An Operational Transformation Based Synchronization Protocol for Web 2.0 Applications

Bin Shao[†], **Du Li**[‡], **Tun Lu**[†], **Ning Gu**[†]
[†] School of Computer Science, Fudan University, Shanghai, China
[‡] Nokia Research Center, Palo Alto, California, USA
binshao@microsoft.com[*], lidu008@gmail.com, {lutun, ninggu}@fudan.edu.cn

ABSTRACT
Current Web 2.0 services are making mass collaboration a reality. Using a Web browser, people can participate in cooperative work anytime, anywhere from any computing device as long as there is an Internet connection. Lying in the heart of some well-known services is an optimistic consistency control technique called operational transformation (OT). This paper proposes TIPS, a novel sync protocol that adapts OT for Web 2.0 applications. Based on a recent theoretical framework called ABT, it ensures not only convergence but also the right object order for linear documents. Designed to address the HTTP style of communication, TIPS allows clients to sync with the server by independent time intervals and dynamically join and leave at any time. When processing do operations, its time complexity is linear in the total number of operations generated by all clients during one server interval and independent of the size of history. TIPS is efficient for supporting a spectrum of (near-)realtime to asynchronous collaboration editing tasks.

Author Keywords
Collaborative Systems, Consistency, Operational Transformation, Synchronization, Online Software Service, Web 2.0

ACM Classification Keywords
H.5.3 Information Systems: Group and Organization Interfaces—*Web-Based Interaction*; C.2.4 Computer Systems Organization: Distributed Systems—*Client/Server*

General Terms
Algorithms, Experimentation, Performance

1. INTRODUCTION

The Web has become the platform of choice for mass communication and collaboration over the past decade. Recent years have witnessed the proliferation of so-called Web 2.0 services such as wikis, weblogs, social networking sites, media sharing sites, and a new generation of Web-based office productivity tools. The value and quality of a service usually increase with the number of prosumers, i.e., users who not only consume but also produce contents.

Among the well-known Web 2.0 services, Google Docs pioneered an industrial trend to provide traditionally prepackaged productivity software as online Web services. Microsoft is making a major endeavor to extend its prosperity on desktop to the Web territory. Using a Web browser, the users can access a software service from any networked computer without worrying about software installation and upgrades; they can participate at any time of convenience without having to stay connected. Cooperative work is one-click away.

To address latencies on the Internet, those services usually replicate the shared data from a server. Some of them (e.g., Google Wave & Docs) choose to use operational transformation (OT) for consistency control. OT has been well studied for 20+ years in the CSCW field [2, 13] and experimented in a wide variety of collaborative applications. [1] As an optimistic technique, OT allows each user to modify any part of the local data replica in a lock-free and nonblocking manner. The data replicas eventually converge after the same set of operations are invoked at all sites. As a result, local responsiveness is not sensitive to network latencies. This property particularly appeals to collaborative applications that run over high-latency networks such as the Internet.

In real-world Web 2.0 apps, collaborating sites generally use HTTP for communication, which is asymmetric and client-initiated. A client must send its updates to the server and request for updates by peer clients from the server using HTTP. This paradigm of communication was originally not designed for supporting realtime collaborative apps, as shown in [1, 3]. Although server push can be emulated by keeping a live client-server connection, if it is broken for some reason, a client will not be able to receive new updates from the server until it re-connects to the server by HTTP. Most existing OT works are proposed for P2P systems (e.g., [2, 14, 13, 11, 4, 6, 16, 5, 9, 8]), with only a few for centralized systems (e.g., [7, 17, 10]). Note that a centralized system may not necessarily be HTTP-based. Although those works could be used in Web 2.0 apps, some adaptation would be necessary to address the above HTTP characteristics.

[*] Bin Shao is currently with Microsoft Research Asia (MSRA).

[1] For examples, see http://www.ntu.edu.sg/home/czsun/

CSCW 2011, March 19–23, 2011, Hangzhou, China.

In this paper, we propose a **t**ransformation and time **i**nterval based **p**rotocol for **s**ynchronization (TIPS) in Web 2.0 apps. Specifically, TIPS has the following two key properties:

1. TIPS is based on a recent theoretical OT framework, called admissibility-based transformation or ABT [5], and its correctness can be formally proved with regard to formalized conditions. It ensures not only convergence but also the right object order when the shared data is linear (or can be mapped to a linear address space [15]).
2. TIPS allows clients to sync with the server in independent and adjustable time intervals. It is efficient in handling long operation sequences that may accumulate when communication cannot occur frequently enough. This is desirable for Web-based apps in which clients communicate with the server periodically as well as mobile apps in which clients work with intermittent connectivity [9].

Our ongoing work is extending TIPS with complementary support for undo. Due to space limits, this paper focuses on do operations while leaving out the correctness proofs. In the next section, we will first survey related works. Then we will present TIPS and involved transformation algorithms, which is followed by a performance evaluation. Finally, we summarize the contributions of this paper.

2. BACKGROUND AND RELATED WORK

To illustrate the basic ideas, consider a system in which two sites start with the same initial data "aver". We model the shared data as a string (of atomic objects) and let the first position be 0. Site 1 changes "aver" to "waver" by operation o_1=*ins*$(0, w)$. Concurrently, site 2 changes "aver" to "ave" by operation o_2=*del*$(3, r)$. When o_2 reaches site 1, if it were executed as-is, the result would go wrong, because the object that o_2 tries to delete is no longer at its original position. To correctly execute o_2, we must shift its position to incorporate the effect of o_1. As a result, o_2 is transformed into o_2'=$del(4, r)$, which will reproduce the intended effect of o_2 in the current state of site 1. On the other hand, when o_1 arrives at site 2, it can be safely executed as-is because the execution of o_2 does not invalidate o_1's position. The two sites eventually converge in the correct state "wave".

2.1 Jupiter OT

Jupiter OT [7] is the first that adapts OT to a client/server architecture in which clients communicate and sync via the server. An extended version of Jupiter OT is applied in Google Wave&Docs for consistency control. It uses a 2-dimensional state space to track the **transformation paths**, i.e., the order in which operations are transformed with each other.

Consider the following well-known scenario [14]. Suppose that the initial data is "X" and three clients (1, 2, and 3) concurrently generate three operations o_1=*ins(1,T)*, o_2=*del(0,X)* and o_3=*ins(0,O)*, respectively. There are six possible transformation paths, two of which are shown in Figure 1.

