:     Exclude o's effect on its affected node.
3: end for
4: L ← [ ]
5: for each o ∈ S do
6:     Execute o on the node N.
7:     Put o's affected node into L if o is an Append or an
       InsertBefore operation.
8: end for
9: for each o ∈ CSL do
10:    Restore o's effect on its affected node.
11: end for
12: return L
```

generated. Naively applying concurrent operations will lead to inconsistent data replica even corrupt data replicas.

In this section, an AST (Address Space Transformation) based consistency maintenance technique designed for partial replication architecture is elaborated. Note that we only focus on maintaining the syntactic not semantic consistency (e.g. a user posts a same comment twice in a short time or deliberately posts the answer before the question). The semantic inconsistency issues usually cannot be resolved solely by technical approaches without human interventions.

5.1 AST Basis

AST is originally proposed in [7] for collaborative editing of linear character documents. It ensures that the order of linear character nodes is consistent across all replicas after the same group of operations are executed at all collaborative sites. One of the most important advantages of AST is that a remote operation can be directly executed in its original form, as long as the address space on which this operation was generated exists and all concurrent operations' effects are excluded.

To ensure a remote operation can be correctly executed, AST transforms the address space (i.e. the comment tree structure) to the state at which the operation was generated. This is realized by excluding the execution effects of all concurrent operations. The effects of concurrent operations will be restored after the remote operation is executed. In order to achieve the address space transformation, AST logs all history operations and marks deleted nodes as *ineffective* rather than really removing them from the address space.

In AST [7, 25], history operations are associated with their target nodes. The address space transformation is achieved by the *Retracing* procedure [7], which visits all the nodes and determines their *effective/ineffective* marks to exclude or restore the effects of concurrent operations. However, it is not applicable for partial replication of the tree structure, since a node may not exist in a partial replica.

5.2 Adapting AST for Partial Replication

We have two observations on address space transformation. One is that a concurrent operation does not need to be considered in an address space transformation, if it has no effect on the execution of a remote operation, i.e. there is no conflict between the two operations. Another one is that the essence of address space transformation is to exclude or restore the execution effects of conflict concurrent operations, and it is unnecessary to traverse all the nodes as long as all conflict concurrent operations can be identified. By reviewing the tree structure and its primitive operations, we note that the conflicts of concurrent operations are confined to a node and its children, since the nodes are addressed by global identifiers. Specifically, for concurrent *Append*, *InsertBefore* and *Delete* operations, conflicts can be caused only when their target nodes share the same parent node. The *Update* operations only affect the content of comment nodes and have no effects on the structure of the comment thread, so they do not conflict with the other types of operations.

Therefore, to adapt AST for partial replication of the tree structure, history operations logged in their affected nodes are moved to the history buffer of the parent nodes. Each node maintains an *effectiveness counter*. When a node is created by an *Append* or *InsertBefore* operation, its effectiveness counter is initialized to 1. If a *Delete* operation is executed on the node, its effectiveness counter is decreased by 1. A node is *effective* if and only if its effectiveness counter equals 1. To execute a remote operation, after the concurrent operations are identified, the address space is transformed by directly manipulating the effectiveness counters of the concurrent operations' affected nodes. The partial replication mechanism (in Section 3) and the synchronization protocol (in Section 4) ensure all conflict concurrent operations that may lead to conflicts can be efficiently identified for executing a remote operation.

Given the sequence list of concurrent operations *CSL*, Algorithm 3 specifies the procedure of executing a remote operation sequence S on the node N. Since the operations in S are sequentially generated on their corresponding local replicas, they can be correctly executed in order (lines 5-8) after the effects of concurrent operations are excluded (lines 1-3). Concurrent operations' effects are restored (lines 9-11) after the remote operations are executed. The algorithm *Execute* returns a list of nodes which are created by *Append* and *InsertBefore* operations.

The *Delete* operation can be executed directly by decreasing the *effectiveness counter* of its affected node by 1. The *Append* and the *InsertBefore* operations can be executed by inserting a new node between two reference nodes. Due to concurrent insert operations and preceding delete operations, there may exist ineffective nodes between the two reference nodes. The procedure *RangeScan* [7] will be called to insert the new node at the correct position in the range of ineffective nodes and ensure a consistent sibling node order across all replicas.

Figure 4 illustrates the operation execution and conflict resolution. The node N as well as its children A and B are replicated on both client 1 and 2. N's initial states (Figure 4(b)) on the server, client 1 and client 2 are identical to each other. The server receives O_1 and O_2 from client 1, followed by O_3 from client 2 (Figure 4(a)). O_1 inserts X before B, and then

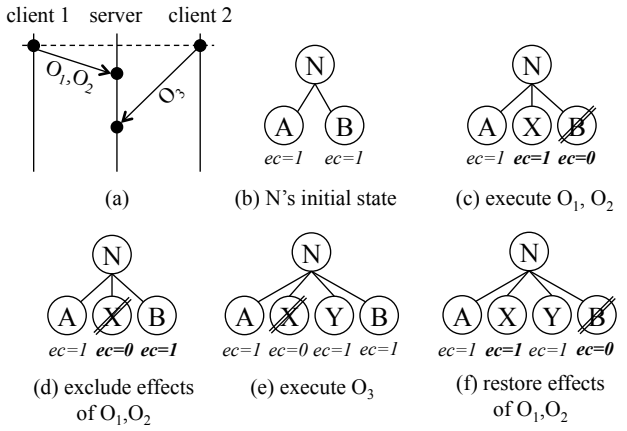

Figure 4. Examples of operation execution and conflict resolution. O_1=*InsertBefore(N,B,X)*, O_2=*Delete(B)* and O_3=*InsertBefore(N,B,Y)*. *ec* is short for *effectiveness counter.*

O_2 deletes B by decreasing B's effectiveness counter by 1 (Figure 4(c)).

O_3's execution is slightly different, because it is concurrent and conflict[4] with O_1 and O_2. Before executing O_3, effects of O_1 and O_2 must be excluded by decreasing X's effectiveness counter by 1 and increasing B's effectiveness counter by 1, respectively (Figure 4(d)). Y, which is intended to be inserted before B (i.e. between A and B), is inserted after the ineffective node X (Figure 4(e)) by calling the procedure *RangeScan*. O_1 and O_2's effects are restored after O_3's execution (Figure 4(f)). After concurrent operations O_1 and O_2 are returned to and executed on the client 2, N's state on the client 2 will be consistent with the one shown in Figure 4(f).

For two *Update* operations targeting the same *attribute* of a node, if the attribute is numerical, they do not conflict with each other since the target attribute is incrementally updated; otherwise, due to the attribute is only allowed to be updated by the comment's author, the conflict can be automatically resolved in a *"Last-Write-Win"* manner. Specifically, the execution of an *Update* operation is different between the server and the client. On the server side, an *Update* operation can be directly executed; while on the client side, a remote *Update* operation will be discarded if it targets a string-valued attribute which has been updated by an unsubmitted local operation.

Operations can be executed in linear time. Since an operation's effect can be excluded or restored by manipulating the affected node's *effectiveness counter* in constant time, the address space transformation can be done in $O(k)$ time, where k is the number of concurrent operations in *CSL*. Primitive operations can also be executed in constant time. So, *Execute*'s time complexity is $O(k + |S|)$.

6. EVALUATION

We have implemented a system prototype called Hydra and have evaluated it using the real comment data crawled from Disqus, which is a popular 3rd party commenting system used by CNN, FoxNews, Time, and Engadget etc. First, we analyze the characteristics of the Disqus comment data. Then, we present the experimental evaluation results.

6.1 Analysis of Disqus Comment Threads

We collected 32,289 comment threads with 8,612,658 comments from Disqus. Disqus lists the latest articles from the web content providers in its Explore Page [5]. We use those article URLs to crawl the articles and their comment threads. These comment threads can be constructed as tree structures in 3.3 seconds, 0.1 milliseconds per thread on average. Figure 5 shows the distribution of the comment thread size. About 25% of the threads consists of less than 10 comments, and 10% of the threads consists of more than 500 comments. On average, a comment thread consists of 267 comments contributed by 115 users. The largest thread consists of 30,232 comments, and the most popular comment conversation attracts 28,652 users. This suggests Hydra has to scale to a large comment thread size and a large number of participants.