Figure 1(A) gives the state space on the server when the clients sync with the server in the order of 3, 1, and 2. When client 3 syncs with the server, the server has no operation

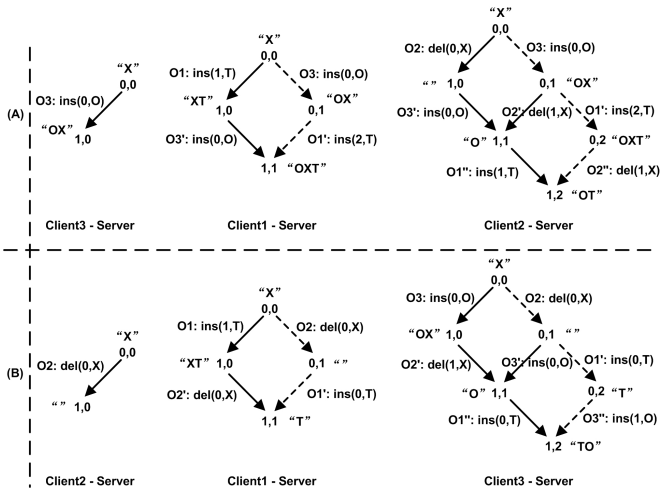

Figure 1. In Jupiter OT, the final state of the shared data depends on the order in which clients sync with the server.

and no transformation is needed. Hence o_3 is executed as-is yielding "OX". Next, when client 1 syncs with the server, o_3 is already there. Hence o_1 is transformed with o_3, resulting o_1'=*ins(2,T)*. Executing o_1' in the current server state yields "OXT". Finally, when client 2 syncs with the server, o_2 must be transformed with o_3 followed by o_1'. That is, the transformation path is $[o_3, o_1']$. The resulting operation o_2''=*del(1,X)* when executed on the server yields state "OT".

Figure 1(B) shows the server state space when the clients sync in the order of 2, 1, and 3. After syncing o_2 and o_1, the state becomes "T". When client 3 syncs, the transformation path is $[o_2, o_1']$, where o_1'=*ins(0,T)*. Hence, o_3 is first transformed with o_2, yielding o_3'=*ins(0,O)*. However, when o_3' is transformed with o_1', their positions tie. Since the site id of client 1 is smaller than that of client 3, Jupiter shifts the position of o_3', yielding $o_3'' = ins(1, O)$ and the final state "TO", which differs from that in Figure 1(A).

From this example, it is clear that Jupiter OT is nondeterministic in that the final data state is dependent on the order in which clients sync with the server. It can achieve convergence by enforcing the same transformation path at all sites. However, there is no guarantee that Jupiter will adopt the "right" transformation paths and yield the "right" final state.

2.2 Other Related OT Works

It is understood that data convergence is not the only goal in collaborative applications and must be constrained [14, 4]. The recent theoretical framework ABT [5] introduces a constraint called *admissibility preservation*, which requires that the invocation of any remote operation not violate the order of objects established by local operations invoked in their generation states. The resulting order between objects is called the **effects relation**, denoted as $\prec$. For example, in the above scenario, when the three operations are originally generated, the effects relation is established as $O \prec X \prec T$. This is due to the fact that 'O' is inserted before 'X' by o_3 and 'T' is inserted after 'X' by o_1. By the admissibility constraint, no matter in which order the operations are trans-

formed, their effects should not violate this relation. Hence, the correct result must be "OT" rather than "TO".

Theoretically, TIPS builds on ABT [5] because its admissibility preservation condition is formalized and subject to formal proofs. Other OT works, including GOT [14], GOTO [13], NICE [10], TIBOT [6], SOCT [11, 17], and COT [16], follow [14] with an alternative constraint called intention preservation. Theoretical comparisons with previous works have been detailed in [5] and will not be repeated here.

TIPS is designed for Web apps in which clients (hosted by browsers) communicate with each other via the server using HTTP strictly following the request/response paradigm. In TIPS, each client sends operations to the server and the server processes operations all by time intervals that are *independent* of each other, which effectively resembles that of typical Web sites today. Hence TIPS is easy to implement and deploy with standard Web technologies, even when the set of participants is unknown *a priori*, dynamic or large. No other OT work directly addresses HTTP as does this work.

The other two aspects of TIPS, linear time complexity and time interval based sync, have appeared separately in previous works. However, TIPS is the first that achieves both and in the context of Web apps. Moreover, its correctness can be formally proved. For space reasons, we only compare TIPS with its two primary sources, ABST [9] and TIBOT [6]. Comparing other works would inevitably involve the theoretical aspect, which is out of scope of this paper.

A few recent OT algorithms (e.g., COT [16] and ABTU [8]) achieve $O(|H|)$ time complexity when integrating one operation into history H, which means $O(|H|\cdot|T|)$ to integrate an operation sequence T. ABST [9] is the latest work optimized for sequences, which takes time $O(|H|+|T|)$ to integrate sequence T. Suppose that n clients each submit a sequence T of the same size at the same time. Even if ABST were run on the server, it would integrate one sequence T into H at a time, effectively totalling $O(n|H|+n^2|T|)$. [2] Nonetheless, the sequence transformation techniques in ABST are extended in TIPS to integrate all those n sequences simultaneously. Its (server) time complexity is $O(logn\cdot n|T|)$, which is independent of the operation history H.

Most existing OT works, including [5, 9], use vector timestamps [2] to determine the concurrent relation between operations. However, they must estimate and assume a maximum number of sites, which incurs considerable overheads in dynamic, opportunistic participation as is expected in Web apps. Although vector timestamps can be compressed [12], the compression complicates the OT algorithm. TIPS does not use vector timestamps. The idea of syncing based on time intervals draws from TIBOT [6]. However, TIBOT assumes a replicated system and its linear logical clocks tick at the same pace at each site. Hence the time intervals are *not independent* in TIBOT. Moreover, TIBOT cannot be formally proven and counterexamples exist [4].

[2]It takes $O(|H|+|T|)$ to sync the 1st T, $O((|H|+|T|)+|T|)$ the 2nd, and so on, totalling $O(\Sigma_{k=1}^{n}(|H|+k|T|))$=$O(n|H|+n^2|T|)$.

3. TIPS: THE SYNC PROTOCOL

We will first present the basic sync protocol that strictly assumes the request/response paradigm of client-server interaction as is typical in current Web environment. Then we will discuss the robustness of this protocol and how to extend it to accommodate the emerging server push paradigm as well as real-time communication and collaboration.

3.1 Overview of Protocol

Conceptually, our system consists of one centralized server and a number of clients, which collaborate to modify a shared data structure. The sync protocol has the following key properties: (1) a client can join/leave a collaborative session at any time, (2) each client independently decides when to sync with the server, and (3) the server independently decides when to process operations received from the clients. These properties make it capable of supporting a wide range of synchronous and asynchronous collaboration tasks.

Figure 2 shows how any client j and the server communicate in TIPS. Each client in our system has a unique id. A client j synchronizes with the server every τ_j time, to submit local operations of this client and to receive remote operations by other clients that have been submitted to the server. Local operations performed during interval τ_j are queued in buffer T_j. Note that the length of τ_j may change over time and may be different across clients.