In the crawled dataset, each user posted 2.32 comments on average. And the most active user posted 850 comments in a single comment thread. We examined the user interactions in the collected comment threads, and considered a reply to another user as an interaction between the two users. About 49% of users read and replied to other users' comments. Only 40% of comments were directly posted to the original web content, while 60% were replied to others' comments. These observations suggest user interactions accounted for a large proportion of comment threads.

6.2 Evaluation of Hydra

We implemented a system prototype called Hydra and a baseline commenting system to evaluate our approach.

- Client side response time was measured to evaluate the local responsiveness in real mobile environments.
- Network bandwidth usage was measured to evaluate the efficiency of partial replication approach.
- The server side response time and throughput were measured to evaluate the system scalability.

Experiment setup

Both the baseline system and Hydra were implemented as web-based commenting systems. For Java servlets were implemented in the baseline server to process comment fetching, posting, editing and deleting requests from clients. The partial replication algorithms were implemented in Hydra serve to process *FETCH* and *SYNC* requests from clients. They were deployed in the same Apache Tomcat server running on a virtual machine with four 2.8 GHz X5600 CPUs, 16G main memory. The operating system is Windows Server 2008 R2.

The evaluation experiments were conducted on the real data set crawled from Disqus. Users commenting behaviors in the

[4] B has been concurrently deleted by O_1, and a new node X has been concurrently inserted between A and B by O_2.

[5] http://disqus.com/explore/

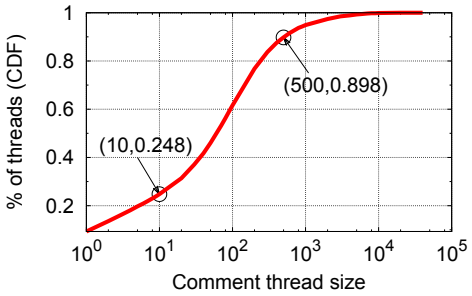

Figure 5. Comment thread size distribution.

Figure 6. Mobile network RTTs between 7:00 a.m. to 9:00 a.m. in Shanghai.

Figure 7. Mobile network RTTs between 12:30 p.m. to 2:30 p.m. in Shanghai.

data were simulated in the chronological order of the comments to generate the client commenting requests. A centralized monitor program was implemented to schedule the commenting clients. Before posting, deleting or editing a comment, the comments depended on by current operation must be fetched first if they did not exist on the client. Since comment deletion and update information cannot be obtained and inferred lexically from the crawled data set, these operations were generated and randomly distributed among comment insertion operations to validate the correctness of operation executions. Users' commenting operations in Hydra were executed immediately on the local replica, while users' operations in the baseline system can only be executed after they were responded by the remote server. The simulation experiment was repeated four times to measure the local responsiveness, network bandwidth usage of mobile clients, and the server side performance.

Local responsiveness

Local responsiveness was measured in real mobile environments. We repeated the experiment twice at different time in a day. They were conducted on school buses between three campuses of Fudan university (i.e. HanDan campus, FengLin campus and ZhangJiang campus shown in Figure 6 and 7). The baseline client and Hydra client were run on a laptop which was connected to China Unicom's 3G mobile network via a laptop connect card. Meanwhile, we also implemented an Android application to record the locations and the round trip time (RTT) of the mobile network.

As expected, the mobile network was intermittent in the experiments. And we observed that the mobile network signal strength had no significant differences at different locations in Shanghai, but changed significantly with the change of time. The network RTTs were marked at different locations on the maps shown in Figure 6 and 7. Figure 8 shows the average local response time under different network conditions for Hydra client and the baseline client. Local response time of Hydra client is consistently close to zero. It means Hydra can provide high local responsiveness for users anywhere anytime. However, local response time of the baseline client increases with larger network RTT, and it can reach tens of seconds in the experiment.

Bandwidth saving

To save the bandwidth of mobile users, the structure of a comment thread is partially replicated on the client; the comment content is replicated if and only if it is explicitly required by a user. The bandwidth usage of each Hydra client can be calculated using the following formula:

$$B_H = b_c \cdot C + b_o \cdot R \tag{1}$$

where C is the number of comments required by a user; R is the number of nodes in the skeleton tree of a replica; b_c is the average comment size in bytes; and b_o is the average commenting operation size in bytes.

Existing collaborative editing techniques (such as OT and AST) can also be used to achieve high local responsiveness if the shared comment thread is fully replicated, we compared Hydra client's bandwidth usage with the following two full replication schemes:

- *Full-Structure-Partial-Content* (F_1): the structure of a comment thread is fully replicated, but the comment content is replicated if and only if it is explicitly required by the user.
- *Full-Structure-All-Content* (F_2): the full structure of a comment thread and its content are both replicated on the client.

Similarly, the bandwidth usage of the clients adopting the full replication schemes F_1 and F_2 can be calculated using the following two formulas respectively:

$$B_{F_1} = b_c \cdot C + b_o \cdot P \tag{2}$$

$$B_{F_2} = b_c \cdot P + b_o \cdot P \tag{3}$$

where P is the number of comments in a comment thread. Therefore, compared with the two full replication schemes, the percentage of bandwidth saved by Hydra are:

$$\delta_{F_1} = \frac{B_{F_1} - B_H}{B_{F_1}} = \frac{P - R}{P + \alpha \cdot C} = \frac{1 - \frac{R}{P}}{1 + \alpha \cdot \frac{C}{P}} \tag{4}$$

$$\delta_{F_2} = \frac{B_{F_2} - B_H}{B_{F_2}} = 1 - \frac{R + \alpha \cdot C}{P + \alpha \cdot P} = 1 - \frac{\frac{R}{P} + \alpha \cdot \frac{C}{P}}{1 + \alpha} \tag{5}$$

where $\alpha = \frac{b_c}{b_o}$ and it equals 4.9 in the prototype implementation. We can see that the percentage of bandwidth saved by Hydra is the function of two ratios: $\frac{C}{P}$ and $\frac{R}{P}$. The ratio $\frac{C}{P}$ denotes the percentage of the comments required by a user, and the ratio $\frac{R}{P}$ denotes to what extend the comment thread is partially replicated.

In the experiment, for comment threads that consist of more than 50 comments, the average percentages of bandwidth

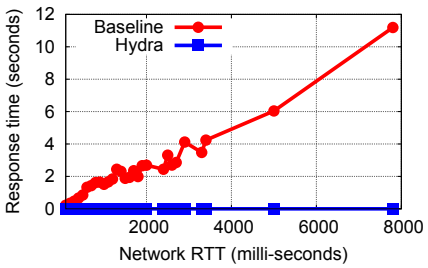

Figure 8. Response times of Hydra and Baseline clients with different network RTTs.

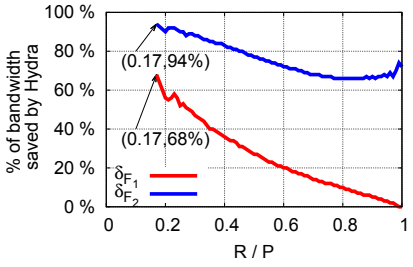

Figure 9. Average percentage of bandwidth saved by Hydra for different replication ratios.

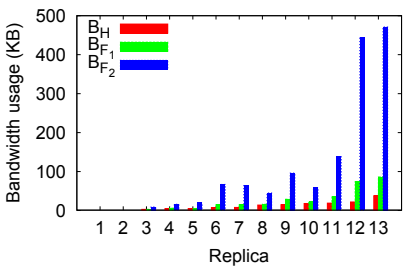

Figure 10. Bandwidth usage comparison of Hydra, F_1 and F_2 clients in an example comment thread.