On the other hand, the server processes operations from clients every τ_s time, the length of which is determined by needs of the collaboration task and may change over time. During every interval τ_s, operations received from client j is queued in buffer RB_j. Operations to be sent to client j are queued in buffer SB_j to be fetched by client j at any time.

Roughly, at the end of each interval τ_s, the server integrates operations in all RB_j's and dispatches the result to all SB_j's. At the end of each interval τ_j, client j fetches operations in SB_j from the server, transforms SB_j with T_j to yield SB'_j and T'_j, respectively, executes SB'_j locally and submits T'_j to the server. Operations in all the buffers, T_j, RB_j and SB_j, are kept in the order of the so-called effects relation $\prec$ for efficiency reasons. We will explain the algorithm details progressively in the following.

3.2 The Core Protocol

Algorithms 1 and 2 specifies the control procedures for client j and the server, respectively. The shared data is replicated at all sites and the server maintains the primary copy. The buffers, T_j, RB_j and SB_j are initially empty sequences. Each site has two threads, which work as follows.

Client: On the side of client j, when a local operation o is generated by the user, the *ERMerge* algorithm is invoked to add o into buffer T_j. As will be explained in the next section, for efficiency reasons, ERMerge adds o into T_j in the effects relation order. The ERMerge algorithm, which merges one operation and a sequence that are contextually serialized, is but a special case of serializedMerge that runs on the server, which merges two contextually serialized sequences.

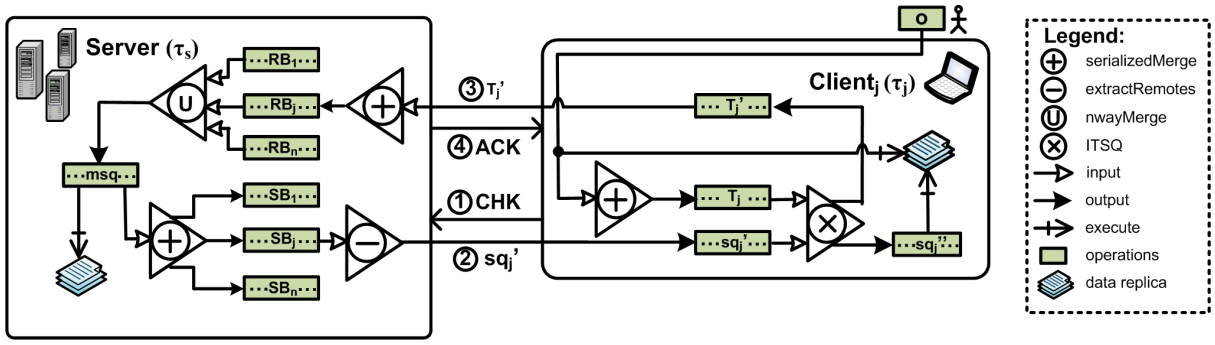

Figure 2. In TIPS, the server and any client j communicate using HTTP, where T_j is the client buffer for local operations from the user, and RB_j and SB_j are server buffers for communicating with client j; every τ_j time, client j requests for remote sq'_j and sends back local T'_j; every τ_s time, the server merges operations from all clients; intervals τ_s and τ_j's are independent of each other. All sequences are in the effects relation order.

Algorithm 1 [C] control procedure at client j (τ_j)

Initialization:
1: $T_j \leftarrow [\,]$;
Thread 1: handle local operation o
2: execute o on local data replica;
3: * $T_j \leftarrow \mathrm{ERMerge}(o, T_j)$;
Thread 2: sync with server every τ_j time
4: send *CHK* request to server; //①
5: receive sequence sq'_j from server; //②
6: * $sq''_j \leftarrow \mathrm{ITSQ}(sq'_j, T_j)$;
7: execute sq''_j on local data replica;
8: * $T'_j \leftarrow \mathrm{ITSQ}(T_j, sq'_j)$;
9: send sequence T'_j to server; //③
10: $T_j \leftarrow [\,]$; //only if T'_j is received by the server ④

Algorithm 2 [S] control procedure at server (τ_s)

Initialization:
1: $\forall i$: $RB_i \leftarrow [\,]$; $SB_i \leftarrow [\,]$;
Thread 1: process request from client j
2: **case** *CHK*: //①
3: * $sq'_j \leftarrow \mathrm{extractRemotes}(SB_j, j)$;
4: respond to client j with sq'_j; //②
5: $SB_j \leftarrow [\,]$; //only if sq'_j is received by j
6: **case** sequence T'_j: //③
7: * $RB_j \leftarrow \mathrm{serializedMerge}(RB_j, T'_j)$;
8: respond to client j with an ACK; //④
Thread 2: merge RB's every τ_s time
9: * $msq \leftarrow \mathrm{nwayMerge}([RB_1, RB_2, \ldots, RB_n])$;
10: $\forall i : RB_i \leftarrow [\,]$;
11: execute msq on the primary data replica;
12: * $\forall i$: $SB_i \leftarrow \mathrm{serializedMerge}(SB_i, msq)$;

Each client j has a configurable time interval τ_j, which controls the frequency in which the client synchronizes with the server. At end of each interval τ_j, client j sends a *CHK* request to the server to check whether there are new remote operations available. The interval τ_j is decided by each client j independently of other clients and the server. Hence the client code is parameterized by τ_j.

In response to the *CHK* request, client j will receive a sequence sq'_j of remote operations from the server. The client first calls the *ITSQ* algorithm to transform sq'_j with T_j, yielding sq''_j, and executes sq''_j on the local data replica. Then the client calls *ITSQ* to transform T_j with the original sq'_j, yielding T'_j, and sends T'_j to the server. Finally, the local buffer T_j is reset. In the case that sq'_j is empty, T_j will be sent as-is; and in the case that T_j is empty, sq'_j will be executed as-is. As will be explained in the next section, algorithm *ITSQ* transforms two contextually equivalent sequences to incorporate the effects of one sequence into the other.

Server: On the server side, when a *CHK* request is received from client j, it first calls algorithm *extractRemotes* to extract subsequence sq'_j from SB_j, then responds to client j with sq'_j and resets SB_j. As will be explained in the next section, algorithm *extractRemotes* extracts all operations in SB_j that are generated by clients other than j. For networking and computation efficiency, there is no need to send operations generated by client j back to itself.

On the other hand, when the server receives a sequence T'_j from client j, it calls algorithm *serializedMerge* to merge T'_j into current receiving buffer RB_j for client j. As will be explained, algorithm *serializedMerge* merges two contextually serialized sequences and keeps the resulting sequence in the effects relation order. This way operations received from client j are accumulated in RB_j until the end of interval τ_s.