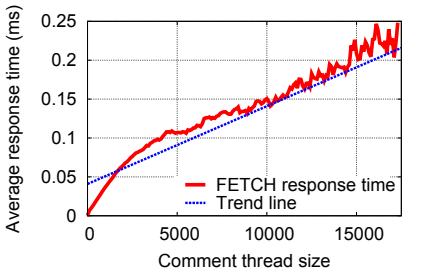

Figure 11. *FETCH* response time to the primary copy size.

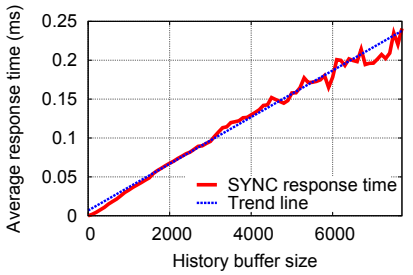

Figure 12. *SYNC* response time to the number of history operations in *SHB* on server.

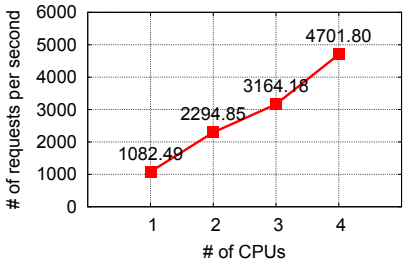

Figure 13. Hydra server throughput when configured different number of single-core CPUs.

saved by Hydra $\bar{\delta}_{F_1}$ and $\bar{\delta}_{F_2}$ were 15.6% and 69.8%, respectively. Figure 9 shows $\bar{\delta}_{F_1}$ and $\bar{\delta}_{F_2}$ for different replication ratios $\frac{R}{P}$. Compared with F_1, the smaller the replication ratio is, the more bandwidth Hydra saves. Compared with F_2, the bandwidth percentage saved by Hydra first decreases and then increases with the increase of the replication ratio. This is because, for large and flat comment threads, the replica size is close to the size of the whole thread even if only a small portion of comments are required by a user. Therefore, in the case where $\frac{R}{P}$ is close 1 and $\frac{C}{P}$ is very small, Hydra can save considerable bandwidth for users compared with F_2.

The minimum of the replication ratios equals 0.17 in the experiment. The corresponding comment thread contained 573 comments and 13 users' replicas. Figure 10 compares the bandwidth usage of Hydra, F_1 and F_2 clients for each user's replica in this comment thread. We can see that the bandwidth usage on the Hydra client is much less than that on F_1 and F_2 clients. For example, the F_1 and F_2 clients consumed 86KB and 471KB of user#13's bandwidth respectively, while the Hydra client only consumed 38KB.

Server side performance

The server side response time and throughput were measured to evaluate the system scalability. Figure 11 and 12 show the execution times of processing *FETCH* and *SYNC* requests, respectively. They are less than 0.25 milliseconds in the experiment, which are negligible compared with the network latency. In addition, the execution times of processing *FETCH* and *SYNC* requests increase linearly with the growth of the comment thread size and the history buffer size, respectively.

Four pressure tests were performed to measure the throughput of Hydra server. The Hydra server was configured with 1, 2, 3, 4 single-core CPUs, respectively. And it was running at 100% CPU usage using excessive commenting requests. Figure 13 shows the throughput under different CPU configurations. The Hydra server throughput is around 1000 requests per second per core on average, and it scales linearly with the number of CPU cores.

7. RELATED WORK AND DISCUSSION

Replication is widely adopted to offer high availability in distributed systems, such as [14, 17, 19]. As an important category of interactive systems, group editors [6, 23] fully replicate the shared document at all clients for high local responsiveness using optimistic concurrency control approaches such as OT (Operation Transformation) [6, 23] and AST (Address Space Transformation) [7]. Over the past two decades, a lot of work has been done for collaborative editing in fully replicated architecture to maintain consistency of replicas and automatically resolve conflicts in P2P architecture [6, 23, 11, 7, 15], client-server architecture [13, 21], mobile environment [20, 22], tree structure of document [5, 8, 25], etc..

However, as analyzed previously, full replication is usually not adopted under a bandwidth-minded environment, such as on mobile devices. To mitigate the problems caused by intermittent connection, online contents are usually prefetched and cached [4, 9] for local responsiveness of content retrieval. The cached content is actually a portion of the whole content, however, they are usually static where no concurrently editing is allowed. Prefetching and cache technologies can be leveraged to decide when and what content to fetch for comment retrieval and our approach can be adopted to maintain

the replicated data and ensure high responsiveness of local operations.

Both OT and AST are potentially promising approaches that can be adapted for partial replication. But to the best of our knowledge, this is the first work proposing a complete solution to applying collaborative editing techniques for partial replication of comment threads. Although AST is used as the underlying consistency maintenance technique in our approach, actually, the partial replication scheme and the synchronization protocol can also be integrated with OT. More specifically, they can ensure right and enough information exchanged between client and server, so that a transformation can be conducted for a remote operation to be executed in a "right" address space (for AST) or in a "right" form (for OT). However, the primitive operations with global identifiers for AST, which enables efficient target nodes addressing, need to be redefined using positional parameters for OT-based partial replication.

Sync [12] is a Java framework for developing mobile collaborative applications, which allows developers to implement the merge matrix to resolve the conflicts in different applications. The main contribution of this paper is a partial replication approach, which has not been addressed in the Sync framework.

Docx2Go [16] is a framework for collaborative editing of XML-based documents on mobile devices, which is the most related work to ours from some technical perspectives. In terms of working on mobile devices anywhere anytime, our system design goal resembles that of Docx2Go. Both Docx2Go and our approach adopt partial replication to accommodate mobile devices. The major differences between our approach and Docx2Go are synchronization mechanisms, conflict resolution and consistency maintenance methods. In Docx2Go, replicas are synchronized by exchanging data of element while operations are exchanged in our system. Due to the difference in manipulated elements and structure, the conflicts in Docx2Go are manually or automatically resolved by merging two conflict versions of an element, but conflicts of tree structure manipulation will be resolved automatically in our system. For consistency maintenance, Docx2Go uses *position indicator* [24, 15] to determine a consistent order of sibling nodes, but its unbounded position indicator and version vector are not suitable for Internet-scale commenting and mobile device with limited resources. Our scalar timestamp based concurrent operation detection and efficient operation integration on server side can ensure higher scalability.

8. CONCLUSION

To address the technical challenges brought by mobile networks, we contribute a novel partial replication approach to address the responsiveness and consistency maintenance issues for anywhere anytime mobile commenting. The main technical contributions are threefold: 1) To the best of our knowledge, it is the first work of applying collaborative editing techniques to achieve high local responsiveness in mobile commenting; 2) A holistic approach for partial replication and synchronization of tree-structure comment threads are proposed; 3) A scalar timestamp based synchronization protocol is designed to meet the scalability requirements of commenting systems and users' dynamic participation.

A prototype system called Hydra was implemented to evaluate our approach. The evaluation results show that our approach is capable of providing high local responsiveness for users in high-latency networks, save bandwidth of mobile users compared with full replication approaches. And it has high efficiency and good scalability for achieving large scale commenting systems.

Existing commenting systems can leverage our approach to achieve high local responsiveness in intermittent mobile networks and do not need to worry about the issues of replication, synchronization and data consistency of comment threads. In the future, we will integrate our approach with existing commenting systems, and conduct user experience studies to better evaluate our approach from the perspective of usability.

ACKNOWLEDGMENTS

We thank Silvia Lindtner for her great help with the paper revision. The work was supported by the National Natural Science Foundation of China (NSFC) under Grants No.61272533, No.61332008, No.61300201, and Shanghai Science & Technology Committee Project under Grant No.11JC1400800 and No.13ZR1401900.

REFERENCES

1. **http://socialmediainasia.blogspot.com/2012/05/more-than-60-of-sina-weibo-active-users.html**.