The other thread of the server runs by a configurable interval τ_s. Every τ_s time it performs an n-way merge to synthesize operation sequences in all the n receiving buffers, where n is the number of clients. As will be explained, algorithm *nwayMerge* merges n contextually equivalent sequences and keeps the output sequence in the effects relation order. If no operations are received from client j during τ_s, then RB_j is an empty sequence. The n-way merge will proceed as scheduled with an empty RB_j.

After the n-way merge, the server resets all the RB's and executes the merged sequence msq on the primary data replica. Then msq is merged into the sending buffer SB_j for every client j by calling algorithm *serializedMerge*. Hence opera-

tions in SB_j accumulate until the entire buffer is fetched by client j. If client j checks more frequently than τ_s, buffer SB_j could be an empty sequence.

3.3 Discussions: Robustness and Extensions

The core protocol specified above can be augmented to handle issues such as dynamic joining, client or network failures, real-time communication, and server push.

Dynamic Participation: When a new client j joins, the server only needs to allocate a pair of buffers RB_j and SB_j. At the same time, the client receives a copy of the data and starts to work on the local replica. When a client j leaves the session normally, it sends a *CHK* request for remote operations SB_j dangling on the server and flushes its remaining local operations in T_j to the server. The leaving is indicated so that RB_j and SB_j are recollected on the server.

Client/Network Failures: Between the times a client j joins and leaves a session, the client periodically sends a *CHK* request to the server to fetch new updates. This can be extended with a more frequent, negotiated liveness report in the case that the interval τ_j is long enough. If the server fails to receive some number of liveness reports from a client in a row, it assumes that the client has crashed or left abnormally and recollects (or swaps out) the buffers RB_j and SB_j.

When a client crashes, it may lose its copy of the shared data and unsubmitted operations. In this case, the client needs to rejoin the session to fetch the latest copy of the shared data. If a client encounters a temporary network failure and the server has not recollected its operation buffers when the client reconnects, a *CHK* request will resync the client as usual. However, the network failure may be so long that the server has already recollected its buffers when the client reconnects. Resync is still possible with a *CHK* request, provided that the server saves all integrated sequences.

We assume that the server does not crash, which is reasonable because the shared data and operations can be stored in a database. If a server is temporarily unavailable, it can be treated similarly on the clients as a network failure. On a modern Web client that implements HTML version 5 or above, we can also make the data replica, operations and other key data structures persistent in the local storage. This client-side persistence eases client failure recovery and mitigates loss of data (e.g., uncommitted operations) when failures occur. Then it is possible to achieve resync by a usual *CHK* request after a client crash or a long network failure.

Real-Time Collaboration: By choosing the right values for τ_j and τ_s in a collaborative application, we can seek a balance between awareness (i.e., the time for users to see each other's actions) and costs (e.g., client and server CPU utilization, network traffic, and energy consumption on mobile clients). In general, the shorter the intervals, the more timely the awareness, and the more expensive the costs. Those costs could translate into slow system responsiveness, battery drainage and bloated bills for data communication, especially when mobile devices are used [3].

In real-time applications that require to replay every remote operation, e.g., as in Google Wave, it is trickier than just reducing the intervals. Under the classical request/response paradigm of the Web today, all client-server communications are ultimately initiated by the clients. To maintain a FIFO channel between a client and the server, the best we can do is probably to send one operation, wait for its response, then immediately send the next operation, and so forth. Due to the high frequency in communication, the number of operations will be very small for each interval and transformation time negligible. As a result, the timeliness of awareness is effectively determined by the HTTP round-trip time.

Although currently emulated, **server push** can be a more efficient approach to implementing real-time communication and collaboration on the Web. When the server is able to proactively push new operations to the clients, TIPS can be conceptually simplified, which is sketched as follows.

Server: Every τ_s time the server broadcasts a *SYNC* message to all clients asking for new operations. Once all operations are received, the server performs the n-way merge and then broadcasts merged sequence to all clients.

Client: After receiving the *SYNC* message from the server, each client sends the newly-generated operation sequence (empty if none) to the server. When the merged sequence is received from the server, the client extracts the remote subsequence and transforms it with new local operations. Then, the transformed remote operations are executed locally. Note that there are no client intervals.

4. TRANSFORMATION ALGORITHMS

In this section, we present the algorithms used in TIPS. We will first introduce some useful notations and utility functions (Table 1) and then discuss the transformation functions.

4.1 Notations

To simplify presentation, we model the shared data as a linear string. Two primitive operations are considered: $ins(pos, char)$ and $del(pos, char)$. Given any operation o, notation $o.type$ stands for its type (ins or del), notation $o.pos$ is the operation position, and notation $o.char$ is the effect character o inserts or deletes. If o is an insert operation, we use $o.id$ to denote the id of the site that generates o. If o is a delete operation, we instead use notation $o.ids$ to denote the set of sites that delete the same character concurrently; these deletions are combined during transformation.

The position of any operation o is always defined in some state, denoted as $dst(o)$. As is conventional [13], if $dst(o_1) = dst(o_2)$, they are contextually equivalent, denoted as $o_1 \sqcup o_2$; if o_2 is defined in the state resulted from executing o_1, then o_1 and o_2 are contextually serialized, denoted as $o_1 \mapsto o_2$. Two operations o_1 and o_2 are contextually comparable, denoted as $o_1 \bowtie o_2$, if one of the three conditions holds: (1) $o_1 \sqcup o_2$; (2) $o_1 \mapsto o_2$; or 3) $o_2 \mapsto o_1$.

An operation sequence sq is a list of operations, in which any two adjacent operations are contextually serialized. Given a

non-empty sequence sq, its definition state is given by that of its first operation, i.e., $dst(sq) = dst(sq[0])$. Any two sequences sq_1 and sq_2 are contextually equivalent, denoted as $sq_1 \sqcup sq_2$, if $dst(sq_1) = dst(sq_2)$; they are contextually serialized, denoted as $sq_1 \mapsto sq_2$, if the last operation in sq_1 and the first operation in sq_2 are contextually serialized. Let $s \circ sq$ denote the state after executing sq in state $dst(sq)$=s. Given any two sequences sq_1 and sq_2, we say that sq_1 is effects equivalent to sq_2, denoted as $sq_1 \cong sq_2$, if $dst(sq_1) = dst(sq_2) = s$ and $s \circ sq_1 = s \circ sq_2$.

To make transformations more efficient, we keep operations in sequences according to the relative position of their effect characters, which is called the **effects relation** $\prec$. For example, suppose that a user issues a sequence of four operations successively, sq = [ins(2,x), del(1,b), ins(4,y), del(2,c)], changing string "abcd" into "axdy". After reordering these operations by the $\prec$ order, the resulting sequence will be sq'=[del(1,b), ins(1,x), del(2,c), ins(3,y)]. The two sequences are effects equivalent, i.e., $sq \cong sq'$, because applying sq' to the initial string will also yield "axdy".

Note that there is no extra step (or cost) for reordering sequences in the presented algorithms. Also note that a sequence could mix insertions and deletions, which differs from sequences in ABT [5] and ABST [9] that separate insertions and deletions. However, this mixing requires special handling of operations that have the same effect characters.

4.2 Some Utility Functions

Given two operations o_1 and o_2, we use notation $o_1 \equiv o_2$ to denote the case in which o_1 and o_2 have the same effect character. There are two interesting cases: (1) o_2 deletes the effect character inserted by o_1, denoted as $o_1 \overset{id}{\equiv} o_2$; and (2) they delete the same character, denoted as $o_1 \overset{dd}{\equiv} o_2$. We define function *ideq*(o_1, o_2) to determine whether or not $o_1 \overset{id}{\equiv} o_2$ holds, which returns *true* only if $o_1 \mapsto o_2$, $o_1.type = ins$, $o_2.type = del$ and $o_1.pos = o_2.pos$. We define function *ddeq*(o_1, o_2) to determine whether or not $o_1 \overset{dd}{\equiv} o_2$ holds, which returns *true* only if $o_1 \sqcup o_2$, $o_1.type = o_2.type = del$ and $o_1.pos = o_2.pos$.

Now we explain how to determine the $\prec$ order between two operations o_1 and o_2. Given $o_1 \mapsto o_2$, function *isprecA*(o_1,o_2) determines whether or not $o_1 \prec o_2$, which returns *true* if (1) $o_1.pos < o_2.pos$, or (2) $o_1.pos = o_2.pos$ and $o_1.type = del$. The precondition $o_1 \mapsto o_2$ means that o_2 is executed immediately after o_1. In case (2), for example, consider state "ab" and o_1 deletes 'a', yielding "b", and o_2 deletes 'b'; if o_2 inserts 'c' before 'b', we mandate $o_1 \prec o_2$ by policy.

Given that $o_1 \sqcup o_2$, function *isprecB*(o_1,o_2) determines whether or not $o_1 \prec o_2$, which returns *true* only if (1) $o_1.pos < o_2.pos$, or (2) $o_1.pos = o_2.pos$, $o_1.type = ins$, $o_2.type = del$, or (3) $o_1.pos = o_2.pos$, $o_1.type = o_2.type = ins$, $o_1.id < o_2.id$. The precondition means that the two operations are defined in the same state. In the third case, two operations try to insert at the same position. We compare their site ids to break the tie. It is worth noting that this tie-breaking policy does not always work correctly. Its correctness relies on the control procedure to ensure the sufficient conditions under which it works, which is elaborated in [5].

Table 1. A summary of notations and utility functions.

Notation	Brief Description
$o_1 \prec o_2$	$o_1.char$ precedes $o_2.char$ in some state
$o_1 \sqcup o_2$	o_1 and o_2 are contextually equivalent
$o_1 \mapsto o_2$	o_1 and o_2 are contextually serialized
$o_1 \bowtie o_2$	o_1 and o_2 are contextually comparable
$sq_1 \cong sq_2$	sq_1 and sq_2 are effects equivalent
ideq(o_1, o_2)	$o_1 \mapsto o_2$ and o_2 deletes what o_1 inserts
ddeq(o_1, o_2)	$o_1 \sqcup o_2$ and both delete the same character
isprecA(o_1, o_2)	$o_1 \mapsto o_2$ and $o_1 \prec o_2$
isprecB(o_1, o_2)	$o_1 \sqcup o_2$ and $o_1 \prec o_2$

4.3 Transformation Functions

Algorithm 3 [C] *ERMerge*(o, T): T'

```
1: T' ← T; hit ← |T'|
2: δ ← (o.type = ins) ? 1 : -1
3: for (i ← |T'| − 1; i ≥ 0; i−−) do
4:     if ideq(T'[i], o) then
5:         remove T'[i] from T'; return T'
6:     else if isprecA(T'[i], o) then
7:         break          // with hit = i + 1
8:     else
9:         T'[i].pos ← T'[i].pos + δ
10:        hit ← i
11:    end if
12: end for
13: insert o into T' at position hit
14: return T'
```

ERMerge: Algorithm 3 is called when a new local operation o is generated, it adds o to the local operation buffer T. Since o is generated in the current state, $T \mapsto o$ and o can be directly appended to T. In order to keep T in the $\prec$ order, however, we swap o with the operations in T from right to left until the appropriate position is found according to the $\prec$ order. By **swap**, we exchange the execution order of two contextually serialized operations. For every $T[i]$ in question, we have $T[i] \mapsto o$. If $T[i] \overset{id}{\equiv} o$, the effect of o cancels that of $T[i]$ and neither of them will be included in the resulting sequence. By function *isprecA*, if $T[i] \prec o$, we have found the right position to insert o and the swap process stops. Otherwise, it means $o \prec T[i]$ and we need to swap $T[i]$ and o such that $o' \mapsto T'[i]$ holds in the result. Since $o \prec T[i]$, the position of $T[i]$ must be shifted to incorporate the effect of o depending on its type (line 9).

serializedMerge: Algorithm 4 generalizes ERMerge to merge two ordered sequences, sq_1 and sq_2, that are contextually serialized ($sq_1 \mapsto sq_2$) and keeps the resulting sequence sq ordered by the effects relation. Its working structurally resembles that of the classical 2-way merge algorithm.

Let o_1 be any operation in sq_1 and o_2 be any operation in sq_2. To decide their $\prec$ order, we transpose o_1 to the end of

Algorithm 4 [S] serializedMerge$(sq_1, sq_2) : sq$

```
1: sq ← [ ]; i ← 0; j ← 0; Δ ← 0
2: while (i < |sq2|) and (j < |sq1|) do
3:     alias o1 = sq1[j]
4:     alias o2 = sq2[i]
5:     o2.pos ← o2.pos − Δ
6:     if ideq(o1,o2) then
7:         Δ ← Δ − 1; i ← i + 1; j ← j + 1
8:     else if isprecA(o1,o2) then
9:         o1.pos ← o1.pos + Δ
10:        sq ← sq · o1;  j ← j + 1
11:    else
12:        sq ← sq · o2
13:        δ ← (o2.type = ins) ? 1 : −1
14:        Δ ← Δ + δ;  i ← i + 1
15:    end if
16: end while
17: for (;i < |sq2|;i++) do
18:    sq ← sq · sq2[i]
19: end for
20: for (;j < |sq1|;j++) do
21:    sq1[j].pos ← sq1[j].pos + Δ
22:    sq ← sq · sq1[j]
23: end for
24: return sq
```

sq_1 and transpose o_2 to the head of sq_2. Since sq_1 is ordered by relation $\prec$, swapping o_1 and operations with effects following o_1 will not affect the position of o_1. Hence, after the transposition, $o_1.pos$ remains as-is. However, swapping o_2 and operations with effects preceding o_2 does affect the position of o_2. After the transposition, $o_2.pos$ will be adjusted by Δ, which is the number of insertions that precede o_2 subtracting the number of deletions that precede o_2. Because $sq_1 \mapsto sq_2$, after the transposition, $o_1 \mapsto o_2$ holds.