2. More than talk: Action in mobile marketing. **http://www.hubspot.com/more-than-talk-action-in-mobile-marketing/**.
3. Twitter statistics - updated stats for 2011. **http://www.marketinggum.com/twitter-statistics-2011-updated-stats/**.
4. Billsus, D., Pazzani, M. J., and Chen, J. A learning agent for wireless news access. In *Proceedings of the 5th international conference on Intelligent user interfaces*, IUI '00, ACM (New York, NY, USA, 2000), 33–36.
5. Davis, A. H., Sun, C., and Lu, J. Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, ACM (New York, NY, USA, 2002), 58–67.
6. Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, ACM (New York, NY, USA, 1989), 399–407.
7. Gu, N., Yang, J., and Zhang, Q. Consistency maintenance based on the mark & retrace technique in groupware systems. In *Proceedings of ACM GROUP'05 Conference on Supporting Group Work* (Nov. 2005), 264–273.

8. Ignat, C.-L., and Norrie, M. C. Customizable collaborative editor relying on treeopt algorithm. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, Kluwer Academic Publishers (Norwell, MA, USA, 2003), 315–334.
9. Jiang, Z., and Kleinrock, L. Web prefetching in a mobile environment. *IEEE Personal Communications 5* (1998), 25–34.
10. Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T., and Englund, M. Cored: browser-based collaborative real-time editor for java web applications. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, ACM (New York, NY, USA, 2012), 1307–1316.
11. Li, D., and Li, R. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work 19*, 1 (2010), 1–43.
12. Munson, J. P., and Dewan, P. Sync: a java framework for mobile collaborative applications. *Computer 30*, 6 (1997), 59–66.
13. Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology* (Pittsburgh, Pennsylvania, USA, 1995), 111–120.
14. Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), 288–301.
15. Preguiça, N., Marquès, J. M., Shapiro, M., and Leia, M. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, IEEE Computer Society (Montreal, Québec, Canada, 2009), 395–403.
16. Puttaswamy, K. P., Marshall, C. C., Ramasubramanian, V., Stuedi, P., Terry, D. B., and Wobber, T. Docx2go: collaborative editing of fidelity reduced documents on mobile devices. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, ACM (New York, NY, USA, 2010), 345–356.
17. Ramasubramanian, V., Rodeheffer, T. L., Terry, D. B., Walraed-Sullivan, M., Wobber, T., Marshall, C. C., and Vahdat, A. Cimbiosys: a platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, USENIX Association (Berkeley, CA, USA, 2009), 261–276.
18. Saito, Y., and Shapiro, M. Optimistic replication. *ACM Computing Survey 37*, 1 (2005), 42–81.
19. Salmon, B., Schlosser, S. W., Cranor, L. F., and Ganger, G. R. Perspective: semantic data management for the home. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, USENIX Association (Berkeley, CA, USA, 2009), 167–182.
20. Shao, B., Li, D., and Gu, N. A sequence transformation algorithm for supporting cooperative work on mobile devices. In *CSCW '10: Proceedings of the 2010 ACM conference on Computer supported cooperative work* (Savannah, GA, USA, 2010).
21. Shao, B., Li, D., Lu, T., and Gu, N. An operational transformation based synchronization protocol for web 2.0 applications. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, ACM (New York, NY, USA, 2011), 563–572.
22. Shen, H., and Reilly, M. Personalized multi-user view and content synchronization and retrieval in real-time mobile social software applications. *J. Comput. Syst. Sci. 78*, 4 (2012), 1185–1203.
23. Sun, C., and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Dec. 1998), 59–68.
24. Weiss, S., Urso, P., and Molli, P. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS*, IEEE Computer Society (2009), 404–412.
25. Yang, J., Wang, H., Gu, N., Liu, Y., Wang, C., and Zhang, Q. Lock-free consistency control for web 2.0 applications. In *WWW* (2008), 725–734.
26. Zhao, S., Li, D., Lu, T., and Gu, N. Back to the future: a hybrid approach to transparent sharing of video games over the internet in real time. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, ACM (New York, NY, USA, 2011), 187–196.