Now $o_1 \mapsto o_2$ and we can determine their $\prec$ order. In the case of $o_1 \overset{id}{\equiv} o_2$ (line 6), o_2 cancels the effect of o_1 and none of them will be included in the result sq. However, note that Δ is adjusted for next operations in sq_2 because this o_2 is a delete. By function $isprecA(o_1, o_2)$, if $o_1 \prec o_2$, we add o_1 into sq before o_2, after including the effects of operations in sq_2 that precedes o_1 by the $\prec$ order (line 9). Otherwise, o_2 will be added into sq before o_1 (line 12); and Δ will be adjusted according to the type of this o_2 (lines 13–14).

The two for-loops (lines 17-19 and lines 20-23) handle the cases when sq_2 and sq_1 are still not exhausted, respectively. Note that these two cases are mutually exclusive. Hence, we either append the remainder of sq_2 to sq, or append the remainder of sq_1 to sq after including the effects of sq_2.

nwayMerge: Algorithm 5 is called by the server side of the protocol (Algorithm 2) to perform n-way merge of n ordered sequences that are contextually equivalent. The input is a list of sequences. It calls the 2-way merge function *mergeSQ* to merge two sequences at a time. For example, given 5 sequences $sq_0, sq_1, sq_2, sq_3, sq_4$, it first merges sq_3 into sq_0 and sq_4 into sq_1. Then, with new sequences sq_0, sq_1, sq_2, it merges sq_2 into sq_0. Finally, in the remaining sq_0, sq_1, it merges sq_1 into sq_0 and returns resulting sq_0. Each pass roughly halves the number of sequences for the next pass.

Algorithm 5 [S] nwayMerge$(sqlist) : sq$

```
1: L ← sqlist;   N ← |L|
2: while N > 1 do
3:     N, M ← ⌈N/2⌉, ⌊N/2⌋
4:     for (i ← 0; i < M; i++) do
5:         L[i] ← mergeSQ(L[i], L[N+i])
6:     end for
7: end while
8: return (N==1) ? L[0] : []
```

Algorithm 6 [S] mergeSQ$(sq_1, sq_2) : sq$

```
1: sq ← [ ]; i ← 0; j ← 0; Δ1 ← 0; Δ2 ← 0
2: while (i < |sq2|) and (j < |sq1|) do
3:     alias o1 = sq1[j]; o1.pos ← o1.pos − Δ1
4:     alias o2 = sq2[i]; o2.pos ← o2.pos − Δ2
5:     if ddeq(o1,o2) then
6:         o1.pos ← o1.pos + Δ2
7:         o1.ids ← o1.ids ∪ o2.ids
8:         sq ← sq · o1; i ← i + 1; j ← j + 1
9:     else if isprecB(o1,o2) then
10:        o1.pos ← o1.pos + Δ2
11:        sq ← sq · o1
12:        δ ← (o1.type = ins) ? 1 : −1
13:        Δ1 ← Δ1 + δ; j ← j + 1
14:    else
15:        o2.pos ← o2.pos + Δ1
16:        sq ← sq · o2
17:        δ ← (o2.type = ins) ? 1 : −1
18:        Δ2 ← Δ2 + δ; i ← i + 1
19:    end if
20: end while
21: for (;i < |sq2|;i++) do
22:    sq2[i].pos ← sq2[i].pos + Δ1
23:    sq ← sq · sq2[i]
24: end for
25: for (;j < |sq1|;j++) do
26:    sq1[j].pos ← sq1[j].pos + Δ2
27:    sq ← sq · sq1[j]
28: end for
29: return sq
```

mergeSQ: Given two ordered sequences sq_1 and sq_2, where $sq_1 \sqcup sq_2$, Algorithm 6 merges them into an ordered sequence sq. Similarly to *serializedMerge*, we consider any operation o_1 in sq_1 and any operation o_2 in sq_2. To determine their $\prec$ order, we first ensure $o_1 \sqcup o_2$ by transposing o_1 and o_2 to the heads of sq_1 and sq_2, respectively. The transposition requires to exclude the effects of those operations in sq_1 or sq_2 that precede o_1 or o_2 by the $\prec$ order (lines 3–4).

Given that $o_1 \sqcup o_2$, if $o_1 \overset{dd}{\equiv} o_2$, meaning that they delete the same character, we combine o_1 and o_2 into one operation whose ids is the union of $o_1.ids$ and $o_2.ids$ (line 6–8).

Now since $o_1 \sqcup o_2$, we call function *isprecB*(o_1, o_2) to determine their $\prec$ order. If $o_1 \prec o_2$, we add o_1 before o_2 in the result sq, after shifting $o_1.pos$ to include the effects of operations in sq_2 that precede o_1 by the $\prec$ order (line 10 to 13). Otherwise, if $o_2 \prec o_1$, similarly o_2 is added before o_1 after including the effects of operations in sq_1 that precede o_2 by the $\prec$ order (lines 15–18).

Algorithm 7 [S] extractRemotes$(sq, j) : sq'$

```
1: sq' ← [ ]
2: for (i ← 0; i < |sq|; i++) do
3:     if (j ≠ sq[i].id) or (j ∉ sq[i].ids) then
4:         sq' ← sq' · sq[i]
5:     end if
6: end for
7: return sq'
```

extractRemotes: Algorithm 7 is called by the server to extract remote operations on behalf of a given client j. The input sequence sq is a result of the n-way merge algorithm, which includes operations from all sites that are generated during the previous server interval τ_s. Consider that the current buffer T_j of client j has not been synced with the server yet. Some of those operations in sq are generated by client j, which happen before T_j, while the others are by other clients, which are concurrent with T_j. The purpose is to extract a subsequence sq' from the ordered sequence sq such that sq' is also ordered and contextually equivalent with T_j.

What we need to do is to transpose sq into two ordered subsequences sq_h and sq_c such that $sq_h \mapsto sq_c$, where operations in sq_h are all generated by client j and sq_c by other clients. Because $sq_h \mapsto T_j$, then $sq_c \sqcup T_j$ must hold and hence sq_c is what we want. Now the problem is reduced to finding operations in sq that are generated by client j, which is solved as follows: We scan sq from left to right, for each $sq[i]$, if it is not generated by client j, append it to sq_c; otherwise, transpose operation $sq[i]$ with sequence sq_c, yielding $sq[i]'$, and then append $sq[i]'$ to sq_h. This would take $O(|sq|^2)$ time. However, considering that sq is already ordered by relation $\prec$, transposing $sq[i]$ and sq_c does not affect operations in sq_c. Also considering that we do not really need sq_h, we can just pick from sq those not generated by client j and discard those that are. This is exactly how Algorithm 7 works, with complexity $O(|sq|)$.

ITSQ: Given two ordered sequences sq_1 and sq_2, where $sq_1 \sqcup sq_2$, Algorithm 8 inclusively transforms sq_1 with sq_2, yielding sq_1' such that $sq_2 \mapsto sq_1'$. Similarly to Algorithm 6, for any operation o_1 in sq_1 and any operation o_2 in sq_2, we transpose them to the heads of the two sequences, respectively. Then $o_1 \sqcup o_2$ holds and we can determine their $\prec$ order by function $isprecB(o_1, o_2)$. Note that ITSQ is for inclusion transformation (IT) [13] rather than merging. That is, for example, when o_1 is added into the resulting sequence sq_1', it will have incorporated the effects of all operations in sq_2 that precede o_1 by the $\prec$ order. As a result, o_1 is included in sq_1' possibly with its position value shifted by those operations in sq_2; however, none of the operations of sq_2 is included into sq_1'. In the case that $o_1 \overset{dd}{\equiv} o_2$ (line 5), o_1 will be transformed into an identity operation ϕ and be removed from sq_1' (line 19) since its effect is already in sq_2.

Algorithm 8 [C] ITSQ$(sq_1, sq_2) : sq_1'$

```
1: sq1' ← sq1; i ← 0; j ← 0; Δ1 ← 0; Δ2 ← 0
2: while (i < |sq2|) and (j < |sq1|) do
3:     alias o1 = sq1[j]; o1.pos ← o1.pos − Δ1
4:     alias o2 = sq2[i]; o2.pos ← o2.pos − Δ2
5:     if ddeq(o1,o2) then
6:         sq1'[j] ← φ; i ← i + 1; j ← j + 1
7:     else if isprecB(o1,o2) then
8:         sq1'[j].pos ← sq1'[j].pos + Δ2
9:         δ ← (sq1[j].type = ins) ? 1 : −1
10:        Δ1 ← Δ1 + δ; j ← j + 1
11:    else
12:        δ ← (sq2[i].type = ins) ? 1 : −1
13:        Δ2 ← Δ2 + δ; i ← i + 1
14:    end if
15: end while
16: for (;j < |sq1|;j++) do
17:     sq1'[j].pos ← sq1'[j].pos + Δ2
18: end for
19: remove the φ operation from sq1'
20: return sq1'
```

5. COMPLEXITIES AND EXPERIMENTS

We choose not to experimentally compare the performance with other OT works for two reasons: First, fully implementing other works is very difficult, if not impossible. Some would require substantial extensions to achieve the same properties, while some others are not published with full details. Voluntarily offering such extensions and details could be an error-prone and unwelcome effort itself. Secondly, our purpose is to measure the performance of our own work, not to claim that our work performs better than the others.

5.1 Asymptotic Complexities

The space complexity of client j is linear in the size of the local operation buffer and the remote sequence received from the server at each sync, i.e., $O(|T_j| + |sq_j'|)$. The space complexity of the server is linear in the size of the buffers allocated for all n clients, i.e., $O(\Sigma_{j=1}^{n}(|RB_j| + |SB_j|))$. The time complexities of all the transformation functions are linear in the size of the input sequence(s). The complexity of nwayMerge is $O(\log n \cdot \Sigma_{j=1}^{n}|RB_j|)$, which is roughly linear in the size of the receiving buffers when n is not too large.

5.2 Performance Experiments

Rationale: As shown in Figure 2, the runtime performance of TIPS has a client side and a server side. The server-side performance has two aspects: (1) The time to process a client request: (a) The time for extractRemotes(SB_j, j) to serve *CHK*. (b) The time for serializedMerge(RB_j, T_j') to merge a client sequence T_j' to its receiving buffer RB_j. (2) The time to synthesize all the RB's accumulated in one server interval, including the time for performing nwayMerge (RB_1,

$\ldots, RB_n$) and n calls to algorithm serializedMerge (SB_j, msq), where msq is the result of nwayMerge.

The client-side performance has two aspects: (1) The **user** perceived local response time, which is the time for ERMerge (o, T_j) given a locally generated operation o and local buffer T_j. (2) The **client** perceived system response time, i.e., the time to sync with the server, which consists of two HTTP requests (in the background): (a) One to request for the remote sequence sq'_j, which includes an HTTP RTT, the server time for extractRemotes(SB_j, j) and the client time for two symmetric ITSQ(sq'_j, T_j) and ITSQ(T_j, sq'_j). (b) The other request to send T'_j to the server, which includes a RTT and the server time for serializedMerge(RB_j, T'_j).

The HTTP RTT is mainly contributed by the Web and the networking infrastructure. Hence we only measure contributions by our OT work, as asterisked in Algorithms 1 and 2.

Setup: From the above analyses, the performance of TIPS ultimately depends on the average size of each T_j and the number of clients n. In a (near-)realtime collaborative editing application, a distributed group of users concurrently modify a shared document. We can assume that the client interval τ_j and server interval τ_s are short and hence the number of operations generated by a user in τ_j is small. We can also assume that the number of clients is not too large. Hence we set the two parameters, $|T_j|$ and n, both to range from 10 to 100. While these numbers are large enough to show the performance trends, we expect that the number of active participants in a typical session be 10 or less.

In all the experiments, we generate the operations as follows: At one extreme, when $|T_j|$=100 and n=100, there are 10,000 operations in total. Hence we use a shared document with an initial size of 1,000,000 characters. The first operation is generated with its position uniformly distributed over the current document, then the second operation is generated similarly after the first one is executed, and so forth. The ratio of insertions is 50%. In addition, we avoid generating operations that have the same effect characters so that no operations will be combined or removed from the sequences during transformation. Each experiment is repeated 10 times and the average execution time is recorded.

Client Side Experiments: The client side experiments are performed on a laptop with a 1.50 GHz Intel Pentium M processor and 1 GB DDR memory running Windows XP Professional. The client algorithms (e.g., ERMerge and ITSQ) are implemented using Javascript. The experiments are run on the Google Chrome browser v5.0.375.99.

Even when $|T|$ goes up to 10,000, the average execution time of ERMerge(o, T) only takes 0.6ms. Some selected data points of the symmetric ITSQ in Algorithm 1 are shown in Table 2. Note that the 0ms's should be read as "negligible".

Server Side Experiments: We use a Gentoo Linux server (kernel 2.6.30) with a 2.2 GHz Dual Core AMD Opteron 275 Processor and 2 GB Registered ECC memory. The server side algorithms (e.g., extractRemotes, serializedMerge and nwayMerge) are implemented using C++ and compiled by g++ v4.3.4 with option -O3.

Table 2. Client time to do symmetric ITSQ

n	$\|T\|$	$\|sq\|$	ITSQ(sq,T)	ITSQ(T,sq)
10	10	90	0.8ms	0ms
10	50	450	3ms	0ms
10	100	900	8.8ms	0.2ms
100	10	990	9ms	0.8ms
100	50	4950	12ms	0.8ms
100	100	9900	18.2ms	1.2ms

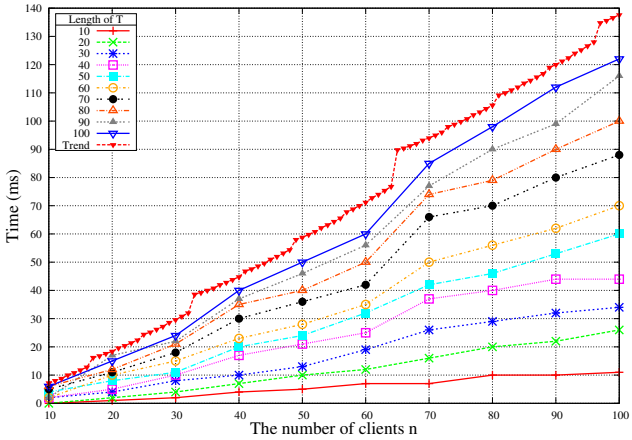

Figure 3. Server time to merge n sequences each of size $|T|$.

When $|sq| = 10,000$, extractRemotes(sq, j) takes 6ms to extract the 9,900 remote operations for client j. Function serializedMerge is used for merging T_j into RB_j and for merging msq into SB_j. In the former, the two input sequences are normally small and the time is negligible. In the latter, the overhead is negligible unless SB_j is huge. When client j crashes or $\tau_j \gg \tau_s$ and all other clients are active, SB_j builds up and it could take a few hundred milliseconds. However, as will be explained, the cost can be eliminated by slightly adjusting the protocol. The measured time for nwayMerge under different n and $|T|$ is plotted in Figure 3.

5.3 Discussions: Results and Next Steps

Client Side: The user-perceived local response delay is generally negligible because it takes less than 1ms even when merging a local operation into a very large T. Even if a system has 100 concurrent clients each submitting 100 operations every interval, the total transformation overheads are below 20ms, which are also negligible. The client-perceived sync delay is dominated by the two HTTP RTTs.

Server Side: The nwayMerge dominates the server side computation. The experimental data shows that the performance is good enough to support near-realtime Web 2.0 applications. In such applications, $|T|$ is usually small (say 10) and clients may sync around the same time. Then, even with 100 active concurrent clients, it takes only 11ms to do nwayMerge, which are no longer than a typical HTTP RTT.

Further Considerations: In practical situations, clients may submit sequences with different lengths to the server. Algorithm nwayMerge can be further optimized, e.g., by merging shorter sequences first in each iteration.

On a more powerful server, the nwayMerge algorithm could be parallelized. Similarly, the n calls to serializedMerge can also be parallelized. Hence it is reasonable to only count the serializedMerge time once in our experiments.

In Algorithm 2, the call to extractRemotes(SB_j, j) could be moved to right before serializedMerge(SB_j, msq) such that, by extractRemotes(msq,j), those not generated by j are extracted from msq and merged into SB_j. Then SB_j can be directly sent to client j in response to its CHK request.

Some of the computation could be offloaded from the server. In particular, when SB_j grows long, a new msq can be simply appended to SB_j to avoid the costs of serializedMerge. The sequences can be compressed and transmitted to the client and transformed with the local T_j one by one. Then extractRemotes can be done on the client or the server.

The above experiments focus on near-realtime collaboration. As it goes asynchronous, the participants will less likely sync with the server around the same time. Then each time nwayMerge effectively only needs to merge few sequences in time linear in their total size. Prior work [9] provides complementary performance studies for asynchronous cases.

6. CONCLUSIONS

This paper proposes TIPS, a novel OT-based sync protocol for Web 2.0 applications, with three major contributions:

1. TIPS ensures data *convergence* as well as the right *order* of objects in a *client/server* architecture. These three properties are not achieved together in previous OT works, to the best of our knowledge. Its correctness is based on a recent theoretical framework and has been formally proven, which will be presented in the journal version.
2. TIPS adapts OT for Web-based collaborative apps. Clients sync with the server by time intervals and any client can join, leave and sync independently of each other, a property not seen in previous OT works. The protocol strictly follows the standard request/response paradigm in HTTP and is easy to deploy with current Web technologies.
3. The time complexity of TIPS is roughly linear in the number of operations generated by all clients during one server interval and independent of the operation history, whereas all other OT works are dependent on the history. The protocol is efficient for supporting a spectrum of near-realtime to asynchronous collaborative editing tasks.

ACKNOWLEDGMENTS

The authors thank the expert referees for their constructive reviews. The work was supported in part by NSF of China (NSFC) under Grants 60736020 and 60803118, and the Shanghai Leading Academic Discipline Project under Grant B114.

REFERENCES

1. R. Bentley, T. Horstmann, and J. Trevor. The World Wide Web as enabling technology for CSCW: The case of BSCW. *JCSCW*, 6(2-3):111–134, June 1997.
2. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD 1989*, pages 399–407.
3. D. Li and M. Anand. MaJaB: Improving resource management for web-based applications on mobile devices. In *MobiSys 2009*, pages 95–108.
4. D. Li and R. Li. Preserving operation effects relation in group editors. In *CSCW 2004*, pages 457–466.
5. D. Li and R. Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work*, 19(1):1–43, 2010.
6. R. Li, D. Li, and C. Sun. A time interval based consistency control algorithm for interactive groupware applications. *ICPADS 2004*, pages 429–436.
7. D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST 1995*, pages 111–120.
8. B. Shao, D. Li, and N. Gu. An algorithm for selective undo of any operation in collaborative applications. In *GROUP 2010*, pages 131–140.
9. B. Shao, D. Li, and N. Gu. A sequence transformation algorithm for supporting cooperative work on mobile devices. In *CSCW 2010*, pages 159–168.
10. H. Shen and C. Sun. Flexible notification for collaborative systems. In *CSCW 2002*, pages 77–86.
11. M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *ICDE 1998*, pages 36–45.
12. C. Sun and W. Cai. Capturing causality by compressed vector clock in real-time group editors. *IPDPS 2002*, pages 59–66.
13. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW 1998*, pages 59–68.
14. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM ToCHI*, 5(1):63–108, Mar. 1998.
15. C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM ToCHI*, 13(4):531–582, 2006.
16. D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE TPDS 2009*, 20(10):1454–1470.
17. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *CSCW 2000*, pages 171–180.